Distributed Systems

Distributed Systems

design and algorithms

Edited by
Serge Haddad
Fabrice Kordon
Laurent Pautet
Laure Petrucci

ISTE

WILEY

First published 2011 in Great Britain and the United States by ISTE Ltd and John Wiley & Sons, Inc.

ISTE Ltd
27-37 St George's Road
London SW19 4EU
UK

www.iste.co.uk

John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030
USA

www.wiley.com

Library of Congress Cataloging-in-Publication Data

Distributed systems : design and algorithms / edited by Serge Haddad ... [et al.].
p. cm.
Includes bibliographical references and index.
ISBN 978-1-84821-250-3
1. Electronic data processing--Distributed processing. 2. Peer-to-peer architecture (Computer networks) 3. Computer algorithms. 4. Embedded computer systems. 5. Real-time data processing. I. Haddad, Serge.
QA76.9.D5D6144 2011
004'.33--dc22

2011012243

British Library Cataloguing-in-Publication Data
A CIP record for this book is available from the British Library
ISBN 978-1-84821-250-3

Printed and bound in Great Britain by CPI Antony Rowe, Chippenham and Eastbourne.

MIX
Paper from responsible sources
FSC® C013604
FSC
www.fsc.org

Contents

Foreword

It is hard to imagine today a single computation that does not rely on at least one distributed system directly or indirectly. It could be a distributed file system, a distributed database, a content distribution network, a peer-to-peer game, a remote malware detection service, a sensor network, or any other distributed computation. Distributed systems have become the equivalent of economic globalization in the world of computing. Adopted for economic reasons, powered by highly efficient and ubiquitous networking, distributed systems define the default architecture for almost every computing infrastructure in use today.

Over the last two decades, distributed systems have taken many shapes and forms. Clusters of computers were among the earliest generations of distributed systems, whose goal was to provide a cost-effective alternative to highly expensive parallel machines. File servers were first to evolve from the cluster-based distributed system model to serve an increasing hunger for storage. The World Wide Web introduced the web server and, with it, the client-server distributed system model, on which millions of other Internet services have been built. Peer-to-peer systems appeared as an "anti-globalization movement", in fact an anti-corporate globalization movement that fought against the monopoly of the service provider in the client-server model. Cloud computing turned distributed systems into a utility that offers computing and storage as services over the Internet. One of the emerging and least expected beneficiaries of cloud computing will be the mobile world of smart phones and personal devices, whose resource limitation can be solved through computation offloading. At the other end, wireless networking has initiated the use of distributed systems in sensor networks and embedded devices. Finally, online social networking is providing a novel use for distributed systems.

With this multitude of realizations, distributed systems have generated a rich set of research problems and goals. Performance was the first one. However, although the performance of distributed systems has increased, there has been a resultant increase in the programming burden. For a decade, research in distributed systems had tried to reconcile performance and programmability by making the distribution of computation transparent to the programmer through software distributed shared memory. In the end, things have not become simpler as achieving performance under distributed shared memory comes with a non-negligible semantic cost caused by the relaxed memory consistency models.

With the shift of distributed systems towards file systems and Internet-based services, the research changed focus from performance to fault tolerance and availability. More recently, the ubiquity of distributed system architecture has resulted in an increased research interest in manageability aspects. Concerns of sustainability resulted in energy-aware distributed servers, which essentially proposed dynamic reconfiguration for energy saving without performance loss. In the mobile arena, wireless networking introduced the important issues of location-awareness, ad-hoc networking, and distributed data collection and processing. Finally, as computation and storage is increasingly offloaded to the cloud, issues of security and privacy have recently gained momentum.

This book is a journey into three domains of this vast landscape of distributed systems: large-scale peer-to-peer systems, embedded and real-time systems, and security in distributed systems. The authors have recognized expertise in all three areas, and, more importantly, the experience of building real distributed systems. This book reflects the expertise of its authors by balancing algorithms and fundamental concepts with concrete examples.

Peer-to-peer systems have generated a certain fascination amongst researchers. I see at least two reasons for this. First, peer-to-peer systems come from the position of the challenger who wants to take away the crown from the long-reigning client-server model. Essentially, the challenge is whether it is possible for a democratic society of systems to function efficiently without leadership. I am not sure whether history has ever proven that this is possible, but the peer-to-peer systems researchers have shown it to be possible. They employed efficient peer-to-peer data structures called distributed hash tables (DHT) to achieve scalable data retrieval when peers come and go, fail or misbehave.

Tribal instinct might also be responsible for our interest in peer-to-peer systems: it is more likely to seek help from our peers whenever possible rather than from the outsiders. This may explain the popularity of peer-to-peer applications, such as Gnutella, BitTorrent, and the peer-to-peer games discussed in the book, some of them (Gnutella) developed even before researchers showed how to design peer-to-peer systems efficiently.

However, take heed, occasionally, peer-to-peer systems can be an illusion. Popular social networks today may look like peer-to-peer systems to the user, but, in reality, their implementation is heavily centralized. Recent concerns of data ownership and privacy have triggered an appetite for building truly peer-to-peer online social networks. It is better to understand how peer-to-peer systems work rather than be fooled again.

The distributed embedded and real-time systems, which make the middle part of the book, take distributed systems' computing labs or centers, into the real, uncontrollable world. Whether embedded in cars, buildings, or our own bodies, embedded systems must function without continuous operator assistance, adapting their functionality to the changing demands of the physical systems they assist or control. Physical systems may also incorporate highly inter-connected embedded computers in order to become cyber-physical systems. Computer scientists have always been good at designing systems for themselves: languages, operating systems, and network protocols. However, embedded systems are about others. They represent a prerequisite in implementing Mark Weiser's vision of pervasive computing, according to which computers will not just become ubiquitous, but also invisible.

Embedded computing often demands real-time guarantees, a requirement that has been shown to be challenging for any kind of computing, not just for distributed systems. This part of the book covers distributed real-time systems, how to build adaptive embedded systems from a software engineering perspective, and concludes with an interesting real-world example of software design for an aerospace system using the modeling tool they developed. After reading this book, whenever you fly, I am sure you will hope that the engineer who designed the plane's software has read it too.

Finally, the last part of the book covers security in distributed systems. Distributed systems inherently require security. Whether they are clients and servers or just peers, these parties, as in real life, rarely trust each other. The authors present key aspects of grid systems' security and dependability such as confidentiality, authentication, availability, and integrity. With the increasing popularity of cloud computing, security and privacy issues will be an even greater concern. Virtual machine environments are shown not to be sufficiently trustworthy as long as they are in the hands of the cloud providers. Users are likely to ask for stronger assurances, which may come from using the Trusted Platform Module (TPM) support, presented in this book, as well as from intelligent auditing techniques. The book's last section is about cryptography, the mystical part of computer science, which we always rely on when it comes to protecting the confidentiality of our communications.

Who should read the book? The authors recommend it for engineers and masters students. I am inclined to agree with them that this book is certainly not for the inexperienced. It requires some background knowledge, but also the maturity to read further from the recommended bibliography in order to fully internalize the material. If you finish the book and want to read more, stay tuned; this is just the first book: more is coming.

Professor Liviu Iftode
Rutgers University

Chapter 1

Introduction

Problematics

Most do not know but, in 1946, ENIAC, one of the first known computers (after Z3 in 1941 and Colossus in 1943), was already a parallel machine [WIK 11a]. The very basic programming mechanisms [1] of this computer hid this capability that was never really exploited.

Then, for years, programming remained sequential. In the 1960s, interest in parallel programming increased again. Two approaches were then explored:

– supercomputers;

– multi-processor computers.

Parallel programming on supercomputers

In the 1960s, the notion of supercomputers emerged. It was a brute-force [2] computer able to perform complex scientific calculi such as meteorologic predictions.

Technologies of this time made such computers extremely costly. As an example, the Cray-1 (1976), which constituted a major achievement in this domain, cost US$

Introduction written by Serge HADDAD, Fabrice KORDON, Laurent PAUTET and Laure PETRUCCI.

1. From documents and pictures, it is clear that ENIAC looked more like an old-fashioned telephone center than a modern computer. It was programmed by means of cables wiring switches.

2. As a comparison, a modern laptop is far more efficient than the Cray Y-MP from 1988, the most powerful computer at this time.

Distributed Systems: Design and Algorithms Edited by Serge Haddad, Fabrice Kordon, Laurent Pautet and Laure Petrucci
© 2011 ISTE Ltd. Published 2011 by ISTE Ltd.

8.8 million [CRA 11]. It weighed 5 tons and performed 166 MegaFLOPS (*FLOPS means FLoating point Operations Per Second*). It required a dedicated electrical power supply and had a very complex cooling system.

Parallelism in such computers was mainly based on the "vector calculus" principle. The idea is to apply a given operator to a vector instead of a scalar. This objective was achieved thanks to *pipe-line* in processors, which enabled "pseudo-parallelism" to process several data simultaneously. The same operation was performed on several data but at different stages in the pipe-line.

The Cray Y-MP, in 1988, was based on this principle but also comprised parallel configurations from four up to eight vector units. Then, two types of parallelism coexist: vector-based and multi-processor based.

FORTRAN was the traditional programming language for numerical applications. It was rapidly extended to enable parallel programming. Other languages were enriched with new instructions or associated libraries to also handle parallelism. However, to get all benefits from such computers, a deep understanding of their architecture was required. Then, when a computer was replaced by a new one implementing a more efficient architecture, programs had to be thoroughly rewritten to exploit the new capabilities. This additional cost could only be supported by large institutions such as the army, aircraft manufactories, state research agencies, etc.

Parallel programming on multi-processor machines

During the second half of the 1990s, when network usage was growing, some started to build *clusters* of computers. It was first a way to build "supercomputers for the poor" by assembling PC motherboards connected via a dedicated local network. The time of the first configurations involving 64 to 128 nodes is now gone: current clusters are gathering thousands of machines via high-speed networks such as glass reinforced plastic fibers. The Jaguar machine [3] is made of 18,688 processors, each one being an hex-core (that makes 112,128 cores in total) [WIK 11b]. Its peak computing capacity is 1.75 PetaFLOPS (10^{15} FLOPS).

The cost of clusters dramatically increased the affordability of supercomputers and thus almost all the fastest machines in the world are of this type. Most companies selling "traditional supercomputers" have reduced their activity.

Another aspect made this new generation of supercomputers popular; their programming is much easier and reusable compared with the old ones. This is because the programming paradigm is that of a distributed application that is not architecture

3. This was the fastest computer recorded in June 2010 [TOP 11].

dependent. Hence, “classical” distributed programming techniques can be used and preserved when transferring the program from one machine to another.

In fact, the Internet itself can be seen as a gigantic supercomputer, as applications like `SETI@home` [UCB 11] do. Thus, some experiments involve dozens of thousands of machines over the Internet. The main difference with clusters is that the nodes are not connected via a high-speed network. There is also a need to check for trust between nodes.

Thus, the problem of distributed computing is now mainly a software problem. However, distributed applications belong to a difficult class of problems.

Objectives of this book

This book is aimed at engineers or masters students or anyone familiar with algorithmic and programming who wants to know more about distributed systems.

We deliberately chose, in this first book, to group the presentation of distributed systems in relation to their design and their main principles. To do so, we present both the main algorithms and replace them in their application context (i.e. consistency management and the way they are used in distributed file systems).

Description of chapters

The first part is dedicated to large-scale peer-to-peer distributed systems. This is currently a very active area with new improvements (especially those induced by mobility and the numerous small devices we use daily):

– Chapter 3 presents the main principles of large-scale distributed peer-to-peer systems. It details the main algorithms used to communicate and ensure trust in such systems.

– Chapter 4 deals with peer-to-peer storage, an application domain which has already accumulated several years of experience. Some well-known protocols and tools, such as BitTorrent and Gnutella, are detailed.

– Chapter 5 presents another hot application domain for such systems: gaming. Once again, the principles adapted to this class of applications are put into a practical perspective.

The second part is dedicated to distributed real-time embedded systems: a domain that has always been very active. The topic of distributed systems is now gaining importance in the design of the next real-time systems generation.

– Chapter 7 presents the holistic analysis, a well-known method used to compute the schedulability of distributed real-time systems. This chapter provides some background knowledge of scheduling analysis in the distributed real-time systems area.

– Chapter 8 deals with the design of adaptative real-time embedded systems. This second contribution provides the results for some schedulability theories, it also details some of the fundamental insights of the design process of adaptative real-time embedded systems.

– Chapter 9 presents an innovative approach to designing the new generation space systems. This approach is supported by a toolset which is required to automate the process where possible.

The third part is devoted to security issues in distributed systems. This is a critical area that is of the utmost importance in such systems where trust among communicating entities and confidentiality of data are key issues:

– Chapter 11 presents the main characteristics of grid computing, with a focus on security. The security properties that have to be guaranteed are detailed, and how they are achieved in practice is presented through several case studies.

– Chapter 12 tackles the issue of data confidentiality using cryptography. It describes the core techniques which use symmetric key and public key protocols, and details their main characteristics.

The *MeFoSyLoMa* community

MeFoSyLoMa (Méthodes Formelles pour les Systèmes Logiciels et Matériels[4]) is an association gathering several world-renowned research teams from various laboratories in the Paris area [MEF 11]. It is composed of people from LIP6[5] (P. & M. Curie University), LIPN[6] (University of Paris 13), LSV[7] (École Normale Supérieure de Cachan), LTCI[8] (Telecom ParisTech), CÉDRIC[9], (CNAM), IBISC[10] (University of Évry-Val-d'Esssone), and LACL[11] (University of Paris 12). Its members, approximately 80 researchers and PhD students, all have common interest in the construction of distributed systems and promote a software development cycle based on modeling, analysis (formal), and model-based implementation. This community was founded in 2005 and is federated by regular seminars from well-known researchers (inside and

4. This acronym stands for Formal Methods for Software and Hardware Systems (in French).
5. Laboratoire d'Informatique de Paris 6.
6. Laboratoire d'Informatique de Paris Nord.
7. Laboratoire de Spécification et de Vérification.
8. Laboratoire Traitement et Communication de l'Information.
9. Centre d'Études et de Recherche en Informatique du CNAM.
10. Informatique, Biologie Intégrative et Systèmes Complexes.
11. Laboratoire d'Algorithmique, Complexité et Logique.

outside the community) as well as by common research activities and the organization of events in their domains such as conferences, workshops, or book writing.

The editors of this book, as well as most authors, are from this community.

Bibliography

[CRA 11] CRAY-RESEARCH, "Cray history", http://www.cray.com/about_cray/history.html 2011.

[MEF 11] MEFOSYLOMA, "MeFoSyLoMa, home-page", www.mefosyloma.fr 2011.

[TOP 11] TOP500.ORG, "BlueGene/L", http://www.top500.org/lists/2010/06 2011.

[UCB 11] UCB, "SETI@home", http://setiathome.berkeley.edu/ 2011.

[WIK 11a] WIKIPEDIA, "ENIAC", http://en.wikipedia.org/wiki/ENIAC 2011.

[WIK 11b] WIKIPEDIA, "Jaguar (computer)", http://en.wikipedia.org/wiki/Jaguar_(computer) 2011.

FIRST PART

Large Scale Peer-to-Peer Distributed Systems

Chapter 2

Introduction to Large-Scale Peer-to-Peer Distributed Systems

2.1. "Large-Scale" distributed systems?

For several years now, the term "Large-Scale" has applied to distributed systems. It indicates that they involve a *very large number of computers* that are usually spread worldwide.

As a typical example, we are all regular users of a large-scale distributed system: the Internet. So, the "Internet universe" provides a rough idea of the way such systems work. How programs contact and cope with other programs (such as the relationship between a web browser and a web server) can be easily imagined. Moreover, it is possible to feel how the Internet is a worldwide parallel machine of which each connected computer is a part. The structure of such distributed systems can thus be comprehended by anyone regularly using the Internet.

However, only a few really understand the implication of "large scale" regarding such distributed systems. Creating a program to be executed on a few (or several dozens) nodes is completely different from the same exercise for a few thousand nodes or more. This is a problem when ubiquitous systems [1] become more and more present in our day-to-day life.

Chapter written by Fabrice KORDON.

1. This denotes the set of devices (computer, PDA, cellular and smart phones, specialized devices, etc.) connected into a network in a transparent way. Thus, users deal with several *media* in their day-to-day life to accomplish personal or professional matters.

Distributed Systems: Design and Algorithms Edited by Serge Haddad, Fabrice Kordon, Laurent Pautet and Laure Petrucci
© 2011 ISTE Ltd. Published 2011 by ISTE Ltd.

2.2. Consequences of "large-scale"

What are the consequences of having "large-scale" distributed systems? Simply the number of components in the involved systems (to illustrate the concepts, we consider here that all components are programs, even if the human factor is important too). The following are some of the problems encountered:

– communication;

– fault tolerance;

– decision making.

2.2.1. *Communication in large-scale distributed systems*

For distributed systems, every component can be connected to any other. However, such a connectivity cannot be maintained when the system grows. It is obvious that the management of dozens of thousands of connections cannot be handled by a single program.

Point-to-point communication can no longer be adapted; new mechanisms are required, such as *broadcast* or *multicast*. It should be noted that such mechanisms already exist (for example, they are used in the Internet, when a new router is connected to the global network).

Thus, one of the main characteristics of communications in large-scale distributed systems is their anonymous aspect. Components of such systems communicate without knowledge of their location or identity (e.g. IP[2] address). This is due to the fact that during the system execution, due to the network load, or faults mentioned in the next section, the location of the components may change. Thus detecting a component via its IP address is absurd.

2.2.2. *Fault tolerance in large-scale distributed systems*

No problems are usually encountered during one hour of computing on a single host. This is not the case when this computation time employs 10,000 machines. The probability of the occurrence of failure tends to 100%. In fact, the more CPUs[3] that are involved in computing, the higher the probability that a failure will occur in the involved machines (host crash, network breakdown, etc.).

2. IP stands for "Internet protocol".

3. CPU stands for "central processing units".

Thus, it is necessary to create dedicated mechanisms to handle such failures and ensure that the program is able to overcome these. Such mechanisms have a dramatic impact on the software architecture of large-scale distributed systems.

2.2.3. *Decision making in large scale distributed systems*

Decision making in large-scale distributed systems is a difficult task. If, when parallelism is limited, each component (piece of program) has a full copy of the involved data, such hypothesis cannot be considered when the number of hosts increases: managing and updating data would require too many resources.

Thus, each component in the distributed system is expected to handle decisions based on a (very) partial view of the global system state. Collection of the involved data remains a difficult task that also has far-reaching impacts on the software architecture.

2.3. Some large-scale distributed systems

Earlier, we mentioned the Internet as an example of a large-scale distributed system. In fact, this example is not typical because the Internet itself consists of a worldwide network of machines (evaluation in June 2010 was 1.97 billion users according to [IWS 10]); most users only operate a very limited number of machines simultaneously (typically, a web server, composed of a cluster of machines to handle services – most online vendors have such an architecture).

Thus, to provide a more accurate view of large-scale distributed systems, we briefly present two examples that emphasize their main characteristics.

2.3.1. *SETI@home*

Our first example is an application launched in 1999. An improved version is still in use today (the first version is now called `SETI@home` "classic").

When `SETI@home` [UCB 11] was released, it raised a huge interest from both Internet users and computer science professionals. The main objective of this application is to exploit the useless CPU time of machines connected to the Internet to compute and analyze radio signals coming from outer space. The objective is to look for traces of non-human intelligence[4].

4. SETI is the acronym for *Search for Extra-Terrestrial Intelligence*.

The way SETI@home operates is quite simple. Users willing to offer unused CPU capacity download an application to create an account on a server and set-up its configuration (enabling the conditions for SETI to activate, etc.).

Once launched, the application remains paused until the host computer becomes inactive. Then, it connects to the server, downloads data to be analyzed and, once computation is over, uploads the results to the server. If the host machine resumes its activity, the application stops until the next inactive period.

SETI@home met great success in the Internet community and more than 5 million people have used it since it was launched. From a performance point of view, it is also a success as the computation strength has increased up to 364.8 TeraFLOPS [5] in January 2008 with about 1.7 million simultaneous users. As a comparison, let us note that the most powerful computer is about 1,759 TeraFLOPS [6] for an incredible cost. However, no non-human activity has been detected so far.

Let us consider the three main problems noted in section 2.2 and the way they are addressed by SETI@home:

– communications: SETI@home mainly relies on the classic Internet architecture, the so-called *client-server* approach. The downloaded application connects to a server to obtain data and sends back the results to this same server;

– fault tolerance: its management is simple. If a participating machine (a client) does not send back results for a while, the server may resend the set of data to another client. Let us note that users trust the system as they accept the program to run on their computer, which communicates with another machine. They expect SETI@home not the behave in a malicious way;

– decision making: all decisions are taken by servers, clients only analyze local data they retrieve from the server.

2.3.2. *The automated motorway*

Our second example concerns *intelligent transport systems* (ITS). These remain a challenge as there is no implementation yet. However, this is an important and active research domain in the USA, Japan, and in Europe, with investments involving billions of Euros and concerning numerous applications. Here, we will focus on one of them: the automatic motorway (seen from the perspective of distributed systems).

The objective of the automated motorway is to let especially equipped vehicles drive without human intervention. Such a system is of interest for main roads; it is

5. 1 TeraFLOP corresponds to 10^{12} FLOPS.

6. This is, in June 2010, the Jaguar system at NCSS, a 18,688 node computer (each node runs a dual hexcore AMD Opteron 2435 (Istanbul) processors running at 2.6 GHz, 16 GB of DDR2-800 memory [WIK 11].

also intended for trucks. Let us note that some experimentation started around the mid-1990s in the context of the PATH project [UCC 04], whereby a platoon of automated vehicles successfully drove along a dedicated freeway. However, due to the cost of road adaptation, this solution got no further and was abandoned.

Based on these first successes and due to technological advances, the principle has changed. Current solutions tends to minimize the adaptation of infrastructures and put more "intelligence" in the vehicles.

Let us now draw a more up-to-date vision of the automated motorway. The components if the system are:

– *the motorway infrastructure*, which offers lanes, communication mechanisms, and global information about the road itself (such as speed limit, information about traffic or accidents, etc.). The network may only exist from time to time (e.g. WIFI communication spots may be located alongside emergency phones);

– *vehicles*, which are equipped with sensors and a network interface enabling local communication with both other vehicles and the road infrastructure. Only local communication are needed (e.g. with close vehicles only).

A priori, each vehicle communicates with its close neighbors. Information (e.g. acceleration and speed data, emergency braking, etc.) is propagated to the surrounding vehicles. Information then travels faster than it does from driver to driver (when they notice other drivers' behavior).

Vehicles can also receive messages from the road infrastructure.

Such a system can be structured as a hierarchy of subsystems (see Figure 2.1). Here, there are two levels of hierarchy, each one dedicated to a given set of services.

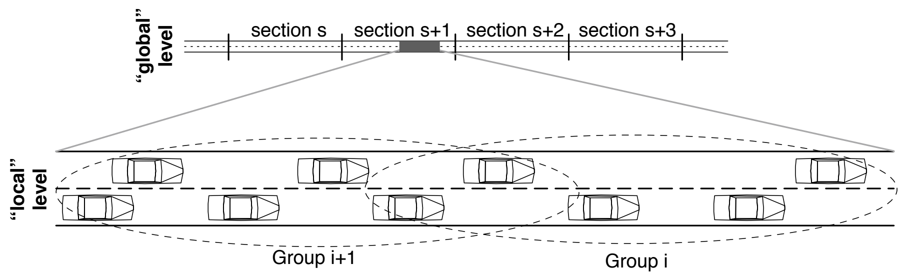


Figure 2.1. *Possible hierarchical architecture for the automated motorway*

The first level (called "local" in Figure 2.1) deals with a group of circulating vehicles. Obviously, this group is composed dynamically and evolves when cars enter or leave the motorway. It is also possible that some vehicles go from one group to another when their speed becomes slightly higher than another group (overlapping between groups can also be considered, in which case, one vehicle may belong to two groups for a while).

Inside the group, the security of each vehicle must be handled. To do so, information is shared between vehicles and possibly the road infrastructure. In which case, decisions are made by vehicle controllers to avoid collisions and ensure safety.

The second level (called "global" in Figure 2.1) focuses on higher-level services and does not handle collision avoidance at all. Its objective is to propagate information regarding unexpected events, such as an accident or bad weather conditions (e.g. localized fog). Any change in the dynamic of the system (e.g. a sudden braking of a group) must be analyzed and backward propagated to let later vehicles anticipate the problem if necessary.

Traffic management (to prevent traffic-jams) can also be handled at that level. The system may decide to reduce the speed limit of some sections to limit the number of vehicles coming to a section where an unexpected event is occurring.

Let us consider the three main problems noted in the section 2.2 and the way they are addressed here:

– communications: each group of vehicles must have its own broadcast mechanism. If a vehicle belongs to two groups, it receives information from the two groups and may propagate data backward (e.g. from the front group to the back group). Thus, information goes from hop to hop. When a vehicle sends information to the group channel, it does not have an idea of which participant will get this information (anonymous communication). The principle is that "involved vehicles" will get the appropriate data to operate;

– fault-tolerance: such systems require fault-tolerance mechanisms. Typically, a vehicle losing connection with the group must continue to behave safely within the group (until communication is back).

Vehicles must trust each other: a situation where a component sends information to go faster than other vehicles or cause a crash must be avoided.

– decision making: at the local level, decisions are probably taken by vehicles. At the global level, decisions could be taken by server. There may only be one server for the motorway or a set of servers, each one dealing with a set of sections (and then communicating together or to a higher-level server that handles the motorway).

2.4. Architectures of large scale distributed systems

During the early stages of parallel programming, computer scientists tried to extend "classical" mechanisms to produce a distributed execution. Thus, the notion of procedure call was extended to *remote procedure call* (RPC) [SUN 88].

2.4.1. *Remote procedure call*

Figure 2.2 shows the behavior of a RPC. The two components involved have asymmetric behaviors. When the *invoker* is ready (1 in Figure 2.2), it generates a request to the *invoked* that holds the code to be executed and waits for an answer. Once the *invoked* receives the request (there may be a list of pending requests if the *invoked* receives many of them), it is processed (2 in Figure 2.2) and then an answer is sent back to the *invoker*. When the answer is received, the *invoker* conintues its execution (3 in Figure 2.2).

The first message contains the parameter required by the code to be executed. The second message contains return values (or modified parameters). When there is no return value, an empty message is sent as the RCP is a synchronous mechanism (the *invoker* must not continue its execution before the end of the remote procedure execution).

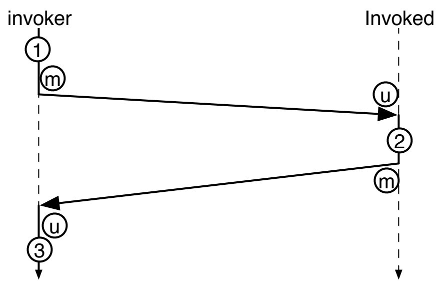


Figure 2.2. *The Remote Procedure Call mechanism*

Some typical characteristics of distributed systems must be outlined there:

– the two hosts executing the *invoker* and the *invoked* may have different architectures. the data format may thus differ (e.g. an integer is stored on 32 bits for the first machine, on 64 bits for the second one). It is then necessary to encode data instead of sending a memory segment. The marshaling operation (*m* in Figure 2.2) encodes data into a message and the unmarshaling operation (*u* in Figure 2.2) decodes this message to reconstitute the original data. Both marshaling and unmarshaling must respect the same conventions of course;

– identification of the *invoked* is explicit. Thus, each component in the system must "know" each actor to interact with;

– finally, a crash of the machine hosting the *invoked* makes the RPC unavailable.

RPC rely on two principles: message passing and point-to-point communication. This is the case for any high-level communication mechanism that requires the exchange of messages over the network. A mechanism like RPC aims to structure interactions between components into a protocol.

Parallel to the creation of RPC, scientists were formulating the main Internet mechanisms based on simple message passing such as broadcast and multicast. So, if the Internet follows the client/server protocol (an extension of RPC presented in the next section), it is based on message passing (this is a peer-to-peer approach that is also presented later in this chapter).

2.4.2. *The client/server model*

The client/server (see Figure 2.3) is a natural extension of the RPC. In the early 1990s, middleware such as CORBA [OMG 06] extended this initial notion to the object model. Registration mechanisms enable the dynamic identification of the server address when necessary and it is not necessary to encode its IP address in the program. This is an important evolution that increases the portability and enables evolution of distributed applications.

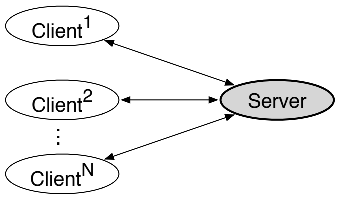


Figure 2.3. *Architecture of the client/server model*

However, the client/server model remains a point-to-point protocol in which clients initiate contact with a server. The server is reactive and only answers requests.

The context of the system (that enables decision making) is usually centralized in the server. Thus, to tolerate faults, the server must be replicated on another machine. This approach is often used for large servers (e-commerce, Google, etc.).

2.4.3. *The master/slaves model*

The master/slaves model (see Figure 2.4) is a variant of the client/server model. In that model, all initiative is taken by the “master” that provides jobs to the “slaves”.

In that model, the “slaves” are the reactive ones and the communication employs the point-to-point model. The “master” handles the context of the application and

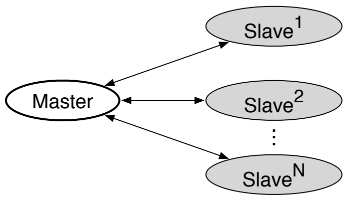


Figure 2.4. *Architecture of the master/slaves model*

takes decisions. Usually, to avoid problems in case of a crash, the context and related data are regularly backed up. If a problems occurs, the context can be reloaded and the execution continues from the last *checkpoint* (backup time).

2.4.4. *The peer-to-peer model*

Contrary to the "urban legend", the peer-to-peer model is not a way to upload files but is a way to structure distributed applications. Its architecture is presented in Figure 2.5. It shows that no component is handling others. Each component collaborates to achieve a common goal. For that reason, each participant might be able to communicate with each other. Of course, there will not be a physical dedicated communication link between each executing process but, logically, we assume this is the case.

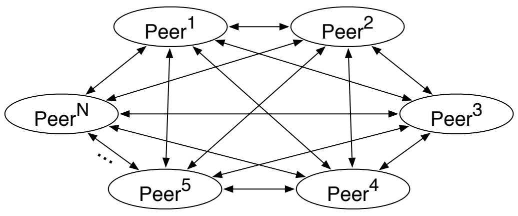


Figure 2.5. *Architecture of the peer-to-peer model*

Communication mechanisms are usually based on broadcast or multicast to enable data exchange between components. Point-to-point liaison may be used when a large amount of data has to be passed from one component to another for performance reasons (it is more efficient).

"Peers" may structure themselves for performance reasons. Ring-based, tree-based, or star-based configurations are common. These structures aim to facilitate:

– actor localization in the system;

– message routing between actors.

2.4.5. *Use of these models*

In some cases, distributed systems follow one architecture scheme only. This is typically the case for Internet client/server applications where the "intelligence" usually resides in the server. However, for more complex applications, several architecture models may be used at different stages of the execution, or even simultaneously when the selected model is more efficient for the actions to be performed at this stage.

This is the case for the examples we mention in this chapter:

– the Internet, as previously stated, is built on a peer-to-peer architecture for message routing but on a client/server one for high-level services (http, mail, etc.);

– `SETI@home` relies on a client/server for the connexion phase but data analysis is provided using a master/slaves method;

– the automated motorway mixes peer-to-peer (locally between vehicles) and client/server (to handle informations provided by the road infrastructure).

The role of an component may also change over its execution. As an example, in some cases a component is sometimes a server and also a client (e.g. it requires another sub-service to operate the one it offers). Such changes of roles and mix of architectures is typical of large-scale distributed systems, as the following chapters will show.

2.5. Objective of Part 1

The goal of this part is to show, using several existing examples, the way large-scale distributed systems work. We are more precisely focused on peer-to-peer systems and we aim to describe their characteristics and the way they behave.

This part contains three chapters.

First, Chapter 3 defines the main basic concepts of such systems. It details their structure and presents how messages are routed among numerous hosts. Finally, it deals with the problem of building trust in such systems since, as for human societies, malicious components may ruin the goal of a distributed application.

Then, Chapter 4 details two distributed storage systems: BitTorrent and Gnutella. The analysis os these systems shows the high dynamicity of such applications and illustrates the use of several structuring models regarding the execution of an application. The idea is to illustrate a typical application of large-scale peer-to-peer distributed systems: file sharing.

Finally, Chapter 5 details another typical and recent example of large-scale distributed systems: massively parallel games. The authors present the limitations of current client/server-based approaches, such as those of *Second Life* or *World of Warcraft*, and present new approaches to be implemented in new massively multiplayer on-line games. This chapter shows how basic mechanisms (such as those presented in Chapter 3) must be adapted to fit new needs for this class of applications.

2.6. Bibliography

[IWS 10] IWS, "World Internet users, June 2010", http://www.internetworldstats.com/stats.htm 2010.

[OMG 06] OMG, "CORBA component model 4.0 specification", 2006.

[SUN 88] SUN, "RPC: remote procedure call protocol specification Version 2; RFC1058", 1988.

[UCB 11] UCB, "SETI@home", http://setiathome.berkeley.edu/ 2011.

[UCC 04] UCC, "California PATH annual report 2004 (University of California, Caltrans)", 2004.

[WIK 11] WIKIPEDIA, "Jaguar (computer)", http://en.wikipedia.org/wiki/Jaguar_(computer) 2011.

Chapter 3

Design Principles of Large-Scale Distributed System

3.1. Introduction to peer-to-peer systems

This chapter reviews the general concepts of peer-to-peer (P2P) systems. In the last few years P2P systems have become incredibly popular among Internet users. The key factors behind their success are scalability (in terms of number of hosts), highly distributable without any central coordination, the resilience (to faults and churn), and low cost of ownership. This is achieved by aggregating the resources of a large number of computers, and using replication to mask individual faults.

P2P architectures are a alternative approach to traditional distributed schemes based on client/server. They were particularly effective to locate resources in large configurations consisting of millions of nodes.

The phrase "P2P" is used to describe a wide variety of applications and architectures [LUA 05]. Several P2P applications have already been successfully deployed over P2P networks, including distributed computing, such as *SETI@home* [SUL 97] and *XtremWeb* [CAP 05], data sharing, such as *Publius* [WAL 00], *Kazaa*[1], and *Gnutella* [JOV 01], and instant messaging, such as *ICQ*[2] and *Jabber*[3]. P2P file sharing, in particular, has become the high bandwidth-consuming application, representing 20% of

Chapter written by Xavier BONNAIRE and Pierre SENS.

1. See http://www.kazaa.com
2. See http://www.icq.com
3. See http://www.jabber.org

Distributed Systems: Design and Algorithms Edited by Serge Haddad, Fabrice Kordon, Laurent Pautet and Laure Petrucci
© 2011 ISTE Ltd. Published 2011 by ISTE Ltd.

Internet traffic in 2009. In all these applications, network participants (peers) share a portion of their resources, such as processing power, disk storage, or network bandwidth, directly available to other peers spread on the networks.

In this chapter, we introduce first concepts that are common to P2P systems. Then, we detail the distributed hash tables (DHT). DHT is a distributed storage service that provides persistence and fault tolerance, and can scale to a large number of nodes. By hiding the complexity of scalable routing, fault tolerance, and self-organizing overlays, DHTs provide a powerful abstraction that greatly simplifies the task of building large-scale distributed applications. We also focus on how to build trust services on top of P2P systems.

3.2. The peer-to-peer paradigms

In P2P distributed systems, each participant (peer) knows a set of logical neighbors to which it can directly send messages via IP[4] protocol. These sets of neighbors form a logical graph connecting all the nodes of the network. We call this graph an *overlay* above the IP network.

Roussopoulos *et al.* [ROU 04] identified as P2P systems those that meet the following criteria:

Self-organization: the network automatically adapts itself to the peers' arrival or departure without compromising the integrity of the system. Peers exploit local information provided by its neighbors to organize the network.

Distribution: there is no centralized control to manage peer behavior.

Scalability: issues such as bottleneck, overloaded nodes, and single-point-of-failures are avoided allowing to the system to scale up to millions of peers.

3.2.1. *Classification of peer-to-peer systems*

P2P systems can be classified according to two general criteria: degree of centralization and structure.

Degree of centralization

Decentralization is a key characteristic of P2P systems. In practice, most use a centralized or partially centralized architecture in which some important functions, such as handling metadata, are performed by only a few nodes in the network. In

4. IP stands for "Internet protocol".

most cases a decentralized solution also exists, but is usually less efficient. In fact, the choice between a centralized and a decentralized approach is usually a trade-off between performance, fault tolerance, and cost.

Centralized systems

In these architectures, a set of dedicated nodes (servers) carries out critical operations, typically peer authentification or the location of resources owned by peers. This small set of nodes is usually located in the same local-area network (LAN) or spread geographically but connected through high bandwidth links. For instance, for file sharing the critical operation is the identification of peers having a copy of a specific file. This group of servers is only used to locate peers in order to link them. When peers are connected, they directly exchange their data. The *Napster* file sharing system and more recently *OceanStore* [KUB 00] a general purpose data storage system from Berkeley are examples of such architectures. Figure 3.1 illustrates location of resources in a centralized P2P system. Peer S sends a lookup request to the core to locate some data. Then, based on its global view of data indexes, the core forwards the request to nodes A and B. Finally A and B reply directly to S. S usually arms a timer at the beginning of its request, when the timer expires it then chooses its responses among the set of replies it receives.

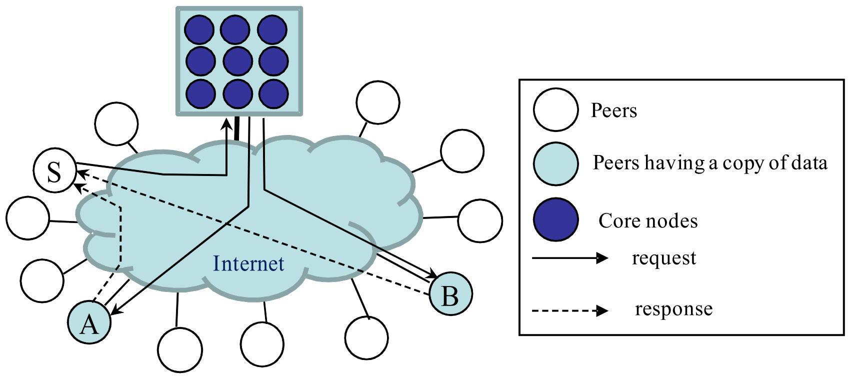


Figure 3.1. *Location in a centralized P2P system*

The set of servers maintains a global view of all resources (meta-data). This centralization allows servers to perform complex operations such as data indexing, searches, ordering client requests, and trust management requiring less network communications than in a decentralized design. Efficient load balancing can also be achieved due to the low cost of reconfigurations and state migrations between nodes. Finally, a small number of core machines are easier to manage and administrate compared to thousands of machines spread all other the Internet.

However, centralization has two important disadvantages. First, the system core is a single point of failure. It then requires redundant hardware and network links in order to prevent the isolation of servers due to node crashes or network partitioning. If the core is inaccessible no client request can be satisfied. Second, this solution is not scalable as all the localization traffic goes through the same core. Then, servers need powerful hardware and high bandwidth links, resulting in substantial operational costs.

To overcome these two disadvantages, most P2P systems rely on a partially centralized architecture.

Partially centralized architecture

These systems lack a single central point in the network but employ a small group of nodes to carry out critical tasks, they are sometimes referred to as partially centralized, or hybrid architectures. In this approach, systems distinguish several kinds of nodes, assigning them different roles. A two-level hierarchy is a widely used architecture, in which powerful, well-connected nodes, commonly referred as supernodes, form a higher level overlay used to store metadata and broadcast search queries. Other nodes connect to one or more supernodes, and in some networks may also form a lower-level overlay through which messages can be routed.

Since supernodes are spread on the network, they are less sensitive to partioning compared to centralized architectures. However, complex operations on metadata or data searching may be more expensive since they require numerous exchanges of message between supernodes through Internet. Nevertheless, this distribution model meets a certain success and is the basic scheme of the most popular file-sharing systems such as *Kazaa* or *eDonkey*.

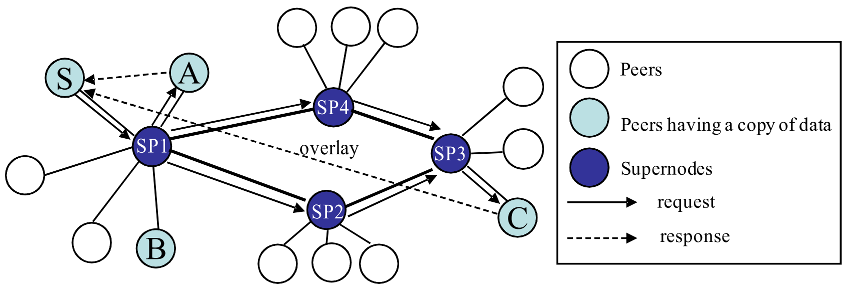


Figure 3.2. *Example of search request in partially centralized architecture*

Figure 3.2 shows the principle of a search query in a partial centralized system. Peer S sends a request to its supernode. This request is then forwarded to the connected peer, A, responding to the search criteria. The request is also forwarded to other supernodes, which send the request to their peers having a copy of the data, C. Finally, A and C directly send a response to S.

Distributed architectures

Partially centralized systems are not suitable and reach their limits when the number of resources is high or in presence of churn (i.e. the continuous process of node arrival and departure). In these cases, a supernode cannot maintain a correct view of its peers. When the number of peers associated with a supernode becomes too high, the supernode becomes overloaded and views its peers incorrectly.

Distributed approaches do not consider any dedicated nodes and aim to balance load and partition the global view among all peers. In such architectures, each peer belongs to the global overlay network and only maintains the view of resources of its logical neighbors. To tolerate node crashes, information from each peer is replicated on its neighborhood in the overlay.

As decentralized systems have no single points of failure, a robust system can withstand and recover from a variety and number of failures. However, a decentralized approach can be much more inefficient than the corresponding centralized one. Typical example are data location and search queries. A central directory can respond to a search query providing the network address of a matching resource almost immediately. Conversely, a decentralized search algorithm may require contacting tens or hundreds of nodes, generating a considerable amount of traffic and long delays before the result is sent to the client. Even worse, simple search algorithms (e.g. based on flooding) may produce partial results if the resource is located too far away from the source.

3.2.2. *Structure of overlay networks*

Overlay networks were designed to search information following logical links between peers. Besides the degree of centralization, overlay networks can also be classified into structured and unstructured.

Unstructured overlays are constructed in a non-deterministic manner. Therefore, they do not have a defined topology. If a peer wants to search for a desired piece of data in the network, the query has to be flooded through the overlay in order to find as many peers as possible that share the data.

Structured overlays are constructed so that the node graph follows a particular topology. Contrary to unstructured overlays, a node cannot freely choose its neighbors. Instead, it selects them from a set of possible neighbors, which depends on the node identifier.

Let us now detail these two approaches.

Unstructured overlays

Unstructured overlays are easily built: to join the network a peer has to contact a participant node and use its neighborhood information to built its connections. The resulting topology is then *unstructured*. Therefore, these networks are not sensitive to *churn* (i.e. the continuous process of peer arrival and departure). No state is transferred to or from the joining node, and nodes do not maintain any routing information besides that from the unique identifiers of their direct neighbors and of some remote peers to increase the lookup process.

However, the location of resources is costly. If a peer wants to search for a piece of data in the network, the query has to be flooded through the network in order to find as many peers as possible that share the data. Typical search techniques are breath first search (BFS), random walk [ZEI 04], and iterative deeping [LI 06] among others. Basically most of these techniques forward queries to all neighbors. Neighbors originating the queries return the results, and consecutively, they will forward the query to their respective neighborhood. The process will continue until an end condition is reached (time-to-live: TTL). For instance, this technique is used by Gnutella unstructured networks [JOV 01].

Figure 3.3 illustrates the flooding principle. Peer S generates a search request with a TTL of three. All peers (A, B and C) having a copy of the corresponding data which is reachable at three hops from the source directly sends a response to S.

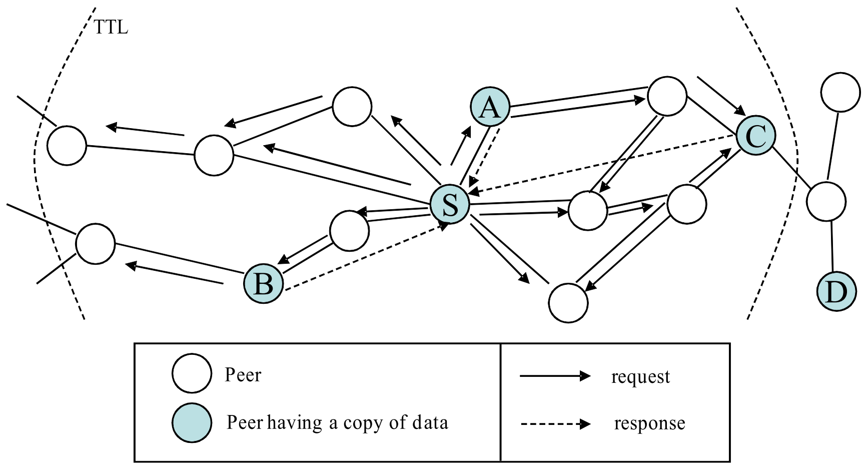


Figure 3.3. *Flooding of a search request in an unstructured overlay*

To reduce the flooding cost, some systems use random walks [GKA 06]. In random walks, peers send their request to one of their neighbors chosen uniformly at random. When a peer receives a request, it locally evaluates the request and possibly responds directly to source. It also forwards the request to one of its neighbors until the TTL is reached. At the end of a random walk, the source evaluates whether it receives the response; if the full response was received, it generates a new random walk.

Figure 3.4 illustrates the principle of random walks. Peer S looks for three copies of data. In the first random walk, only one peer B responds to S. Then S generates a new walk, which discovers peers A and C.

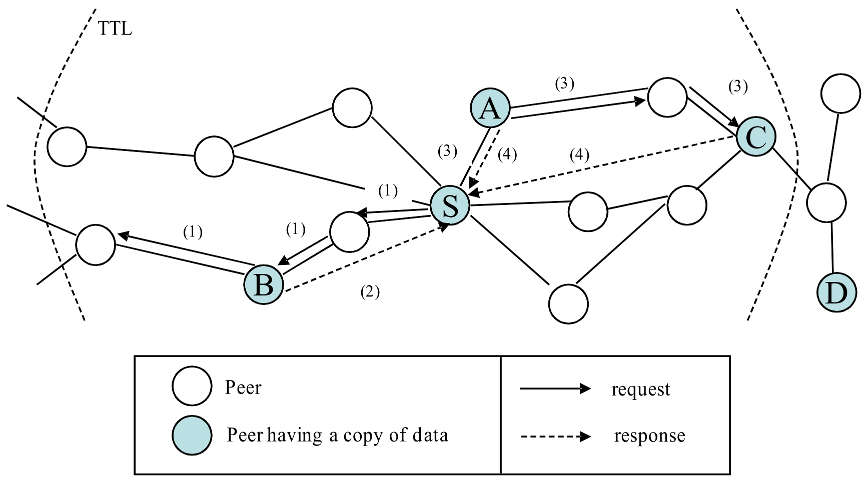


Figure 3.4. *Random walk search in an unstructured network*

Even if random walks significantly reduce the search cost compared to flooding, unstructured networks cannot undertake exhaustive requests as the probability to find a data is directly proportional to the depth of the search (TTL). For instance, in Figures 3.3 and 3.4, peer D located at five hops is never found.

Structured overlays

Structured overlays have emerged to improve search requests and to find data in an exhaustive way. These networks are totally decentralized and organized based on a specific topology [GUM 03] (ring, butterfly, hypercube, Bruijn graphs, plaxon-trees, and others). This topology defines a name space where each peer is associated with a unique identifier (a node ID).

Many structured overlays have been developed, including *Pastry* [ROW 01a], *Chord* [STO 01b], *CAN* [RAT 01], or *D2B* [FRA]. These systems are based on the same basic principles and mainly differ in their topology.

Systems	Topology	Average number of hops	Size of the local routing table
Pastry	ring	$O(logN)$	$O(logN)$
CAN	d-tore	$O(N^{1/d})$	$O(1)$
Tapestry	tree	$O(logN)$	$O(logN)$
Chord	ring	$O(logN)$	$O(logN)$
Kademlia	XOR	$O(logN)$	$O(logN)$
Koorde	ring/de Bruijn	$O(logN)$	$O(1)$
D2B	de Bruijn	$O(logN)$	$O(1)$
Viceroy	Butterfly	$O(logN)$	$O(1)$

Table 3.1. *Characteristics of structured protocols*

Data are identified by *storage keys* randomly chosen in the space of node IDs. Each key is then dynamically associated with a unique node of the network called the *root of the key* whose ID is the closest to the *storage key*.

Each node maintains a routing table to locate data from its key. This table includes the identifier and the IP address of other nodes according to the topology. Each node also maintains a list of its near neighbors in the space of identifiers. Knowledge of this neighborhood allows the local node to determine the list of keys for which he is responsible.

Application programming interface (API) of a structured overlays has three basic primitives:

– `join()` function inserts the calling node in the overlay according to its ID,

– `leave()` function removes the calling node,

– `route(k,m)` routes message m to the root of the key k.

There are several protocols based on a *key-based-routing* (KBR). We provide here a non-exhaustive list of other KBR designs (characteristics of these protocols are summarized in Table 3.1). Tapestry [ZHA 02] uses a routing algorithm based on the work by Plaxton *et al.* [PLA 99], and is therefore very similar to Pastry. CAN [RAT 01] uses a d-dimensional Cartesian coordinate space that wraps around its edges, forming a d-torus. Nodes maintain a list of 2d neighbors, and the average routing path length is $O(N^{1/d})$ where N is the total number of nodes. Chord [STO 01b], like Pastry, uses a ring topology in which node IDs belong to a circular address space. Instead of leaf sets, nodes keep a successor list of the s closest nodes clockwise on the ring. Routing tables are also slightly different, each node maintaining a list of other nodes located at exponential distances on the ring (known as the finger table). Kademlia [MAY 02] uses a XOR metric to compute the distance between two node IDs, treating nodes as leaves in a binary tree. The resulting number of hops for lookup operations is $O(logN)$. Viceroy [MAL 02] uses a butterfly network to achieve a logarithmic number of hops with O(1) state per node, while Koorde [KAA 03] and D2B [FRA] achieve the same result using de Bruijn graphs.

3.3. Services on structured overlays

This section focusses on services built on top of structured overlays.

3.3.1. *Services*

Similarly to the OSI layer model for network protocols, the protocol stack of a structured P2P network can be divided into three layers, or tiers [DAB 03] as shown in Figure 3.5. The bottom layer, known as tier 0, represents the routing layer. Routing is accomplished using routing keys, so this layer is also referred to as KBR. The second layer, or tier 1, corresponds to the P2P protocols that use the KBR layer to provide a specific service, such as a DHT or a multicast service. Finally, tier 2 designates any software that uses the services provided by tier 1.

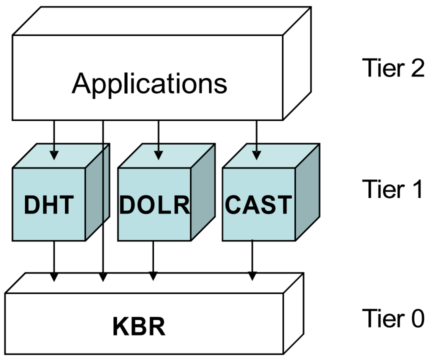


Figure 3.5. *Services on P2P systems*

Several popular services have been developed at the tier 1 layer:

– DHTs are a distributed storage services. DHTs distribute <key, data> pairs on nodes of the network. Placement of copies of data is usually constrained in the neighborhood of the root of the key. This service is detailed is the following section;

– DOLRs (*decentralized object location and routing*) provide a distributed directory service. As DHTs, DOLRs associate a key with each data set. However, in DOLRs clients can store objects anywhere in the system, and publish the location of the replicas under the object key. Tapestry [ZHA 02] is a example of DOLR;

– CASTs provide a group communication service. Nodes can join or leave a group or broadcast messages to group members. Scribe [CAS 02] and Pagoda [BHA 04] are example of CAST service.

Examples of tier 2 layers are P2P file systems, such as Ivy [MUT 02], OceanStore [KUB 00], or Pastis [BUS 05]; resource allocation services such as Vigne [JEA 07], or services to index databases, such as PinS [VIL 04].

Finally, in some cases, the user application may be considered as a fourth layer on top of tier 2. This is the case, for instance, of a legacy application using a P2P file system.

3.3.2. *DHTs*

DHTs provide the substrate to build scalable, fault-tolerant, and load-balanced distributed applications on top of large-scale systems. DHT is a distributed data structure that holds pairs <key, $data$> in a completely distributed manner by mapping each key-data to a P2P node or, in case of replication, a set of them. To this end, a DHT exploits hashing techniques that provide uniform distribution. DHT-based overlays such as Past [ROW 01a], DHash [DAB 01], or OpenDHT [RHE 05] are widely used to build P2P applications and they can locate objects (resources) in a small number of hops (i.e. $logN$ among N peers). DHT provides a scalable and efficient search method. On the contrary to unstructured networks, structured networks enable complete search.

A DHT stores a data d uniquely identified by a storage key k. To be located quickly, the d is stored on the root node of k. The block is then replicated in the logical neighborhood of the root on a set of nodes called the *i-roots*, which are defined as follows: the *i-root* of a key k is the node that becomes root of k when all j-roots of k for $j < i$ have failed or left the network. Thus, if r is the degree of replication, all copies of data are located on the i-roots $0 \leq i < r$ of the storage key.

At the basic functional level, all DHTs provide the following two functions to the application:

– the `put(b,k)` primitive stores a data block under the specified key k;

– the `b` $\leftarrow$ `get(k)` function returns a copy of the object inserted under key k.

Figure 3.6 illustrates the principle of storage in a DHT. Node 5230 calls the put primitive to store the block B with key 8959. A message is then routed until it reaches the node 8957, which is the root of key 8959 (i.e. the node in overlay whose ID is closest to 8959). Then the block is replicated three times in the neighborhood of the root (blocks B_0, B_1, and B_2). In this example, nodes 8957, 8954 and 895D, are the 0-root, 1-root and 2-root of block B, respectively.

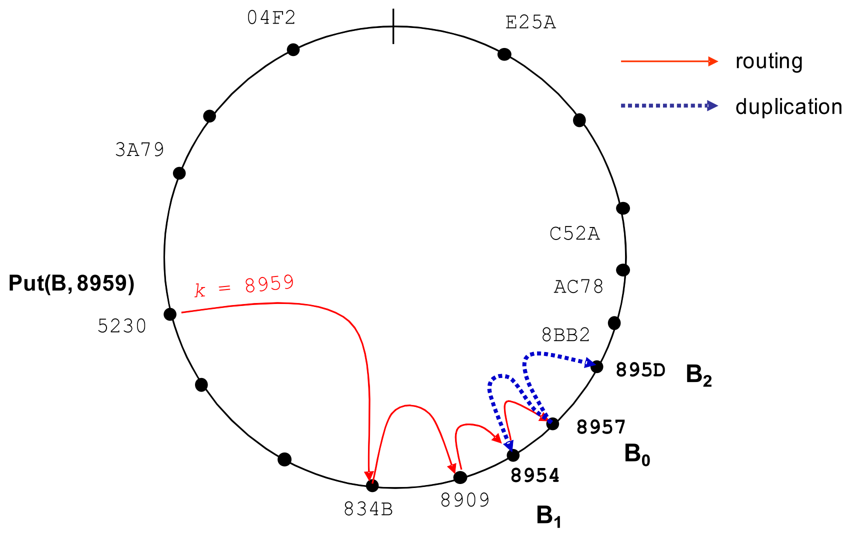


Figure 3.6. *Data insertion in a DHT*

3.4. Building trust in P2P systems

Building trust is a major concern in P2P systems [ABE 01, YU 04]. The self-organization of the nodes as well as the non-existent control of whether nodes can enter the system, makes the presence of malicious nodes inevitable. In this context, the self-organization and maintenance of the nodes, which appears to be initially one of the strongest features of a P2P system, becomes one of the most vulnerable areas because of the presence of malicious nodes.

A malicious node is a node that does not follow the normal rules of the system. It can be a byzantine node or a node that do not always behave maliciously (alternative nodes). The behavior of malicious nodes can have various consequences on the P2P system, as they can:

– reject queries and may not respond to *Get()* or *Put()* operations. The transaction will then fail;

– make the routing algorithm fail, denying the forwarding of messages (this is a typical behavior in DHT);

– provide false information to other nodes (data integrity problems);

– prevent other nodes from correctly achieving their maintenance tasks.

Using traditional solutions to build trust based on a set of servers providing certification or any kind of control over the nodes that are members of the system is impossible in P2P systems. The main reason is because such solutions are not scalable. It then becomes very difficult to manage networks that can grow from thousands to millions of nodes.

Building trust in P2P systems is then a very difficult task. In this context, two solutions have been shown to be feasible to ensure trust in P2P networks: reputation systems and accountability.

3.4.1. *Reputation systems*

First, let us note that reputation systems [RES 00, DAM 02] are very difficult to implement in an unstructured P2P network. This is mainly due to the lack of decidability in such networks, where finding a given dataset may fail even if this dataset exists in the network. Then we assume that we are in the context of a structured P2P network, such as a DHT [FED 07] (see 3.3.2).

The main idea of a reputation system is to associate a reputation $R(X)$ to each node X in the network. The reputation is then used by an application to decide wether or not to make a transaction with a given node. $R(X)$ can be seen as the probability for node X to be a trusted node, which is the probability for X to be honest. Then:

$$\begin{cases} P^{Trusted}(X) = R(X) \\ P^{Malicious}(X) = 1 - R(X) \end{cases}$$

It is important to understand that a reputation system is never able to fully decide whether a node can be trusted or not. In other words, $P^{Trusted}(X)$ or $P^{Malicious}(X)$ will never be 0 or 1. Therefore, there is always a risk attached to making a transaction with a malicious node, even if it has a high reputation value (high probability to be honest). A reputation system does not decide itself if a node is a trusted one. The threshold to decide if a node can be considered as honest or not is application dependent.

Every reputation system has to fulfill the following properties:

– reliability: the system must be able to discriminate between trusted and malicious nodes. That is, the computed reputation value for node X must correspond to a value that effectively represents the behavior of node X. This maintains the integrity and the completeness of the reputation information, so that no node will be able to modify or to delete its own reputation value;

– robustness: a reputation system can be the target of several types of attacks. A collusion of nodes can artificially decrease the reputation of a given peer, or even to increase the reputation of the colluded nodes themselves. Rumors and false recommendations are also common problems, where nodes can lie about the behavior of a node, or can generate false recommendations about transactions that have never occurred;

– scalability: as with every P2P network, a reputation system must be scalable and also self-organized to avoid central administration. However, every reputation system also has a limit regarding the number of malicious nodes it can handle. Outside this limit, the system usually becomes unstable and unable to discriminate among trusted and malicious nodes.

Various reputation systems have already been proposed like Pride [DEW 04], TrustMe [SIN 03], PeerTrust [XIO 04], TrustGuard [SRI 05], PowerTrust[ZHO 07], Pet [LIA 05], EingenTrust [KAM 03], or WTR [BON 09]. Each reputation system is based on a metric [SCH 05] to compute the reputation value, and possibly an associated risk.

As an example, we present in the following WTR (Worth The Risk) designed for DHT.

Reputation computation in WTR

The computation of the reputation in WTR is based on recommendations emitted by the clients after a transaction (see Figure 3.7).

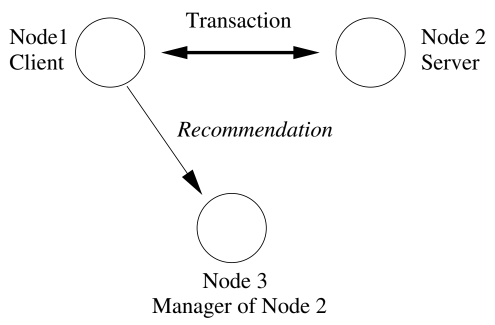


Figure 3.7. *The client side emits a recommendation after a transaction*

A recommendation represents the evaluation of the client regarding the transaction. In most of the existing reputation systems, a recommendation is a discrete value. This is done to mitigate the effect of similar evaluations from different nodes leading to different recommendation values. Table 3.2 lists the recommendation values used in WTR.

A set of nodes is used to manage the reputation of node X. In a DHT, a typical set of nodes can be chosen using the following method. A root manager M_0 is assigned

Evaluation	Recommendation	Description
Excellent	1	Very good transaction and service fully completed.
Good	0.75	Good transaction but service has suffered some degradation.
Neutral	0.5	The transaction has not completed correctly but the reason could be independent of the node.
Bad	0.25	Transaction and service not completed. Maybe a malicious node.
Malicious	0	Fully malicious behavior of the node.

Table 3.2. *Possible Recommendation Values in WTR*

using $M_0 = H(X)$ where H is the hash function used to build the DHT. Then the set of managers M_i must be chosen such that they will be readily available. An easy way to do this in a DHT-like Pastry [ROW 01b] is to chose the leaf set of M_0 as the set of managers, or the n successors of M_0 in Chord [STO 01a].

Each manager receives a copy of the recommendation emitted by the client of node X, and maintains a list of recommendations for X. The latest m recommendations are used to compute the reputation (sliding window).

$$R_T(X) = \frac{\sum_{i=0}^{m-1} log(m-i+1) \times F_i^K(X) \times C_t(K)}{\sum_{i=0}^{m-1} log(m-i+1) \times C_t(K)} \tag{3.1}$$

Equation (3.1) shows the computation of the reputation of node X at time T by a manager of X. $F_i^K(X)$ represents the recommendation of index i, emitted by node K in the recommendation list of node X. The log factor is used as a fading factor, in order to give a higher weight to the latest recommendations for X, lowering the influence of the old ones. $C_t(K)$ represents the credibility of node K when emitting the recommendation at time t. The reputation is then normalized between 0 and 1.

The credibility of K is computed as follow:

$$C_t(K) = \begin{cases} 1 & if\, R_t(K) \in [0.75 \ldots 1] \\ 0.75 & if\, R_t(K) \in [0.5 \ldots 0.75[\\ 0.2 & if\, R_t(K) \in [0.25 \ldots 0.5[\\ 0.1 & if\, R_t(K) \in [0 \ldots 0.25[\end{cases}$$

The credibility is a function of K's reputation at time t. This is an important point in reputation systems, where recommendations coming from malicious nodes should not influence the recommendation of honest ones. Note that the credibility scale is not linear. This is done to heavily mitigate recommendations from untrusted nodes.

The initial reputation value of a node is an important problem in reputation systems. There are two existing policies for the initial value:

– negative discrimination: this consists in assigning a low reputation value to a new node that enters the network;

– positive discrimination: this consists in assigning a high reputation value to a new node that enters the network.

The main advantage of negative discrimination is to prevent a potential malicious node undertaking a malicious transaction. Nevertheless, negative discrimination does not give the opportunity to a new honest node to make a transaction, as it will not appear as a good candidate in the system. Subsequently, most of the existing reputation systems use positive discrimination for new nodes to give them the opportunity to make honest transactions.

In WTR, positive discrimination is controlled by the reputation computation (equation (3.1)) where it is more difficult for a node to increase its reputation than to decrease it. A new node will have a high reputation initially, but will rapidly become untrusted with a very few number of malicious transactions.

Computation of the risk

The reputation of a node is not always sufficient to decide wether or not to make a transaction with this node. For example, if the reputation of node X has been computed using very few recommendations (for example, only two recommendations), the reputation does not necessarily represent the real behavior of X. The notion of risk associated with the reputation enables the application to decide whether it is worth the risk to make a transaction with X, even if it has a good reputation. Most of the existing reputation systems integrate the notion of risk.

In WTR, the risk is computed as shown in the equation below:

$$J_t(X) = \underbrace{\alpha\left(1 - \frac{r}{m}\right)}_{T_1} + \underbrace{4(1-\alpha)\frac{\sum_{i=0}^{k}\left(F_i(X) - \overline{F_i(X)}\right)^2}{r}}_{T_2} \tag{3.2}$$

T_1 is used to evaluate the risk associated with the number of recommendations taken into account to compute the reputation of node X. The smaller the number of available recommendations r in the sliding window of X, the higher the risk (i.e. the less reliable) the computed reputation value;

T_2 corresponds to the variance of the recommendations emitted by other nodes. The key idea of T_2 is to enable the discrimination of nodes that do not have regular behavior. The role of the factor 4 is to normalize T_2 to obtain a value in $[0 \ldots 1]$.

The application can chose the value of α in order to give more weight to T_1 or T_2.

Security and robustness considerations

A reputation system must resist a typical attack from malicious nodes, or from a collusion of malicious nodes. We present here the set of attacks that WTR can handle:

– false recommendations: to deal with this problem, the credibility value has been introduced. Recommendations emitted by a node with poor credibility will have a minor effect on the computation of the reputation of X. This prevents the reputation system from being liable to malicious nodes that emit false recommendations. This also avoids part of the collusion attack (see below);

– rumors: a rumor occurs when a recommendation is emitted by node A for node X and no transaction has been undertaken between A and X. This problem is not directly considered here, but is quite easy to take into account. A good and efficient solution has been presented in TrustGuard where evidence of the transaction is collected and a recommendation is accepted only if the transaction has actually been be executed;

– collusion of nodes: this is one of the most difficult problems for reputation systems to which few have any solution. As it is highly probable that a collusion attack will come from already malicious nodes, the WTR reputation metric will be able to detect this using the credibility of the nodes involved in the attack. The recommendation emitted by these nodes will have a very small effect on the reputation computation, and it will be near impossible for a set of malicious nodes to succeed in artificially decreasing the reputation of a given node. As in most cases, colluded nodes will try to use rumors, this is easily to managed using evidence of transactions. Nevertheless, if a collusion attack is started by trusted nodes (that change their behavior), it is nearly impossible to detect as they will appear as credible nodes;

– white washers: a white washer is a node that leaves the network and joins again using a different identity. White washers represent a common problem in P2P networks and are quite a difficult task to manage for reputation systems. The risk factor mitigates the effect of white washers. Leaving and rejoining the network with a new identity implicitly clears the recommendation history of a node. Choosing a high value for the α parameter will give more weight to the term T_1 of the risk factor. Therefore, it will be more risky to make a transaction with a white washer that should only register very few transactions due to its dynamicity, leaving and rejoining the network;

– data integrity: as each node generates a self-certificate, the public key of each manager is used to encrypt the recommendations stored by this manager. This ensure that no other node will be able to modify this information to alter the recommendations. Thus, the managers of node X is the only node that can decrypt the recommendation list to compute the reputation for X.

Scalability and portability considerations

A good reputation system must have a metric that does not rely on the network size. Moreover, the reputation information must be readily available and should resist attacks such as strong distributed denial of service attacks (DDoS). In order to resist strong DDoS, a reputation system must provide a way to store a certain number of replicas of the reputation information at an acceptable cost in terms of the number of messages to access the information. WTR is a reputation system that provides this facility.

WTR uses a technique called recursive replication [BON 07] to increase the number of replicas of the reputation information, without increasing the maintenance cost of the overlay. The main idea is to make several levels of replicas using a recursive hash computation. The recursive hash function is defined as follows:

$$M_i = H^i(X) = \underbrace{H(H(\ldots H(X)))}_{i\ times}$$

The i reputation managers for node X are chosen such that the root managers are:

$$\begin{cases} M_1 = H^1(X) \\ M_2 = H^2(X) \\ \quad\ldots \\ M_i = H^i(X) \end{cases}$$

Then, all nodes of the leaf set of M_i in Pastry, or the n successors of X in Chord become a manager of X's reputation.

Performance of reputation systems

A reputation system is never able to fully decide if a node is honest or not, as the reputation value only represents the probability of a node to be honest. However, there is also a limit to the number of malicious nodes a reputation system can handle. We say that the reputation system collapses when it can no longer discriminate honest nodes from malicious ones.

The practical limit for all existing reputation systems is around 30% of malicious nodes. Over this limit, the influence of malicious nodes is too high for the reputation system to be able to provide credible results. WTR provides one of the best known results. The system is completely stable up to the point where 30% of the nodes are malicious, and collapses after 40%.

3.4.2. *Accountability*

Accountability is a very powerful technique to build trust in large-scale distributed systems, but it is also a slightly complex one. Therefore, for space reasons, it is not possible to provide full details regarding how accountability works and can be implemented. Thus, we will give some global ideas on how accountability works. For a complete description, please refer to [HAE 07, DRU 08].

Accountability is very well adapted to DHT. The key idea is to be able to verify whether a node has honest behavior (i.e. whether a node is doing what it is supposed to). In other words, accountability enables the results of a transaction to be verified, according to its input parameters.

The principle relies on a set of witnesses that will check whether the result of a transaction between two nodes corresponds to the expected one. Suppose that we want to check the behavior of node X. A set of witnesses is chosen for node X. Typically, a node is a witness for a set of the numerically closest nodes in the system. For example, in Pastry, a node is a witness for all the nodes in its leaf set. In Chord, a node can be a witness for its n successors. Therefore, a node X has various witnesses in the network.

The goal of each witness is to collect evidence that node X is faulty, to declare X as untrusted, or to declare it as honest or trusted.

Log and witnesses

Each node of the network maintains a log that contains all the input messages received by the node, as well as all the results produced by the node. The log is managed using strong cryptographic techniques and has the following properties:

– only node X can add events to its own log;

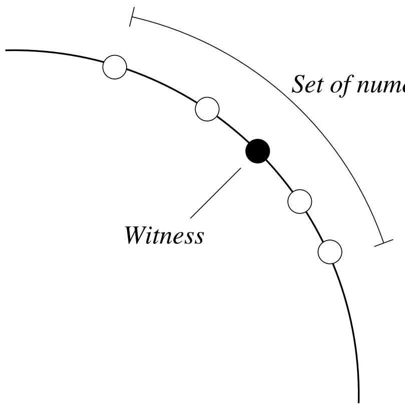


Figure 3.8. *The set of witnesses in accountability*

– node X cannot modify its own log, and any other node cannot do it. If a log of X has been altered, it is detected by the witnesses of X;

– events can only be added to the log, and events are time stamped by X.

Collecting evidence

All witnesses of node X can access its log. The use of strong cryptographic techniques for data integrity management allow the witnesses to detect whether X's log has been altered by X. This constitutes evidence that X is faulty and must be considered as untrusted. *expose(X)* is raised. Each Δt period of time, the witnesses make an audit of node X using its log. Auditing X requires the following assumptions:

– the algorithm or protocol that X runs is a deterministic state machine;

– a witness knows the protocol ran by X, and also has a detector module to check the correctness of the state of X.

The detector module detects whether messages transmitted by X are not correct (ordering problem, lack of message, etc.). It predicts what the messages produced by the state machine of X should be, and verifies that they match the prediction. If a witness detects evidence that the produced messages are not correct, then X is declared as unstrusted (an *expose(X)* is raised).

Due to message transmission delays, a message emitted by X could arrive late. When a witness detects that delay, *suspected(X)* is raised.

As the witnesses know the state machine used by X, they can use the log during the audit to run the same protocol or algorithm with the same messages received by X as the input. If the results obtained by the witnesses are different from those in X's log, then evidence that X is faulty is collected. The witnesses declare X as untrusted, and *exposed(X)* is raised.

Size of the witness set

Let us suppose that N is the size of the network, and φ is the fraction of malicious nodes. If there is a small probability $P_f > 0$ that an all-faulty witness set exists, then ψ defines the number of witnesses that must be assigned to each node.

$$\psi = \left\lceil \frac{ln(1-(1-P_f)^{\frac{1}{N}}}{ln\varphi} \right\rceil$$

The number of witnesses ψ grows with $O(logN)$, which makes accountability highly scalable for P2P networks.

3.4.3. *Discussion on reputation sytems and accountability*

The major problem of reputation systems is that they are not able to fully decide wether or not a node can be trusted. The reputation only represent the probability for the node to be trusted, and the final choice always depends on the application requirements. However, reputation systems can be used for all kind of applications, whatever the protocol used by the nodes during the transactions. The only requirement is to have an upper limit for the fraction of malicious nodes (usually around 30%).

Conversely, accountability appears to be much more powerful with the ability to fully decide whether a node can be trusted or not, by collecting evidence of faults. Nevertheless, all witnesses need to know the complete protocol used by nodes during a transaction (message protocol and state machine corresponding to the algorithm ran by a node). Accountability also has a higher cost in terms the amount of data that nodes need to manage.

Both methods are highly scalable and can easily support networks comprising millions of nodes.

3.5. Conclusion

P2P systems have been shown to be a good solution to build large-scale distributed systems. Unstructured P2P networks such as Gnutella, are very popular, as they provide a good level of anonymity, and usually, the possibility for the node to choose which data it accepts to store. The main problem of unstructured P2P networks is the non-decidability of the search algorithm, which may not find existing data in the network. Conversely, structured P2P networks such as DHT, provide a high availability of data, and a search method that is decidable. Structured networks usually have better performance than unstructured networks but lack anonymity.

In all P2P networks, building trust is a major problem. Reputation systems and accountability represent a great advance in this context, but much work is still required. One of the main future research areas will be to enable some kind of pseudo-certification authority in P2P networks, able to certify that a given event has occurred with a very low probability of failure.

3.6. Bibliography

[ABE 01] ABERER K., DESPOTOVIC Z., "Managing trust in a peer-2-peer information system", *CIKM '01: Proceedings of the Tenth International Conference on Information and Knowledge Management*, New York, NY, ACM Press, p. 310–317, 2001.

[BHA 04] BHARGAVA A., KOTHAPALLI K., RILEY C., SCHEIDELER C., THOBER M., "Pagoda: a dynamic overlay network for routing, data management, and multicasting", *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA'2004 (Barcelona, Spain, June 27-30, 2004)*, New York, ACM SIGACT, ACM SIGARCH, ACM Press, p. 170–179, 2004.

[BON 07] BONNAIRE X., MARIN O., "Recursive Replication: A Survival Solution for Structured P2P Information Systems to Denial of Service Attacks", *PPN '07: Proceedings of the 1st International Workshop on Peer to Peer Networks*, Lecture Notes in Computer Science 4806, Springer-Verlag, p. 931–940, 2007.

[BON 09] BONNAIRE X., ROSAS E., "WTR: a reputation metric for distributed hash tables based on a risk and credibility factor", *J. Comput. Sci. Technol*, vol. 24, p. 844–854, 2009.

[BUS 05] BUSCA J.-M., PICCONI F., SENS P., "Pastis: a highly-scalable multi-user peer-to-peer file system", *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference (11th Euro-Par'05)*, vol. 3648 of *Lecture Notes in Computer Science*, Lisbon, Portugal, Springer-Verlag, p. 1173–1182, August -September 2005.

[CAP 05] CAPPELLO F., DJILALI S., FEDAK G., HÉRAULT T., MAGNIETTE F., NÉRI V., LODYGENSKY O., "Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid", *Future Generation Comp. Syst*, vol. 21, p. 417–437, 2005.

[CAS 02] CASTRO M., DRUSCHEL P., KERMARREC A., ROWSTRON A., "SCRIBE: a large-scale and decentralized application-level multicast infrastructure", *IEEE Journal on Selected Areas in communications (JSAC)*, vol. 20, p. 1489–1499, 2002.

[DAB 01] DABEK F., BRUNSKILL E., KAASHOEK M. F., KARGER D. R., MORRIS R., STOICA I., BALAKRISHNAN H., "Building peer-to-peer systems with Chord, a distributed lookup service", *HotOS*, IEEE Computer Society, p. 81–86, 2001.

[DAB 03] DABEK, ZHAO, DRUSCHEL, KUBIATOWICZ, STOICA, "Towards a common API for structured peer-to-peer overlays", *International Workshop on Peer-to-Peer Systems (IPTPS)*, vol. 2, 2003.

[DAM 02] DAMIANI E., DI VIMERCATI D. C., PARABOSCHI S., SAMARATI P., VIOLANTE F., "A reputation-based approach for choosing reliable resources in peer-to-peer networks", *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, New York, NY, USA, ACM Press, p. 207–216, 2002.

[DEW 04] DEWAN P., DASGUPTA P., "Pride: peer-to-peer reputation infrastructure for decentralized environments", *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, New York, NY, USA, ACM Press, p. 480–481, 2004.

[DRU 08] DRUSCHEL P., "Accountability for distributed systems", BAZZI R. A., PATT-SHAMIR B., Eds., *Symposium on Principles of Distributed Computing*, ACM, p. 13–14, 2008.

[FED 07] FEDOTOVA N., BERTUCCI M., VELTRI L., "Reputation management techniques in dht-based peer-to-peer networks", *ICIW'07: Second International Conference on Internet and Web Applications and Services*, Page 4, IEEE Computer Society, 2007.

[FRA] FRAIGNIAUD P., GAURON P., "D2B: a de Bruijn based content-addressable network", *TCS: Theoretical Computer Science*, vol. 355, p. 65-79.

[GKA 06] GKANTSIDIS C., MIHAIL M., SABERI A., "Random walks in peer-to-peer networks: algorithms and evaluation", *Perform. Eval.*, vol. 63, p. 241–263, Elsevier Science Publishers B. V., March 2006.

[GUM 03] GUMMADI K., GUMMADI R., GRIBBLE S., RATNASAMY S., SHENKER S., STOICA I., "The impact of DHT routing geometry on resilience and proximity", *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, ACM, p. 381–394, 2003.

[HAE 07] HAEBERLEN A., KOUZNETSOV P., DRUSCHEL P., "PeerReview: practical accountability for distributed systems", *Proceedings of the 21st ACM Symposium on Operating Systems Principles (21st SOSP'07)*, Stevenson, Washington, USA, ACM SIGOPS, p. 175–188, October 2007.

[JEA 07] JEANVOINE E., RILLING L., MORIN C., LEPRINCE D., "Using overlay networks to build operating system services for large scale grids", *Scalable Computing: Practice and Experience*, vol. 8, p. 229–239, 2007.

[JOV 01] JOVANOVIC M., ANNEXSTEIN F., BERMAN K., "Scalability issues in large peer-to-peer networks - a case study of gnutella", 2001.

[KAA 03] KAASHOEK F., KARGER D., "Koorde: a simple degree-optimal distributed hash table", *International Workshop on Peer-to-Peer Systems (IPTPS)*, vol. 2735 of *Lecture Notes in Computer Science*, 2003.

[KAM 03] KAMVAR S. D., SCHLOSSER M. T., GARCIA-MOLINA H., "The Eigentrust algorithm for reputation management in P2P networks", *Proceedings of the 12th international conference on World Wide Web*, New York, NY, USA, ACM Press, p. 640–651, 2003.

[KUB 00] KUBIATOWICZ J., BINDEL D., CHEN Y., EATON P., GEELS D., GUMMADI R., RHEA S., WEATHERSPOON H., WEIMER W., WELLS C., ZHAO B., "OceanStore: an architecture for global-scale persistent storage", *Proceedings of ACM ASPLOS*, ACM, November 2000.

[LI 06] LI X., WU J., “Searching techniques in peer-to-peer networks”, *In Handbook of Theoretical and Algorithmic Aspects of Ad Hoc, Sensor, and Peer-to-Peer Networks*, CRC Press, p. 613–642, 2006.

[LIA 05] LIANG Z., SHI W., “PET: a personalized trust model with reputation and risk evaluation for P2P Resource Sharing”, *HICSS ’05: Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, Washington, DC, USA, IEEE Computer Society, Page201.2, January 2005.

[LUA 05] LUA K., CROWCROFT J., PIAS M., SHARMA R., LIM S., “A survey and comparison of peer-to-peer overlay network schemes”, *Communications Surveys & Tutorials, IEEE*, p. 72–93, 2005.

[MAL 02] MALKHI D., NAOR M., RATAJCZAK D., “Viceroy: a scalable and dynamic emulation of the butterfly”, *Symposium on Principles of Distributed Computing*, p. 183–192, 2002.

[MAY 02] MAYMOUNKOV, MAZIERES, “Kademlia: a peer-to-peer information system based on the XOR metric”, *International Workshop on Peer-to-Peer Systems (IPTPS)*, vol. 2429 of *Lecture Notes in Computer Science*, 2002.

[MUT 02] MUTHITACHAROEN A., MORRIS R., GIL T. M., CHEN B., “Ivy: A read/write peer-to-peer file system”, *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI-02)*, Operating Systems Review, New York, ACM Press, p. 31–44, December 9–11 2002.

[PLA 99] PLAXTON C. G., RAJARAMAN R., RICHA W., “Accessing nearby copies of replicated objects in a distributed environment”, *MST: Mathematical Systems Theory*, vol. 32, 1999.

[RAT 01] RATNASAMY S., FRANCIS P., HANDLEY M., KARP R. M., SHENKER S., “A scalable content-addressable network”, *ACM SIGCOMM*, p. 161-172, 2001.

[RES 00] RESNICK P., KUWABARA K., ZECKHAUSER R., FRIEDMAN E., “Reputation systems”, *Communications of the ACM*, vol. 43, num. 12, p. 45–48, ACM Press, December 2000.

[RHE 05] RHEA S. C., GODFREY B., KARP B., KUBIATOWICZ J., RATNASAMY S., SHENKER S., STOICA I., YU H., “OpenDHT: a public DHT service and its uses”, *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA, August 22-26, 2005*, ACM, p. 73–84, 2005.

[ROU 04] ROUSSOPOULOS M., BAKER M., ROSENTHAL D. S. H., GIULI T. J., MANIATIS P., MOGUL J. C., “2 P2P or Not 2 P2P?”, *IPTPS ’04: Proceeding of the third International workshop on Peer-To-Peer Systems*, p. 33-43, 2004.

[ROW 01a] ROWSTRON A., DRUSCHEL P., “Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems”, *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[ROW 01b] ROWSTRON A. I. T., DRUSCHEL P., “Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems”, *Middleware ’01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, vol. 2218 of *Lecture Notes in Computer Science*, Springer-Verlag, p. 329–350, 2001.

[SCH 05] SCHLOSSER A., VOSS M., BRÜCKNER L., “On the simulation of global reputation systems”, *Journal of Artificial Societies and Social Simulation*, vol. 9, num. 1, Page 4, January 2005.

[SIN 03] SINGH A., LIU L., “TrustMe: anonymous management of trust relationships in decentralized P2P”, *P2P ’03: Proceedings of the IEEE International Conference on Peer-to-Peer Computing*, Washington, DC, USA, IEEE Computer Society, Page142, September 2003.

[SRI 05] SRIVATSA M., XIONG L., LIU L., “TrustGuard: countering vulnerabilities in reputation management for decentralized overlay networks”, *WWW ’05: Proceedings of the 14th international conference on World Wide Web*, New York, NY, USA, ACM Press, p. 422–431, 2005.

[STO 01a] STOICA I., MORRIS R., KARGER D., KAASHOEK M., BALAKRISHNAN H., “Chord: a scalable peer-to-peer lookup service for internet applications”, *SIGCOMM ’01: Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, ACM Press, p. 149–160, 2001.

[STO 01b] STOICA I., MORRIS R., KARGER D. R., KAASHOEK M. F., BALAKRISHNAN H., “Chord: A scalable peer-to-peer lookup service for internet applications”, *SIGCOMM*, p. 149–160, 2001.

[SUL 97] SULLIVAN W. T., WERTHIMER D., BOWYER S., COBB J., GEDYE D., ANDERSON D., “A new major SETI Project based on project SERENDIP data and 100,000 Personal Computers”, *Proceedings of the 5th Int. Conf. on Bioastronomy*, Bologna, Italie, Editrice Compositori, 1997.

[VIL 04] VILLAMIL M.-D.-P., RONCANCIO C., LABBÉ C., “PinS: peer-to-peer interrogation and indexing system”, *IDEAS*, IEEE Computer Society, p. 236–245, 2004.

[WAL 00] WALDMAN M., RUBIN A. D., CRANOR L. F., “Publius: a robust, tamper-evident, censorship-resistant web publishing system”, USENIX, Ed., *Proceedings of the Ninth USENIX Security Symposium, 2000, Denver, Colorado*, pub-USENIX:adr, USENIX, August 2000.

[XIO 04] XIONG L., LIU L., “PeerTrust: supporting reputation-based trust for peer-to-peer electronic communities”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, num. 7, p. 843–857, IEEE Educational Activities Department, 2004.

[YU 04] YU B., SINGH M., SYCARA K., “Developing trust in large-scale peer-to-peer systems”, *MAS&S2004: Proceedings og the IEEE First Symposium on Multi-Agent Security and Survivability*, Philadelphia, Pennsylvania, USA, p. 1–10, August 2004.

[ZEI 04] ZEINALIPOUR-YAZTI D., KALOGERAKI V., GUNOPULOS D., “Information retrieval techniques for peer-to-peer networks”, *Computing in Science and Engineering*, vol. 6, p. 20-26, IEEE Computer Society, 2004.

[ZHA 02] ZHAO B. Y., KUBIATOWICZ J., JOSEPH A. D., "Tapestry: a fault-tolerant wide-area application infrastructure", *Computer Communication Review*, vol. 32, Page 81, 2002.

[ZHO 07] ZHOU R., HWANG F.-K., "PowerTrust: a robust and scalable reputation system for trusted peer-to-peer computing", *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, num. 4, p. 460–473, IEEE Press, April 2007.

Chapter 4

Peer-to-Peer Storage

4.1. Introduction

Peer-to-peer storage applications are the main actual implementations of large-scale distributed software. A peer-to-peer storage application offers five main operations: a lookup operation to find a file, a read operation to read a file, a write operation to modify a file, an add operation to inject a new file and a remove operation to delete a file. However, most current peer-to-peer storage applications are limited to file sharing: they do not implement the write and the delete operations.

Implementing the lookup and add operations is challenging in a peer-to-peer storage application because it does not provide a centralized host that associates file names with their corresponding locations: the file name database is distributed among the clients. Implementing the read operation is also challenging because it must scale with the number of readers.

Two applications are presented in this chapter: BitTorrent and Gnutella. BitTorrent only provides the read operations but it proposes an efficient algorithm that scales with the number of readers. Gnutella provides the add and lookup operations. The first version of Gnutella relies on a gossip protocol while the second version is semi-distributed: some clients are connected through a gossip protocol and others are connected through a classical client/server architecture.

Chapter written by Olivier MARIN, Sébastien MONNET and Gaël THOMAS.

Distributed Systems: Design and Algorithms Edited by Serge Haddad, Fabrice Kordon, Laurent Pautet and Laure Petrucci
© 2011 ISTE Ltd. Published 2011 by ISTE Ltd.

4.2. BitTorrent

BitTorrent [COH 11, BIT 11] is a peer-to-peer file-sharing protocol that scales with the number of clients. It is specifically designed to handle large files such as movies or bulky software – for instance, linux distributions and games. The main idea behind BitTorrent is: "divide and conquer". As soon as a client has downloaded part of a file, it becomes a new server for this part. Other clients can then download this piece directly from this client. This mechanism avoids the bottleneck induced by a centralized server.

Studying BitTorrent is particularly interesting because it is one of the most efficient communication protocols enabling large files to be broadcast. This mainly explains its success: a study undertaken in February 2009 [SCH 09] estimates that BitTorrent generates between 27% and 55% of all Internet traffic – this value depends on geographical location.

The BitTorrent protocol relies on three main principles in order to ensure its performances [LEG 06]:

– tit-for-tat: a peer sends data to another peer only if the latter has given enough in exchange. This principle ensures the fairness of the BitTorrent network;

– priority to rare pieces: a peer downloads the rare pieces of the file first; rare pieces are those that are hosted only on a small subset of the peers. This principle ensures that even in the presence of churn (frequent connections/disconnections), all the pieces of the file are present in the BitTorrent network;

– optimistic unchoke: the tit-for-tat principle prevents a new peer which has nothing to share, from downloading its first pieces. The optimistic unchoke principle counterbalances the tit-for-tat principle by periodically selecting a random peer and sending it data freely.

4.2.1. *History*

The ancestor of BitTorrent is MojoNation, a communication protocol created by the "Evil Geniuses for a Better Tomorrow" company founded by Jim McCoy in 2000. MojoNation proposed to split a file into small pieces to avoid the bottleneck of a centralized server. The idea of tit-for-tat was already there, but with virtual money: each peer increased its credit by transferring requests of other peers. However, transferring the money degraded the performance as it requires authentication and encryption. The company went bankrupt in 2002. Bram Cohen left MojoNation in 2001 and began the development of BitTorrent. The first BitTorrent specification was presented at the CodeCon conference in 2002 in San Francisco [COH 02].

Until 2004, Bram Cohen's main income was based on donations collected through his web site (`http://bittorrent.org`), which provides the specification and free implementation of BitTorrent. In 2004, he founded the commercial company called BitTorrent, Inc. (`http://bittorrent.com`) with his brother Ross Cohen and a business partner Ashwin Navin. BitTorrent, Inc. sells its services to movie studios.

One of the main original aspects of the BitTorrent specification is that its evolution has never broken the ascendant compatibility. This strength comes from the specification, which does not provide any algorithm as to the behavior of the peers but only defines their intention. Of course, the communication protocol (i.e. the formats of the messages) is fixed and cannot evolve. However, the way a peer takes a decision with respect to the information that it collects about the network is unspecified and varies from one implementation to another. Research is still being undertaken to improve the quality of the algorithms used by the peers.

4.2.2. *Terminology*

BitTorrent introduces a specific terminology to describe the architecture of the network:

– the swarm: the set of all the peers that are downloading or sharing a file;

– a piece: a piece of a file;

– the tracker: a centralized server that has a rough knowledge of the swarm. It knows the peers of the swarm and for every one of them, the pieces they host, which is only used for statistical purposes;

– a client: a peer that does not have all the pieces of the file. A client is therefore a peer that is trying to download new pieces of the file;

– a seeder: a peer that owns all the pieces of the file. When a client completes its download, it becomes a seeder;

– a leecher: a client that is only downloading. Most of the time, a leecher is simply a client that does not have any piece to share. However, a leecher could also be a selfish client.

In this section and the next, we reserve the term “peer” to name a client or a seeder.

4.2.3. *Joining a BitTorrent network*

When a peer wants to join a BitTorrent network, it contacts the tracker of the file. Finding the tracker is out of the scope of the protocol because BitTorrent does not address the problem of file lookup. Most of the time, classical HTTP servers maintain an association between file names and trackers. When the tracker receives a request from an incoming peer, it replies with a list of network peers (generally about 50).

Figure 4.1 illustrates the mechanism. Initially, seeder S_1 contacts the tracker. The tracker sends an empty list to S_1, but is now aware of the existence of S_1 and adds it to its own list. Afterwards, when C_1 contacts the tracker, it responds with its new list containing only S_1 and adds C_1 locally. Finally, as C_2 contacts the tracker it responds with the list $\{S_1, C_1\}$. In Figure 4.1, C_1 and C_2 already have some pieces of the file, this can happen when a client reconnects to the network after a disconnection.

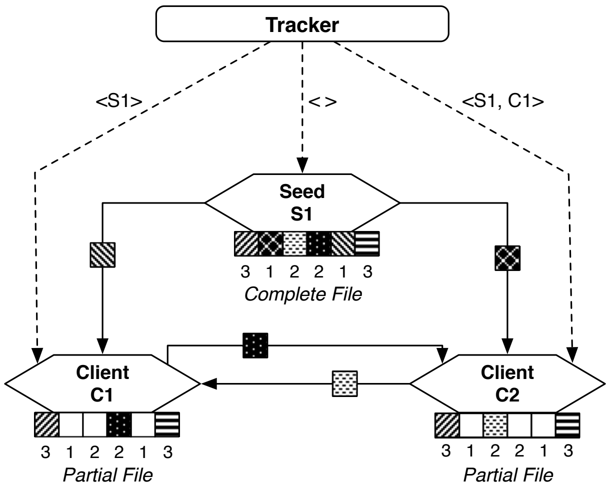


Figure 4.1. *Exchange of pieces*

4.2.4. *Making the pieces available*

Once a peer has acquired a list of neighbors from the tracker, it contacts all of them. A connection is bidirectional: when peers $N1$ and $N2$ are connected, they can exchange pieces of the file in both directions. In Figure 4.1, the graph is complete because C_1, S_1 and S_2 are all connected to one another. Such is generally not the case: neighborhoods do not span over the entire network as several thousands of peers can participate in the network while each peer only has around 50 neighbors.

As soon as two peers are connected, they exchange their list of available pieces. This list is used to determine which peer has which piece, but also to determine which pieces are sparse on the network. As the graph is not complete, a peer only estimates the availability; statistically, this estimation is correct with 50 peers. In Figure 4.1, the estimation corresponds to the real availability because the graph is complete.

When a peer downloads a new piece, it broadcasts this information to its neighbors and they update their availability lists.

A peer maintains three pieces of information about each of its neighbors:

– interested: if a peer C_1 is interested in one piece of C_2, then C_1 notifies C_2 and C_2 registers that C_1 is interested, otherwise, it registers that C_1 is not interested;

– choked: if a peer C_1 is interested in one piece of a peer C_2 and if C_2 cannot send data to C_1 because it already sends pieces to other neighbors, then C_2 registers that C_2 is chocked and notifies it. When C_2 sends data to C_1, it marks C_1 as unchoked and notifies it;

– snubbed: if a peer C_2 optimistically sends a piece to C_1, then C_1 is marked as snubbed for a short duration (see next section).

Figure 4.2 presents the state of two clients C_1 and C_2. C_2 is interested by a piece of C_1 (the third piece) and C_1 is not interested by any piece of C_2.

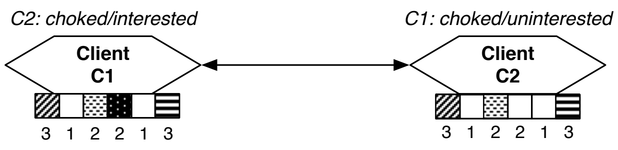


Figure 4.2. *State of two neighbors*

4.2.5. *Priority to rare pieces*

A peer will first try to download the rarest pieces offered by its neighbors. By minimizing the number of rare pieces, BitTorrent uniformly distributes the load on the peer and, therefore, increases the performances. In Figure 4.1, C_1 will therefore download the second piece from S_1 because it is the rarest piece offered by S_1. For the same reason, C_1 will also download the third piece from C_1. Symmetrically, C_2 will download the fourth piece from C_1 and the last piece from S_1 (the piece is chosen randomly among the rarest pieces, see below).

Giving a higher priority to rare pieces has a drawback. At some point, many clients can start downloading the same piece considered as the rarest. The result is counter-productive because these clients may therefore be unable to exchange other pieces amongst themselves. For example, on the Figure 4.1, the protocol must try to avoid the case where C_1 and C_2 download the same piece. To counterbalance this effect, a client randomly choses one of the pieces among the N rarest pieces where N is equal to the number of active connections (i.e. number of neighbors).

Distributing the rarest piece first increases the performance of a BitTorrent network but also has another advantage: as the seeders uniformly distribute their pieces, their presence is requested for a short time. For example, after the exchange represented in the Figure 4.1, C_1 and C_2 can rebuild all the files even in the absence of the sender S_1.

4.2.6. *The tit-for-tat principle*

Neighbors exchange pieces by using a *tit-for-tat* algorithm: the more data a peer sends to its neighbors, the more data are sent by its neighbors. This algorithm forces the peer to send data: according to the tit-for-tat principle if a leecher only wants to download without sharing, it will not be serviced by the other peers.

To avoid overloading a peer, a peer only sends pieces to a small subset of its neighbors (normally four), other neighbors are just waiting (they are choked). To ensure the tit-for-tat algorithm, a peer choses its best neighbors and sends them pieces of the file.

The BitTorrent protocol uses one single criteria to find its best neighbors: *the download rate*. The download rate is equal to the number of bytes received by a neighbor plus one divided by the number of bytes received by this neighbor plus one. A client, i.e. a peer that does not have all the pieces, elects its best neighbors by choosing the neighbors with the highest download rates.

However, this criteria is useless for a seeder as it does not receive data. The goal of a seeder is different: it will try to give its piece as fast as possible. As a consequence, the more a neighbor downloads, the more it is interesting. A seeder therefore elects its neighbors with the smallest download rate. A client with a high network rate is favored because it can send its pieces faster.

As soon as a peer sends data to one of its neighbors, the download rate of this neighbor decrease. If two neighbors have approximately the same download rate and if the peer can only serve one if them, a fibrillation phenomenon could appear because the best neighbor will quickly become the worst. To avoid this phenomenon, the protocol defines rounds. A peer choices its best neighbors at the beginning of a round and does not change them if their download rates change during a complete round. By default, a round is 10 seconds long. Notice that a round is too short to receive a complete piece and that only smaller data, called data blocks, are exchanged. However, a peer can choke a client during a round if a best neighbor is not elected at the beginning of the round because it is not interested. In this case, if the neighbor becomes interested, the worst unchoked neighbor is choked to unchoke the best neighbor.

4.2.7. *The optimistic unchoke principle*

However, applying the tit-for-tat algorithm raises two problems:

– a client without any pieces will never obtain a piece if all its neighbors are clients;

– only the clients that have a lot of pieces will be elected as they can send a lot of data.

To solve this problem, a client reserves a slot to send data to a randomly elected neighbor. We say that the neighbor is optimistically unchoked. The optimistic unchoke algorithm is in contradiction to the tit-for-tat principle and an experience from 2006 [LOC 06] shows that a leecher can download a file without sending any pieces. However, it takes four time longer to download the file. By default, a client chooses optimistically a peer to unchoke every three rounds. At this round, if the client has four slots to send data, it chooses the three best neighbors and chooses the last one randomly.

When a client is served because it was optimistically unchoked, it is then marked as *snubbed* for a small period. When a client is snubbed it can not be elected by the optimistic unchoke algorithm.

4.2.8. *The messages of the protocol*

A message begins with a sequence of 4 bytes that gives the length of the message. Then, the message contains the type of message and possibly its content:

– `choked` (0): message of length 1 sent by a peer to inform one of its neighbors that it is choked and will not receive any data. This message follows an election;

– `unchoked` (1): message of length 1 sent by a peer to inform one of its neighbors that it is unchoked and will receive data. This message follows an election;

– `interested` (2): message of length 1 sent by a peer to inform one of its neighbors that it is interested in one of its pieces. This message follows a `bitfield` or a `have` message;

– `not interested` (3): message of length 1 sent by a peer to inform one of its neighbors that it is not interested in any of its pieces. This message follows a `bitfield` or a `piece` message from another peer;

– `have` (4): message of length 5 sent by a peer to inform its neighbors that it has the piece number encoded by the last four bytes. This message follows a `piece` message when a peer has a new available piece;

– `bitfield` (5) : message of length 1+N containing the list of its available pieces that a peer sends to its neighbors. This message is only sent when two peers meet for the first time. When a peer completes a new piece, it sends the lighter message `have` t oits neighbors;

– `request` (6): message of length 1+(2x4)+4 sent by a peer to request a given data block (number of the piece, number of the block, length). This message follows an `unchoke` message;

– `piece` (7): message of length 1+(2x4)+N that contains a data block (number of the piece, number of the block, data block). This message follows a `request`;

– `cancel` (8): message of length 1+(2x4)+4 sent to cancel a request. This message avoids a `piece` message when a peer receives the block from another peer.

4.2.9. *Conclusion*

The BitTorrent protocol relies on the tit-for-tat principle coupled with a priority given to rare pieces and the optimistic unchoke algorithm. The BitTorrent protocol optimizes the global state of the network while each peer only constructs a partial view of the network.

BitTorrent has a centralized point: the tracker. If the tracker fails, the BitTorrent network will disappear. Some solutions are proposed to avoid this centralized point. The simplest is the multi-tracker [HOF 11] which defines more than one tracker in the BitTorrent description file. Another way by using a DHT based on Kademlia [LOE 11]. The DHT does not change the communication protocol between peers but distributes the tracker: all peers of the swarm are connected through a DHT that lists all the peers.

4.3. Gnutella

Gnutella has remained one of the most popular peer-to-peer file sharing protocols since its creation in 2001. By the end of 2005, the Gnutella network comprized up to around $1,300,000$ nodes [STU 08], accounting for a 400% growth over the previous year. Since the appearance of the Kademlia network [KAD 11], there has been a decrease in gnutella network with users changing to Kademlia. Gnutella is a good example of a very large-scale network; it shows how such networks can be built.

The Gnutella protocol is semi-centralized, and relies on an *unstructured overlay* (see section 3.2.2). Two categories of nodes may be distinguished: ultra nodes (or ultrapeers) and leaf nodes. Contrary to classical semi-centralized network such as eDonkey [KLI 11], Gnutella nodes can be promoted to ultrapeer status or demoted to leaf status dynamically, in an effort to optimize performances on the fly as the network load evolves. Hence Gnutella is not semi-centralized in the same sense as its competitors, since it does not require a fixed infrastructure with predetermined central servers. Ultrapeers that form the core of the Gnutella network are actually promoted end-user nodes. They are interconnected on the basis of an unstructured overlay architecture, and leaf nodes become connected to the network by maintaining client/server links with ultrapeers.

There are several drawbacks to this architecture compared with other semi-centralized networks. The main one is that file lookups cannot be exhaustive, and are achieved by flooding, this is presented in section 3.2.2. Another important drawback is that Gnutella is very sensitive to *churn* (see Chapter 3): as its core is composed of end-user nodes, it is hence highly volatile. Indeed, end-user nodes may enter and leave the network at any time. The Gnutella protocol must therefore be fault tolerant, thus preventing the untimely insertion/removal of nodes from either disturbing other nodes, or partitioning the network – and possibly destroying it.

4.3.1. *History*

Gnutella was initially designed by Tom Pepper and Justin Frankel in 2000 as a side project at NullSoft, Inc. The first prototype was only deployed for a day, on March 14, 2001 [GNU 11a], before the project was abandoned by AOL, which was then owner of NullSoft, Inc. However, even in such a short span, several thousand downloads were carried out: enough to establish a stable peer-to-peer network. After a few days of reverse engineering by third parties, the first Gnutella clones started appearing, and version 0.4 – the first official version of the Gnutella protocol – was defined.

Gnutella offered an alternative to centralized file-sharing solution such as Napster, and to semi-centralized ones, such as FastTrack (KazaA). The Gnutella protocol v0.4 became popular with the disappearance of Napster for legal reasons: Gnutella does not require any fixed infrastructure integrating central servers.

Version 0.4 of the Gnutella protocol assumed a fully decentralized network based on an unstructured overlay (see Chapter 3) in which all nodes are equivalent. This original version quickly demonstrated the limits of a such an architecture in terms of scalability, as the generated traffic increased exponentially with respect to the number of nodes. As a consequence, the Gnutella protocol evolved in 2002 to version 0.6 and has not been upgraded since. This newer version reintroduces the notion of ultrapeer used by semi-centralized networks. However, instead of requiring a fixed infrastructure based on central servers, ultrapeers are elected among end-user nodes.

One of the original aspects of the Gnutella protocol is that it is defined by the developers of the client software. No new version of the protocol has been formally specified since version 0.4: version 0.6 has no official specification [1]. The latter is still in use nowadays, and it is implemented by all clients of the network.

Gnutella version 0.6 proposes GGEP (Gnutella Generic Extension Protocol) which enables arbitrary extensions through messages extension blocks. Clients are allowed to discard the data encapsulated in these blocks, and thus vertical compatibility is maintained. However, no common consensus has been reached about what these

1. An RFC draft was written by the Gnutella community for version 0.6, but no final document was produced as Gnutella2 started appearing at the same period.

blocks should be used for. Yet clever usages have been implemented by most client applications, and hence the protocol keeps evolving without requiring any formal specification.

The Gnutella2 protocol was designed by Michael Stockes in 2002. It is not a successor to the Gnutella protocol being presented here, and both protocols are completely independent; they only share one type of message, and of course their name. There is a lot of disagreement in the Gnutella community over the matter of which protocol is the best between Gnutella v0.6 and Gnutella2. Both protocols address the scalability issues of v0.4, and most client applications provide the ability to participate to both networks simultaneously. This section is dedicated to a presentation of v0.6 rather than Gnutella2, as studying the former underlines more clearly which compromises were reached to overcome the limitations of v0.4.

4.3.2. *Architecture of the Gnutella v0.4 network*

Gnutella v0.6 heavily relies on the design of v0.4 [GNU 11b]. The aim of the present section is to describe this design: it is simpler than v0.6. The overlay, routing, and lookup aspects, before delving into the improvements brought by v0.6 will be the focus of the following section.

Any node in the Gnutella network is called a *servent* – which is a contraction of server and client. A servent is sometimes designated as client; this formulation is inadequate, however, as a servent does not just connect to a server but plugs itself directly into the network. There are many servent implementations, among which some popular programs are: LimeWire [LIM 11] (Java, OS independent), Shareaza [SHA 11] (Windows) and Gtk-gnutella [GTK 11] (Unix/MacOS).

Neighborhood

Every servent is connected to the network through its neighbors. Figure 4.3 illustrates a Gnutella network, where each servent has exactly two neighbors. Notice that, although a node s may consider another node t as its neighbor, t does not have to reciprocate this relationship. In this example, D is considered as a neighbor by nodes B, C, E, and F; yet D limits its set of neighbors to F and G. A node maintains a set of neighbors called its *neighborhood*.

More formally speaking, Gnutella's unstructured overlay is a directed graph where nodes are servents and arcs represent neighborhood relationships. The following three properties must be maintained in order for the network to remain optimally efficient.

PROPERTY 4.1 *The graph must be strongly connected.*

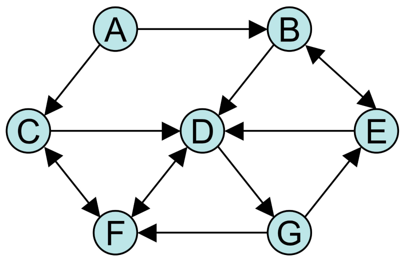


Figure 4.3. *A Gnutella neighborhood example*

Property 4.1 means that, given any two servents X and Y in the network, there must be at least one path leading from X to Y. This property is fundamental in order to avoid a partitioning of the Gnutella network into several disconnected subnetworks.

PROPERTY 4.2 *Neighborhoods must be distributed uniformly.*

Property 4.2 actually states that, if n is the average number of neighborhoods every servent belongs to, then the standard deviation around n must remain low. This property guarantees that no servent will be submitted to excessive workloads. In the example given in Figure 4.3, D is considered as a neighbor by four other nodes; this is significantly above the average value of two.

PROPERTY 4.3 *The neighborhood set of a servent must never be empty.*

A node with an empty neighborhood set will eventually be expelled from the network.

An elegant technique maximizes the probability of guaranteeing properties 4.1 and 4.2: every node selects its neighbors randomly. Random selection ensures that neighborhoods are distributed uniformly and prevents the formation of weakly connected subgraphs. In order to maintain property 4.3, a threshhold value is associated with the size of the neighborhood set, below which new neighbors have to be sought. The Gnutella v0.4 specification recommends a threshhold value of 5.

In a nutshell, Gnutella nodes strive to maintain all three properties by selecting neighbors randomly and by keeping track of at least five neighbors.

Routing and overlay

Gnutella provides query mechanisms to look up files according to multiple criteria: keywords, file size, or file type. It also enables the search for new neighbors when a node's lower threshold on its neighborhood size is reached.

The Gnutella protocol is based on an unstructured overlay and neighbors are selected randomly. Using this class of overlays, a search operation is based on a *flooding* algorithm (see Chapter 3). Hence, in order to maximize its success, a file search must be broadcast to as many nodes as possible. The higher the number of nodes queried, the greater the chance of getting a positive reply from one of them.

To initiate a search, a servent emits a request to all its neighbors. The initiator of a query is the servent that instigates a flood through the network. Receiving a request the neighbor nodes will in turn retransmit the query to their own neighbors. Flooding algorithms raise two major issues:

– a query must terminate eventually, (i.e. it cannot be retransmitted forever throughout the network). For this purpose, a *time-to-live* (TTL) counter is associated with every request message. This counter is decremented at every hop, and the request stops being forwarded when the counter value reaches zero;

– cycles, where the same nodes forward the same message among themselves, must be avoided. This is acheived by associating a unique identifier to every request. A servent will forward a request it receives for the first time, but discards it when it receives it for a second time.

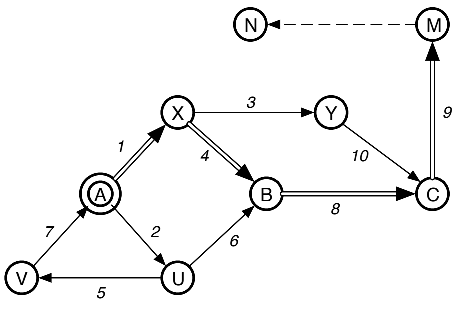


Figure 4.4. *Gnutella network routing*

Figure 4.4 illustrates how Gnutella routes requests. The numbers on the arc labels represent their order of transmission. Servent A instigates the search with a TTL value of 4. Servent N will never be aware of this particular search as it is too far from A with respect to the associated TTL. When A, B and C receive the request for the second time – through messages 7, 6, and 8, respectively – discard it in order to avoid routing through the same paths.

With Gnutella, search instigators remain anonymous as a request does not retain the address of its originating node. Ethically speaking, this mechanism for preserving the anonymity of query initiators is disputable, as it makes it absolutely impossible to ascertain which servent is looking for the given data. This approach prevents a servent from responding directly to the instigator of a request. An other drawback is that this forestalls the identification by law enforcement agents of individuals looking for illicit content, such as illegal copies of movies or outlawed pornographic data.

A naïve approach would be to respond to a request by flooding. This solution is both inefficient and impractical as neighborhood status is asymmetric. The initial TTL value of the request may then be insufficient for the reply to reach the query instigator, this is the case in Figure 4.4 where node U can be reached from A in one hop, but it requires two hops for an answer.

Gnutella provides a more efficient solution. Every time a node receives a request, it extracts the identity of the sender from the message, logs it locally, and replaces it by its own identity before forwarding the message. Thus a routing path to the query instigator is maintained as a request message is being propagated. A response is sent by passing a reply message along this path. In the example shown in bold in Figure 4.4, node M can respond to A through route {M, C, B, X, A}. Thus a request floods the network, while the routing tables get built so as to convey the response to the query instigator.

Bootstrapping and acquiring new neighbors

One of the main issues that must be addressed by Gnutella is the search for new neighbors. Indeed, servents join and leave the network continually – a phenomenon referred to as *churn* – and Gnutella avoids server nodes with predetermined addresses.

The lifecycle of a servent comprises three states:

– an *initial state*, in which the servent knows no other node in the network;

– a *startup state*, in which the servent knows the addresses of other network nodes but is not yet linked to them;

– a *connected state*, in which the servent is linked to at least one neighbor.

Every servent maintains a cache of potential neighbors, of which only a subset constitutes the neighborhood set. Elements of the latter set are nodes to which the servent is actually connected. The servent is not connected to any of the remaining nodes in the cache, and has no clue with regards to their liveness.

Figure 4.5 recapitulates the possible states of a servent. A servent program starts in the initial state if its cache is empty, in the startup state otherwise. As soon as it has achieved the connected state, a servent participates in the network and can start launching queries. Obviously, the aim of the Gnutella protocol is to prevent servents from falling out of the connected state.

Switching from the initial state to the startup state is called the *bootstrap*. The Gnutella protocol does not handle this phase; instead, most servents use a secondary protocol called GWebCache [DÄM 03]. A GWebCache server is actually an HTTP server that maintains a cache of live servents connected to the network. Once connected, a servent registers on a GWebCache server. A servent in the initial state may download the contents of a GWebCache and copy them to its own cache [2]. This approach automates the bootstrap, but requires a fixed infrastructure. To the best of our knowledge, the only possible technique for acquiring new neighbors in a dynamic infrastructure is to exchange addresses among friendly acquaintances through alternative means – IRC, mail, or even over the phone.

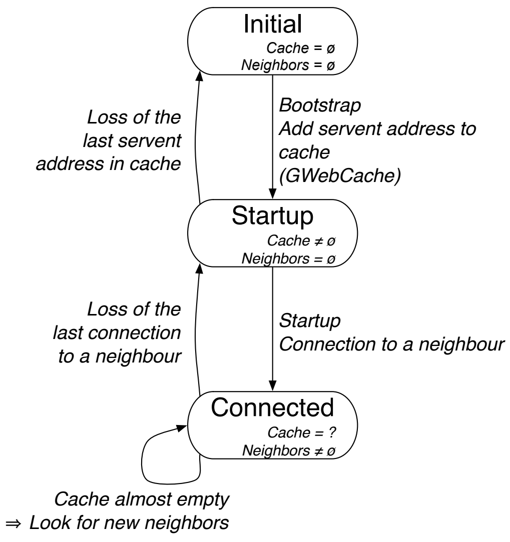


Figure 4.5. *Servent initialization*

Switching from the startup state to the connected state is called the *startup*. A servent randomly selects nodes out of its cache and tries to connect to them. Those

2. To do so, the servent has to know at least one GWebCache server address.

that have left the network and cause the connection to fail are deleted from the cache. As soon as a connection is achieved, the servent switches to its connected state.

In the connected state, a servent strives to sustain its neighborhood set so that its size remains above the threshhold value. For this purpose, it periodically connects to new servents in its cache. The size of the neighborhood set may decrease when a neighbor is assumed to be disconnected: this occurs when a timeout associated with the connection expires.

Obviously, this failure-detection scheme is very basic and not really tailored to an asynchronous network such as the Internet [FIS 85]. The suspected servent may be too overloaded to demonstrate its own liveness, yet still alive. The problem may also come from the network, too overloaded itself to convey messages on time. However, mistaken failure detections have almost no impact on a Gnutella network as all nodes are equivalent. Any node may be replaced by another, and the propagation of messages by flooding ensures that some of these can be lost at very low cost.

In order to avoid the profusion of obsolete addresses in a servent cache, the servent adds every new node address it discovers in received messages. Conversely, to avoid overstretching the cache, only the most recently used addresses are kept.

A servent that is either inactive or offline may not be able to update its cache for a long period of time. When the number of live nodes in its cache becomes too small, it launches a neighbor search request. Receiving such a request, a servent sends its own cache list as a reply. Thus, flooding the network with a neighbor lookup request enables the communication of several cache lists simultaneously. Every node along the path of such a request will benefit from it, as it also updates its own cache with the contents it forwards. This scheme enables servent caches to be updated in a fully distributed way, and preserves GWebCache servers from overload.

4.3.3. *Implementing Gnutella 0.4*

The Gnutella protocol specifies five different message types:

– *Ping*: a neighbor search request;

– *Pong*: the answer to a *Ping*;

– *Query*: a file search request;

– *QueryHit*: the answer to a *Query* sent by a servent when it knows the answer;

– *Push*: a control inversion scheme used by a sender and a receiver in order to bypass a firewall.

Ping and *Query* messages are propagated by flooding. The others are answers: a *Pong* replies to a *Ping*, a *QueryHit* replies to a *Query* and a *Push* also replies to a

QueryHit in some circumstances. Answers are conveyed along the path built during the flooding.

All messages used over the Gnutella 0.4 network display the same header, which enables the routing mechanism. The composition of the header is as follows:

– a unique request identifier, used build the routing table and to prevent cycles during a flooding;

– the TTL of a message, decremented at every hop until it reaches zero, where the message is discarded;

– the number of hops already achieved, incremented at every hop and used to set the TTL of a reply;

– the message type: *Ping*, *Pong*, *Query*, *QueryHit* or *Push*;

– the message size.

File transfer

File download/upload is a feature that is independent of the Gnutella network, in the sense that two servents wishing to exchange a file – one of the servents shares it, the other one downloads it – do so directly without involving any other node. This implies that anonymity cannot be fully enforced in Gnutella, as any servent may find out the location of nodes that share a given file.

Let l be the servent that initiates a file search by emitting an original *Query*. The search eventually terminates, at which time l acquires a list of couples (F, O), where F is a file name and O is the address of a node that shares F[3]. l can distinguish files that are shared at multiple locations: they are the files with the exact same F value. l then associates a list of servents O that share it to every F, and displays all the different values of F that were returned in response to the query to the end user. Once the end user has selected one of the F values, l attempts to download the file from the sharer nodes it knows of. The transfer itself is made by *chunks*: the file is split in equally sized parts, and every part can be downloaded from a different owner.

Every servent provides a small HTTP server with the ability to interpret a specific GET command:
```
GET /get/index/nom HTTP/1.0\n˚
Connection: Keep-Alive\n\r
Range: bytes=0..n\n\r
User-Agent: Gnutella\n\r
```

The only original aspect of the file transfer protocol is the `Range` parameter, which specifies the size of the chunks in order to allow multiple simultaneous downloads.

3. O stands for owner.

As soon as it has successfully downloaded an entire chunk of F, l itself becomes an owner of the file and may send positive replies to queries regarding F. Of course, it can only serve the chunks of F that it has already downloaded. Similarly to BitTorrent, downloads become parallelized very quickly.

Limitations of the Gnutella v0.4 protocol

In order to associate requests with their responses – a *Ping* with a *Pong*, a *Query* with a *QueryHit* and a *QueryHit* with a *Pong* – a servent must log all received messages. However, its memory capacity is limited; thus saved data must have a finite lifespan. The lifespan value depends heavily on the traffic (the number of received requests per time unit), the transit time of a request, and the available memory on the servent.

When a servent memory becomes saturated, there are two alternatives: (i) delete logged requests faster or (ii) let the memory saturate entirely. Neither of these solutions is satisfactory.

The first solution causes a significant increase in traffic as less requests get discarded for flooding when they are received more than once on the same node: this saturates the bandwidth instead of the memory. Moreover, if a node has deleted the address of the sender it received a request from, it can no longer convey the response back to the query instigator. This in turn may prompt the instigator to re-emit its query, and will lead to an even more dramatic saturation of the network.

The second approach, a full saturation of the memory, causes the servent to significantly slow down. Therefore, the forwarding delay increases for every message transiting on the node, as processing data takes more time. The direct consequence is that the neighbors of the servent whose memory is saturated will be required to log messages for longer durations, and hence risk memory saturation themselves.

This snowball effect caused by epidemic memory saturation was indeed observed on the Gnutella v0.4 network [RIT 01] when low-speed dial-up connections were still common.

In order to solve this problem, developers reintroduced the notion of super-peer already in use in semi-structured networks. Super-peers are called *UltraPeers* in the Gnutella network.

4.3.4. *Evolution to Gnutella protocol v0.6*

Version 0.6 of the protocol takes into account the aforementioned heterogenity of nodes in terms of capacity and bandwidth. In order to prevent the entire network from being handicapped by the slowest nodes, servents are separated into two categories:

– ultraPeers: stable nodes with a high bandwidth;

– leaves: all the other nodes, bound to become ultraPeer clients.

UltraPeers constitute the core of a Gnutella network and use its original communication protocol: queries are sent by flooding and results are conveyed along the request paths. Leaves, however, are simple clients of the ultraPeers. The network core thus groups the most efficient servents and is more homogeneous, hiding the limitations of the original protocol.

This architecture breaks the symmetry among servents, and becomes similar to the eDonkey network, where the core is a peer-to-peer overlay and clients are connected to the core through a client/server connection. The main difference with eDonkey resides in the dynamicity of the core structure. eDonkey favours a static overlay where ultraPeers are clearly identified, whereas ultraPeer status is dynamic in Gnutella. Gnutella leaves can be promoted to ultraPeer status if they satisfy several eligibility criteria, and ultraPeers get demoted to leaf status when they stop satisfying them.

Both the Gnutella v0.6 architecture and protocol are very similar to that of v0.4. The following sections describe how some mechanisms may differ from the original version, and detail elements that were not mentioned in previous sections.

Network architecture and protocol mechanisms

Theoretically, ultraPeers have greater bandwidth then the average servent in v0.4, hence they can handle a higher number of neighbors. The typical utlraPeer is supposed to keep up permanent connections with 30 other ultraPeers, and to service between 30 and 45 leaves. A leaf aims to stay connected to three ultraPeers to ensure that it will not leave the network.

In essence the Gnutella v0.6 protocol works in a similar way to v0.4: the message structure remains the same. The fundamental difference is that only ultraPeers use the flooding algorithm. They keep track of all the files shared by their leaves and send responses in their stead. Figure 4.6 shows the scenario that unfolds when leaf L1 initiates a search for file xyz in a v0.6 network. L1 sends a *Query* message to its ultraPeers – U2 in this example. U2 floods the network core with this message and waits for the results. Upon receiving this request, U1 acts as a proxy for L2: it sends a *QueryHit* message back along the path to U2, which in turn forwards it to L1.

Ping and *Pong* messages in the network core are completely isolated from their matching counterparts in the periphery. In the example of Figure 4.6, when L3 sends a *Ping* message to its ultraPeer, U3 does not flood the core with this request and sends its own cache back directly. However, if the cache of an ultraPeer becomes too small, the node does flood the network core with its own *Ping* message and acts exactly as described in Gnutella v0.4.

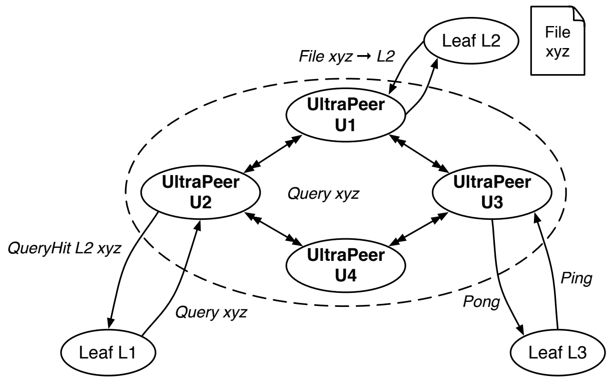


Figure 4.6. *File search example in Gnutella v0.6*

Although it is privileged in terms of access to the data, an ultraPeer remains a servent node with a user behind it. Therefore, an ultraPeer may initiate its own searches, in which case it behaves as if it were a leaf.

This architecture solves the limitations of the v0.4 network by avoiding low bandwidth servents along the message paths. It also prevents the saturation of leaf node resources that are usually more scarce.

Version 0.6 of the Gnutella protocol also offers several additional improvements, such as the ability to cancel a request once the initiator is satisfied with the results. This technique is particularly useful for files that are extremely popular, and get shared by a huge number of nodes.

UltraPeer election

To become an ultraPeer, a servent must satisfy the following properties:

– it must not live behind a firewall;

– it must execute an operating system able to open a large number of socket connections (Linux, Windows 2000/NT and greater, Max OS 10 and greater);

– it must have a good network rate (10 kb/s for download and 15 kb/s for upload);

– it must be stable, i.e. the servent must be connected for some hours;

– it must be powerful (good CPU, large memory).

When a servent has all these properties, it is "able to become an ultraPeer". The servent self-evaluates its ability to become an ultraPeer and nothing prevents a malicious servent from becoming an ultraPeer even if it does not satisfy all these properties.

The election of an ultraPeer implements a simple protocol which aims to stabilize their number. Technically, when a peer A meets a peer B and wants to add B to its neighborhood, different possibilities must be considered:

– **Case 1.** A and B are leaves:

- **1.a.** if B only implements Gnutella 0.4, it accepts the connection of B. B then uses A to find other peers;

- **1.b.** if B implements Gnutella 0.6. If B knows ultraPeers, it declines the connection and send the address of an ultraPeer to A. Otherwise, B asks A to become an ultraPeer. If A is able to become an ultraPeer, it becomes an ultraPeer, otherwise, B becomes a neighbor of A as in the protocol 0.4;

– **Case 2.** A is a leaf and B is an ultraPeer: B accepts the connection of A. However, if B has too many connections, it asks A to become an ultraPeer. If A is able to become an ultraPeer, A reconnects B as an ultraPeer (see case 4). In all cases, if A had neighbors as a leaf (see case 1), A sends its neighbors to B and they try to connect B;

– **Case 3.** A is an ultraPeer and B is a leaf: this case is impossible. If A is an ultraPeer it will never choose a leaf as a neighbor;

– **Case 4.** A and B are ultraPeers:

- **4.b.** if A estimates that it does not have enough leaves, it will ask for B some of its leaves. If B has no more leaves, it becomes a leaf of A.

- **4.b.** otherwise, A and B become ultraPeer neighbors.

To stabilize the number of ultraPeers, two mechanisms are used. A leaf becomes an ultraPeer when it connects an overloaded ultraPeer (case 2) or when it has neighbors (case 1) and becomes able to be an ultraPeer. An ultraPeer becomes again a leaf when one of its neighbors takes all its leaves (case 4). This simple algorithm balances the number of ultraPeers in the network without requiring a global knowledge: the balance is only local to a peer and its neighborhood, but, little by little, a balance between leaves and ultraPeers is respected all over the network.

To summarize, the Gnutella protocol is distributed and does not require a fixed infrastructure: Gnutella does not require clearly identified peers that never crash. The original version of the protocol shows how to build an entirely unstructured overlay where all peers are strictly equivalents. However, the deployment of this first version showed degradations of performances around weak nodes.

The solution brought by the version 0.6 enhances the algorithms by defining ultraPeers and leaves. The backbone of the Gnutella 0.6 network is defined by gossip with the ultraPeers. Each ultraPeer acts as a server for a small set of clients.

The main originality of this protocol is the election of ultraPeers: the overlay does not require a fixed infrastructure and only powerful and stable leaves can become ultraPeers. The algorithm stabilizes the number of ultraPeers without constructing a global view of the network. When an ultraPeer has too many leaves, one of the leaves is promoted to become an ultraPeer and reciprocally. If two ultraPeers with few leaves meet, one of the ultraPeers becomes a leaf. Experimentally, this simple algorithm using a rough estimation globally stabilizes the number of ultraPeers.

4.4. Conclusion

Our study of BitTorrent and Gnutella shows how it is possible to build scalable distributed networks without relying on global knowledge. Each peer only has a partial view of the network and takes its decisions statistically. The algorithm that promotes rare pieces in BitTorrent or the algorithm that elects an ultraPeer in Gnutella only requires an estimation given by the neighborhood of a peer. Globally, these networks stabilize some properties: pieces of files are uniformly distributed in BitTorrent and the number of super-peers remains proportional to the number of leaves.

Studying and developing algorithms that scale with the size of the network remains a complex subject because, most of the time, only experimental observations can uncover the weaknesses or the strengths of these algorithms. For example, only the observation of the Gnutella 0.4 network can show that the presence of some weak peers can drastically degrade the performance ok a large portion of the network.

Both the applications studied in this chapter can intrinsically scale with the size of the network because the manipulated data are only read. If a peer wishes to modify a file it simply creates a new file that is in no way linked to the original. Building a scalable network that supports writable data also remains an open subject because different versions of the same file will exist in the network and these replicas must eventually converge to the same value.

4.5. Bibliography

[BIT 11] BITTORRENT, "Bittorrent protocol specification v1.0", http://wiki.theory.org /BitTorrentSpecification, 2011.

[COH 02] COHEN B., "BitTorrent - hosting large, popular files cheaply", *CodeCon*, 2002.

[COH 11] COHEN B., "The BitTorrent protocol specification", http://www.bittorrent.or g/beps/bep_0003.html, 2011.

[DÄM 03] DÄMPFLING H., "Gnutella web caching system - version 2 specifications client developers' guide", http://www.gnucleus.com/gwebcache/newgwc.html, 2003.

[FIS 85] FISCHER M. J., LYNCH N. A., PATERSON M. S., "Impossibility of distributed consensus with one faulty process", *J. ACM*, vol. 32, p. 374–382, ACM, 1985.

[GNU 11a] GNUTELLA, "Gnutella announcement on SlashDot", http://slashdot.org/art icle.pl?sid=00/03/14/0949234, 2011.

[GNU 11b] GNUTELLA, "The Gnutella protocol specification v0.4 – document revision 1.2", www.stanford.edu/class/cs244b/gnutella_protocol_0.4.pdf, 2011.

[GTK 11] GTK-GNUTELLA, "Gtk-Gnutella home page", http://gtk-gnutella.sourceforge.net, 2011.

[HOF 11] HOFFMAN J., "Multitracker metadata extension", http://www.bittorrent.org/beps/bep_0012.html, 2011.

[KAD 11] KADEMLIA, "Kademlia: a design specification", http://xlattice.sourceforge .net/components/protocol/kademlia/specs.html, 2011.

[KLI 11] KLIMKIN A., "eDonkey protocol specification 0.6.2", http://sourceforge.ne t/projects/pdonkey/, 2011.

[LEG 06] LEGOUT A., URVOY-KELLER G., MICHIARDI P., "Rarest first and choke algorithms are enough", *Internet Measurement Conference*, p. 203-216, 2006.

[LIM 11] LIMEWIRE, "Lime Wire home page", http://www.limewire.com, 2011.

[LOC 06] LOCHER T., MOOR P., SCHMID S., WATTENHOFER R., "Free riding in BitTorrent is cheap", *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[LOE 11] LOEWENSTERN A., "DHT protocol", http://www.bittorrent.org/beps/bep_0005.html, 2011.

[RIT 01] RITTER J., "Why Gnutella can't scale. No, really.", http://www.eecs.ha rvard.edu/jonathan/papers/2001/ritter01gnutella-cant-scale.pdf, 2001.

[SCH 09] SCHULZE H., MOCHALSKI K., "Internet study 2008/2009", http://www.i poque.com/userfiles/file/ipoque-Internet-Study-08-09.pdf, 2009.

[SHA 11] SHAREAZA, "Shareaza home page", http://www.shareaza.com, 2011.

[STU 08] STUTZBACH D., REJAIE R., SEN S., "Characterizing unstructured overlay topologies in modern P2P file-sharing systems", *IEEE/ACM Transactions on Networking*, vol. 16, p. 267-280, 2008.

Chapter 5

Large-Scale Peer-to-Peer Game Applications

5.1. Introduction

Massively multiplayer online games (MMOG) recently emerged as a popular class of applications with up to millions of users, spread over the world, connected through the Internet to play together. Most of these games provide a *virtual environment* in which players evolve, and interact with each other. When a player moves, moves an object, or performs any operation that has an impact on the virtual environment, players around him can see his actions.

In a MMOG, each player runs a piece of client software on a local device (e.g. a personal computer or a game station) called *node* thereafter. The local client software is responsible for maintaining an up-to-date version of the state of the virtual world surrounding the player and to offer him the ability to perform operations on it (i.e. to play). To offer an acceptable gaming experience, the distributed application needs to render the virtual world surrounding the player with minimal latency.

5.1.1. *Limitations of the client-server paradigm*

Current most popular MMOG such as World of Warcraft [ENT 11] or Second Life [LAB 11] are based on the client-server paradigm. Using this model, a server is responsible for keeping the state of the virtual world up to date. Each time a player plays (performs an action), he has to notify the server in order for his actions to be taken into account. The server then computes the player's neighborhood, composed of

Chapter written by Sébastien MONNET and Gaël THOMAS.

Distributed Systems: Design and Algorithms Edited by Serge Haddad, Fabrice Kordon, Laurent Pautet and Laure Petrucci
© 2011 ISTE Ltd. Published 2011 by ISTE Ltd.

all the players that are potentially impacted by the performed action. It then notifies all the impacted nodes about the changes in their virtual environment. The nodes locally render the modification to the human player. Within a game, users are continuously moving, and therefore their neighborhoods are continuously changing. Players also keep modifying their virtual environment while playing. The server thus continuously receives a large amount of notifications. It has to compute many neighborhoods and must send even more notifications.

This client-server model is unfortunately not scalable for this kind of application [KUM 08]. In fact, using the currently available client-server-based MMOG implementations, the millions of registered users are not really playing at the same game instance at the same time: these MMOG introduce new playing rules in order to limit the number of participants within a single game instance. The game is partitioned into *sub-games*. For instance, the Second Life world is not contiguous: it is composed by multiple *islands*; World of Warcraft is split into many *realms* [PIT 07], etc. At a given time, they are no more than a few hundreds players interacting in a same sub-game connected to the same server.

Furthermore, the client-server paradigm implies an expensive financial cost for the provider [ALV 07]. Servers need to provide high computing performances. They are usually replicated to support failures or disconnections for maintenance. Generally, a game instance (e.g. a Second Life island) is served by a cluster of high-performance nodes rather than a single server. These clusters have to be sufficiently powerful to support usage peaks (e.g. when an important event occurs within an game, like a concert for instance). The cost for building and for maintaining these clusters is high.

5.1.2. *The decentralized model*

To circumvent these limitations, a new generation of decentralized *networked virtual environments* (NVE) based on peer-to-peer overlays has emerged [KEL 03, BEA 07a, BEA 07b, HU 06, FRE 08, LEG 10, VAR 09b, VAR 09a, IIM 04, BHA 06, BHA 08, BHA 04] (most of these works are discussed later in the chapter). In addition to scalability and cost, these new solutions offer better availability as it is possible to keep playing even when many players join and leave the game. In fact, several players can play together as soon as they are connected. These new systems also offer more freedom to the player: it becomes possible to have a game without any central organization/company controlling it.

However, distributing such applications using the peer-to-peer paradigm is really challenging. Most existing peer-to-peer overlays do not fulfill these emergent application requirements. Their communication latencies are way too high, they are not able to offer the player an acceptable gaming experience.

Building large-scale peer-to-peer game applications requires new kinds of overlays, tailored for this class of application: as they are particularly dynamic, they require that the underlying overlay adapts itself to keep communications latencies low, while communication patterns evolve.

Section 5.2 describes the requirements of this particular class of application. Then section 5.3 explains in more details why classical overlays are not suitable for large-scale game distributed applications and presents a new class of peer-to-peer network overlays that fulfill these applications requirements. Then, several state-of-the-art solutions are presented for the two main classes of multi-player games: first person shooters (FPS) [BHA 04, BHA 08, BHA 06] in section 5.4 and life simulations [KEL 03, FRE 08] in section 5.5.

5.2. Large-scale game applications: model and specific requirements

This section defines the application model. It provides several notions, and details the requirements for large-scale game applications on the underlying distributed system. Finally it presents the main characteristics of such applications that could be used while designing distributed systems for them.

5.2.1. *Application model*

The emergent MMOGs systems use different terminologies. We present here the most important notions and the terminology used in the remainder of this chapter.

Virtual environment

MMOGs applications are based on the concept of a *virtual environment*. A virtual environment is an n-dimensional applicative naming space in which players evolve. Usually the dimensions are mapped on the coordinates of a virtual world (e.g. the x, y and possibly z axis of a three-dimensional map), but it is also possible to use other notions of semantic distance (e.g. players' contact lists, hidden gates that allow the player to jump somewhere else in the virtual world, etc.).

Avatar

Each player node manages an entity representing the player in the game. This entity is a particular object of the virtual environment, usually called an *avatar*. Each object, and thus each avatar, is assigned coordinates in the virtual world. When a player plays, the player's avatar moves (and potentially moves objects) and its coordinates in the virtual environment change.

Object manager

Regarding scalability concerns, it is not possible for a node to store and maintain up-to-date information retaining to the whole virtual environment: a virtual world could contain up to hundreds of thousands of players at a given time (even millions). Furthermore, it is continuously changing, therefore, each node would have to continuously receive information from all other nodes, which is clearly not scalable. This is exactly what happens using the client-server paradigm. Thus, in a *distributed* large-scale game application, the virtual world is distributed over the nodes.

Each object is therefore managed by a set of nodes called *the managers of the object*. For robustness or scalability, most of the time, each object is replicated on more than one manager. When a node wants to read the state of an object, it must therefore find and then contact one, some, or all the managers of the object, depending on the design choices. Furthermore, if the node wants to modify the state of the object, it delegates this modification to the manager, which agree on the new state of the object.

Knowledge area

Each node therefore manages a set of objects. We define this set as the *knowledge area* of the node, also called "responsibility zone". Each time something changes in its knowledge area, a node has to update it, either by being notified of changes by other players or by periodically probing nodes that the avatar plays in its knowledge area, depending on design choices.

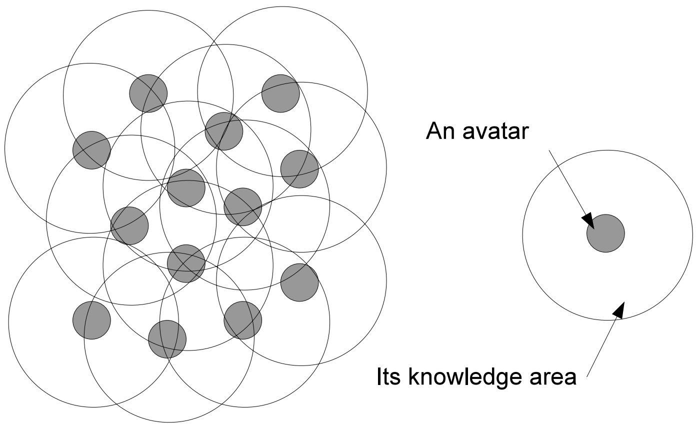


Figure 5.1. *Knowledge areas intersect: the virtual world is replicated*

As objects are replicated among their object managers, the intersection of the knowledge areas of two nodes contains the objects that the two nodes manage: knowledge areas are shared between nodes. With simple replication protocols, the replication granularity is the knowledge area. With more advanced replication protocols, the replication granularity is the object. Figure 5.1 illustrates an advanced protocol where the knowledge area of a node depends on its avatar position [FRE 08]. The figure presents avatars and their knowledge areas. In this example, a portion of the virtual environment is often part of two or more knowledge areas: it is replicated on the nodes running the corresponding avatars. In simple and advanced replication protocols, the virtual environment is distributed (and replicated) among all the participating nodes.

As no node hosts a complete copy of the virtual environment, they have to collaborate in order to give to the players the illusion of a unique continuous virtual world. The distributed MMOG has to ensure that every portion of the virtual world is hosted by at least one node in the system. This means that the whole virtual world has to remain the union of the knowledge areas.

Playing area

The client software (running on player nodes) needs to maintain an up-to-date *playing area* for the player. The playing area is the zone surrounding the avatar that needs to be rendered/displayed to the human player. Figure 5.2 illustrates this notion. When a player plays, its avatar modifies the local playing area. Changes are propagated to nodes that manage objects in this playing area: these are the nodes whose knowledge area contains parts of the modified playing area.

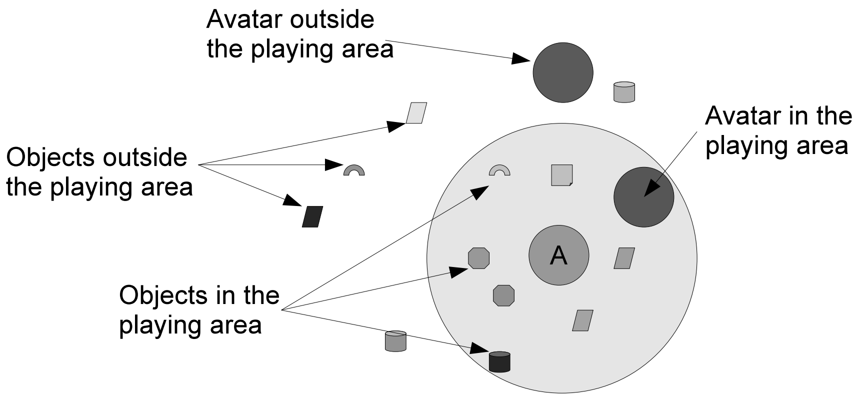

Avatar outside the playing area
Avatar in the playing area
Objects outside the playing area
Objects in the playing area
A

Figure 5.2. *Avatar A's playing area*

Elders of a playing area

To simplify the presentation, we introduce also the notion of elders [LEG 10]. The elders of a playing area are the nodes that manage the objects in the area. In other words, the elders of a node N are the nodes E whose knowledge areas intersect the playing area of N. The elders of a playing area are the servers of the objects located inside the playing area. We also define the elders of a node as the elders of the playing area.

Playing area/knowledge area relations

There are two main design choices.

First, on a node, the playing area can be included in the knowledge area. In this case, the knowledge area of a node is around its avatar and the limits of this area change as the avatar moves. With this design, elders of a node are close to the node's avatar inside the virtual environment. To ensure that the whole virtual environment is replicated even if a player is alone in its playing area, knowledge areas are bigger than playing areas. This is the model chosen by the Solipsis system [KEL 03] detailed in section 5.5.

Conversely, the world can also be statically partitioned in responsibility zones (i.e. knowledge areas), which are then attributed to/replicated on several peers. Therefore, in this model a peer is a server for a part of the virtual environment, its knowledge area; but it acts as a client: its avatar may evolve in a totally different zone and its playing area may be far from its knowledge area in the virtual environment. In this case the knowledge area boundaries are static and fixed while the playing areas are changing with avatars movements. This is usually the case in systems based on peer-to-peer distributed hash tables (DHTs) [VAR 09b, VAR 09a, IIM 04] (see section 3.2.2).

Summary of the terminology

The terminology used in large-scale game applications is summarized below:

– *virtual environment*: the n-dimensional applicative naming space;

– *avatar*: a special object that represents a player in the virtual environment;

– *object managers*: the set of nodes that manages an object;

– *knowledge area*: the set of objects managed by a node;

– *playing area*: the set of objects the player is interested in;

– *elders* of a playing area: the object managers of the objects located in the playing area.

5.2.2. *Main requirements for large-scale game applications*

There are two main requirements in large-scale game applications: playability and consistency.

Playability

To be used, a game has to provide the user with a comfortable gaming experience. The player should have the illusion that there is no latency at all between the moment another player moves and the moment the player sees it. At the distributed infrastructure level, this implies that latency should remain very low (under certain limits). Luckily, many graphical tricks exist at the application level to hide the inevitable network latency [PAN 02, PAN 07].

The admitted latency offered by the peer-to-peer system largely depends on the kind of game application. For instance, a life-simulation game, or a virtual visit of a town/museum may tolerate greater latencies than FPS games. In the first case, high latencies may induce slow movements, slightly degrading the playing experience, while in the second case, a player may still be playing while its avatar is already dead, making the game totally unplayable.

Therefore, the goal of the emergent peer-to-peer overlays for large-scale distributed game applications is to keep communication delays as low as possible. This is challenging because while players play, their avatars move, their playing areas change, which in turn changes the communication patterns among nodes.

Consistency

The whole game, i.e. the whole virtual world, has to remain available. As it is dynamic and spread all over the participating nodes, replication mechanisms have to be implemented in order to ensure that a copy of each portion of the virtual world is available at any time. Furthermore, copies of the same portions of the virtual world have to be mutually consistent. Therefore, nodes have to somehow synchronize their knowledge areas.

5.2.3. *Influence of the avatar mobility on the playability*

Within large-scale game applications, a player shows and acts on a limited part of the virtual environment: the objects of the playing area. Therefore, a node participating in a distributed large-scale game application is interacting with a limited number of other nodes at a given time: its elders.

To increase the playability, it is thus possible to efficiently predict the communication patterns using the avatars coordinates in the virtual environment: avatar coordinates define their playing areas which directly defines their elders. However, during the game, the mobility of the avatar induces continuous changes in the coordinates and thus of the elders and of the communication patterns.

To increase the playability, it is therefore important to characterize how avatars are moving.

Characterization of avatar mobility

The characteristics of avatar mobility may be learned from real traces. However, all existing commercial MMOG projects are based on a client-server architecture which, as stated above, usually implies poor scalability. Therefore, each trace of a real game session involves at most a few hundred avatars simultaneously. Moreover, the number of available real traces is small because they are difficult to obtain [LA 08]. Therefore, there are few real traces.

Recently, some research effort has been applied to characterizing avatar distribution and avatar mobility in large-scale game applications [LEG 10, RHE 08, LIA 08, LA 08, MIL 09]. Avatars in virtual environments usually have total freedom of movement. Resulting virtual environments are then very dynamic: data representing objects and avatars may not be uniformly distributed all over the universe. Recent studies of existing popular large-scale games, such as Second Life [LAB 11] and World of Warcraft [ENT 11], have shown that the distribution of avatars was extremely disparate [LA 08, PIT 07]: most of the avatars were gathered around a few hot-spots of interest, while large parts of the virtual environment were almost deserted. Figure 5.3 shows the distribution of avatars on a Second Life island.

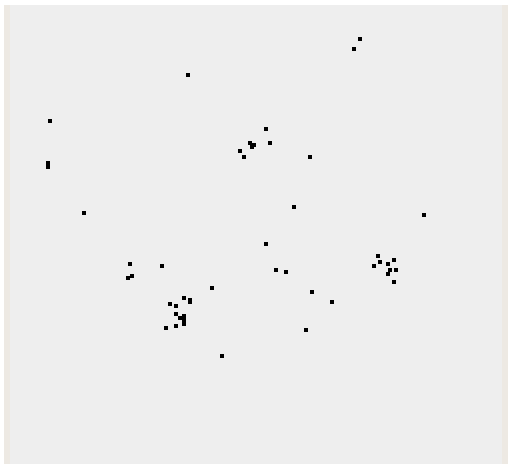


Figure 5.3. *A sample avatar distribution in Second Life*

This kind of distribution with hot-spots also corresponds to real density distributions of human populations, such as the European *blue banana* [BRU 02] that covers one of the world's highest concentrations of populations around the cities of London, Brussels, Amsterdam, Cologne, Frankfurt, and Milan with approximately 20% of the European population.

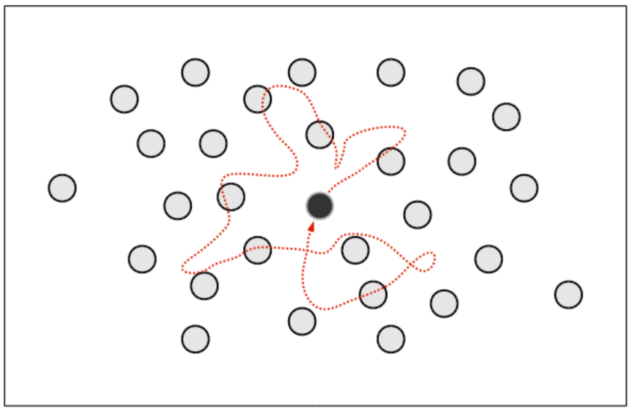


Figure 5.4. *Avatar movement within high density zones*

Moreover, regarding player mobility in virtual environments, studies have shown that it is quite similar to human mobility in the real world [RHE 08, LA 08]. The mobility patterns of avatars has been shown to be highly non-uniform: avatars move slowly and chaotically within the hot-spots as illustrated in Figure 5.4, whereas the movement between the hot-spots is straight and fast [LIA 08]. This mobility pattern is usually modeled with Lévy flights [RHE 08, LA 08]. Indeed, Lévy flights are a particular sort of random walk in which the increments are distributed according to a "heavy-tailed" probability distribution [CHE 06] with short and chaotic movements, and sometimes, long and straight movements. Therefore, Lévy flights naturally differentiate the two observed behaviors of avatars: periods of travel and periods of exploration with chaotic movements. However, Lévy flights do no help to model hot-spots because they do not ensure that avatars stay grouped around hot-spots and that density around hot-spots remains the same whatever avatar mobility is. Recent research effort has focused on designing models able to generate realistic avatar movements on a large scale in order to be able to test really large-scale game applications prototypes [LEG 10].

Influence of game rules on avatar mobility

Finally, the game rules can also influence avatar movement. For instance in war games, or quest games, avatars may be enrolled in fellowships. All the avatars belonging to the same fellowship (or a same army) may move *together* in the same direction with similar speeds, generating *group movement* [MIL 09], as illustrated in Figure 5.5. An avatar within a moving group changes its playing area, but the objects of its playing area remain almost the same because avatars located in its playing area move along with him.

This kind of group movement is particularly efficient for distributed MMOGs where the playing area is included in the knowledge area. In this case, the elders of a player within the group also remain around the player. Only avatars located on the front line of the group have to obtain new elders ahead of the movement.

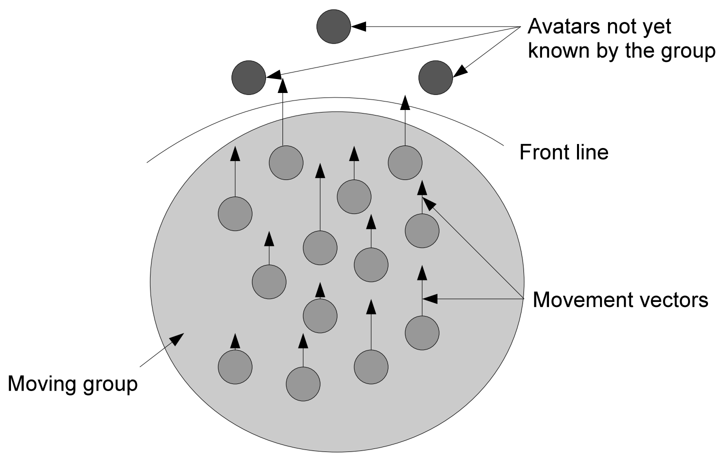


Figure 5.5. *Avatar group movements*

These characteristics can be considered while designing distributed systems for large-scale game applications. By taking them into account, it becomes possible to offer relatively low latency communications between peers that need to interact. The next section presents peer-to-peer overlay networks tailored for large-scale game applications based on these characteristics.

5.3. Overview of peer-to-peer overlays for large-scale game applications

As stated in the previous section, a client of a large-scale MMOG must only have a partial knowledge of the whole virtual environment: its playing area. The peer-to-peer model suits this kind of application well where each node only has a local view, and where the union of the local views offers a global view.

However, even if the peer-to-peer model seems well adapted, all peer-to-peer overlays are not suitable for MMOGs because they must ensure the playability of the game: communications between a node and its elders should be efficient. The underlying peer-to-peer network overlay must, therefore, limit the number of hops between a node and its elders.

In this section we present an overview of the different overlays, from the most *inflexible* to the most *malleable* and highlight their qualities and faults for playability.

5.3.1. *Unadaptable/inflexible overlays*

Considerable research effort has been conducted in the past decade in the field of peer-to-peer overlay networks. As presented in the previous chapter, these overlays have mainly been designed for *one* specific target application: large-scale read-only file-sharing. These overlays are supposed to build a graph that connects all the nodes together and permits efficient random search operations. Nodes choose the function of their neighbor's hash number if the overlay is based on a DHT or randomly if the overlay is unstructured [ORA 01, STO 01]. Except in case of failure, the neighbors of a node do not evolve over time because read-only file-sharing applications do not have this need: searches are supposed random and a fixed random overlay is sufficient.

Building a distributed virtual environment over of a random inflexible overlay is a hard task. First, a node and its elders have no reason to be close in the overlay while they communicate. Moreover, as presented in the previous section, the elders of a node evolve with the movement of the node's avatar. In these inflexible overlays, the neighborhood of a node does not change and a node can, therefore, not choose the closest neighbors to its elders in terms of number of hops.

Varvello *et al.* [VAR 09b] implemented a virtual environment over a DHT. The virtual world is partitioned into knowledge areas that are replicated among the DHT. The authors show that the responsiveness of the DHT is acceptable with minimal avatar mobility but not if mobility increases. In this case, implementing a reverse binary tree on top of a DHT could help to lower the latency [VAR 09a].

Colyseus [BHA 06], a decentralized architecture to support MMOGs with tight latency constraints (typically FPS games), which is detailed in section 5.4, is also based on a DHT for virtual object discovery. At the storage level, Colyseus prefetches objects.

Donnybrook [BHA 08], the sequel of Colyseus, also implements advanced dead reckoning techniques. These techniques predict the position of an avatar based upon its previously determined location. Dead reckoning techniques avoid a lot of messages to update avatar locations and, therefore, decrease the network load. Donnybrook is described in more detail in section 5.4.

5.3.2. *Overlays reacting to application needs*

Recent works have focused on dynamically adapting the overlay to better satisfy application needs. For instance, semantic overlays [VOU 04] build links between semantically close peers. These allow semantically close peers to be close in the overlay (in terms of number of hops), which is a good because they are likely to interact to exchange data. Few recent overlays adapt their structure function to the application needs [MON 06, VOU 06, HU 06]: they react to the evolution of the application, generally by detecting communication between nodes.

Recently, some research effort influenced by these works has focused on building overlays tailored for distributed virtual environments. In such systems, the logical neighborhood of a node in the overlay is determined by the playing area of its avatar in the virtual environment: the overlay tries to keep the elders of the node in the neighborhood of the node. As an avatar moves in the virtual environment, the playing area of its node evolves and the overlay reacts by choosing the new relevant elders as neighbors. Figure 5.6 illustrates how a peer-to-peer overlay adapts itself when an avatar moves in the virtual world. The top of the figure presents the virtual environment with the avatars, and the bottom, the overlays with the nodes. Each node is connected to the avatars located in its playing area. When node A moves, its playing area changes and it therefore chooses new neighbors in the overlay.

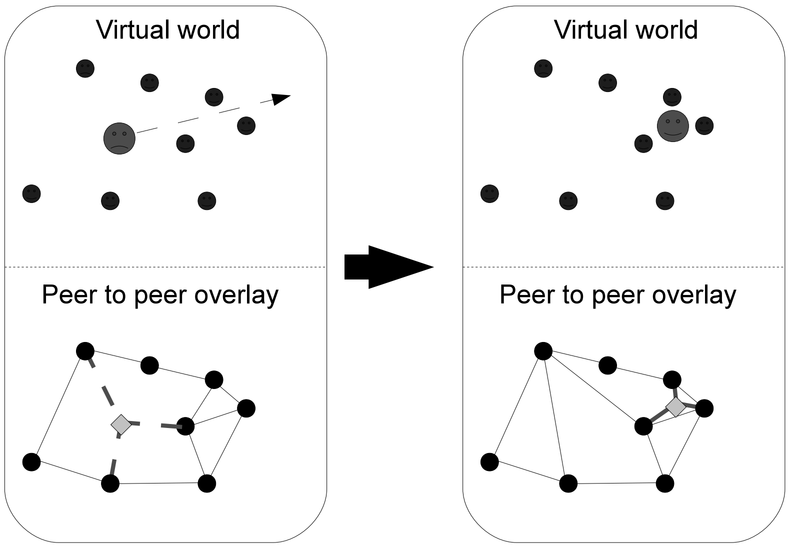


Figure 5.6. *Peer-to-peer overlay network malleability*

Recently, several of these overlays have been designed. Two main families have been proposed. The first is based on an unstructured overlay. This is the case for Solipsis [KEL 03], which is detailed in section 5.5.

The second is based on a structured overlay based on the virtual environment. They use a Voronoi tessellations-based overlay. The n-dimensional applicative naming space is divided into tiles centred around each avatar. Figure 5.7 presents a Voronoi tiling in a two-dimensional space where a point represents an avatar. The sizes of the tiles are dynamically computed function of the object density: the number of objects in each tile is approximately the same. VoroNet/RayNet or VON [BEA 07a, BEA 07b, HU 06, HU 04] are MMOG overlays based on Voronoi. A tile is the knowledge area of a node. The main strength of these overlays is the efficiency of search queries of objects around a given coordinate: the query is forwarded along the shortest possible path. However, maintaining this structure is costly because avatars are always moving.

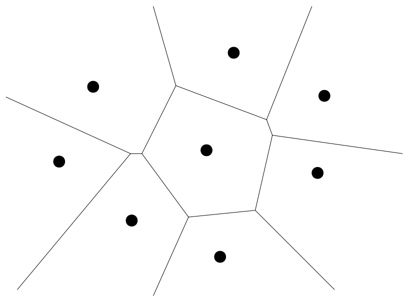


Figure 5.7. *Voronoi tessellations arround players' avatars*

5.3.3. *Anticipating avatar movement to pro-actively adapt the overlays*

Previous overlays react to avatar movements but they do not try to anticipate these movements to pro-actively adapt the overlay.

An avatar movement anticipation module called Blue Banana has been proposed in [LEG 10]. This anticipation mechanism can be implemented on top of the distributed MMOGs presented in the last subsection. Blue Banana detects the movement of an avatar, and tries to prefetch neighbors located ahead of the anticipated movement. Blue Banana is detailed in section 5.5

5.4. Overlays for FPS games

FPS, such as Quake, require very low latency: in FPS games, the fact of winning or loosing is often based on player reactivity. If a player dies before the avatar that shot him enters in its playing area, the game becomes unplayable.

However, this kind of game rarely involves a few thousand participants. The scale is usually tens or hundreds than rather thousands and a DHT is sufficient.

This presents two main research efforts in this direction: Colyseus [BHA 06] and Donnybrook [BHA 08]. These systems are based on DHTs that support range queries, we first introduce Mercury [BHA 04] which is used by Colyseus.

5.4.1. *Malleable overlay for range queries*

Classical peer-to-peer DHT, as presented in section 3.2.2, allow efficient put/get operations. However, the key-based storage is a bit simplistic: it does not provide either range queries, or multi-attributes queries.

Mercury [BHA 04] is a peer-to-peer system allowing both range and multi-attributes queries. To support multi-attributes queries, Mercury organizes peers in multiple logical DHTs: one per attribute. Then, to handle range queries easily within each logical ring, Mercury arbitrary assign ranges to peers. Therefore, each peer becomes responsible for a sub-portion of the semantic space.

Within a virtual environment, objects are not likely to be distributed uniformly in the space. Thus, a peer may quickly become overloaded if the portion it is responsible for contains too many objects. Therefore, Mercury proposes a smart load balancing mechanism: each peer periodically probes the global naming space to identify high-density zones. Underloaded peers change their hash number to join high-density zones. Peer distribution thus converges to the objects distributed in the virtual environment. This overlay is *malleable*: it adapts automatically to the object distribution over the DHT.

However, when the distribution changes (e.g. when objects move) the load may change quickly. This design is not tailored for high variations.

5.4.2. *Colyseus*

Colyseus [BHA 06] is built on top of the Mercury DHT. It is a peer-to-peer system tailored for FPS games. It is based on a publish/subscribe mechanism: each peer publishes its avatar state in the DHT, and each peer subscribes to the area they are interested in (their playing area). Each time an avatar moves, the avatars in its playing area are notified because they have subscribed to the zone in which it moves.

However, as lookups in DHTs may be too slow, Colyseus offers (i) a caching mechanism and (ii) a prefetch mechanism. Therefore, objects may be discovered quickly: based on locality and predictability in data access patterns, Colyseus speculatively prefetches objects. Furthermore, this mechanism is only used for discovery. Once discovered, objects are cached and direct links are drawn between both the primary copies (in the DHT) and the caches.

Colyseus has been tested with Quake II[1] with hundreds of players.

1. http://www.idsoftware.com/business/index.php

5.4.3. *Donnybrook*

Donnybrook [BHA 08] has been designed based on Colyseus. It is also a peer-to-peer system tailored for FPS games.

It is based on the fact that the human brain cannot pay attention to too many objects simultaneously. Each peer, at each period, selects only five objects in its playing area. Then these objects are kept up-to-date with high frequency in order to ensure a good freshness quality until the next period. During the next period, a new subset will be elected.

The selection of these five objects is based on three criteria that allows the computation of the objects requiring high-fidelity rendering:

– physical proximity: a close object is more likely to attract attention than a distant one;

– visibility: an object in the center of the screen attracts the player's attention more;

– temporal proximity: an object having already attracted the player attention will probably also attract his attention during the next period.

Other objects located in the player's playing area are also kept up-to-date but with a lazy mechanism: they are synchronized every second with the primary copy in the Donnybrook system. Such a period may lead to inconsistencies: objects move continuously, they do not make "jumps" each second. To deal with these inconsistencies, Donnybrook uses *dead reckoning* techniques [PAN 02]: the player's nodes simulate behavior of each object between two synchronizations. Therefore, the behavior of these objects is approximate, but it is not critical: they are only secondary objects which are not in the center of the player's attention.

These approximations enable the network to have a reduced load and to enhance the whole scalability of the system. They allow Donnybrook to support hundreds, and even thousand of simultaneous players in Quake II.

To summarize, DHT can be efficient to build overlays for MMOGs if (i) the number of player remains limited, (ii) each object in a playing area of a node is cached, (iii) only important objects are accurately updated, and (iv) the movement of other objects are predicted.

5.5. Overlays for online life-simulation games

Life-simulation games are characterized by (i) their potentially very large scalability; (ii) the lower importance of offering small latencies. Indeed, hundreds of thousands of players may play simultaneously, being spread all over the world. To offer a pleasant gaming experience, latencies should remain low, but this is not as critical as it is for FPS games.

Peer-to-peer systems for life simulation can be classified in two main classes:

1) overlays where the knowledge area of a node does not depend on its coordinate. Knowledge areas are usually a statically distributed function of their hash key on a DTH [IIM 04, VAR 09b, VAR 09a];

2) overlays where the knowledge area of a node is centered around its coordinate. They use either unstructured overlays [KEL 03] or Voronoi tessellations [BEA 07a, BEA 07b, HU 06, HU 04]. In these systems, the playing area is included in, or coincides with, the knowledge area.

This section focuses on one of the state-of-the-art overlays for online life simulation games: Solipsis [KEL 03] and its extension with Blue Banana [LEG 10].

5.5.1. *Solipsis overview*

Solipsis is an overlay designed to sustain a distributed virtual environment. Each node of the Solipsis overlay is responsible for one avatar. In Solipsis, the knowledge area and the playing area coincide. The objects of a playing area are, therefore, replicated on the nodes that manage the avatars within this playing area. In Solipsis, elders of a node are in its playing area: they are the nodes that manage the objects in the playing area.

Solipsis maintains a set of direct neighbors for each node. Nodes communicate using message passing through the overlay: latency increases with the distance (measured in number of hops) in the overlay. Solipsis tries to maintain its elders in its neighborhood to communicate efficiently. If two avatars A and B are neighbors in the virtual environment, the Solipsis overlay adapts itself so that B will *eventually* be in A's neighborhood and vice versa. In order to ensure that behavior, Solipsis is based on two fundamental rules:

1) *local awareness*: an avatar a has a circular playing area ω_a centered on the avatar. If another avatar b is inside ω_a, the nodes of a and b must be neighbors in the overlay. The size of ω_a is dynamically adjusted to ensure that a has a number of neighbors contained between a minimum and a maximum limit. When a node enters a high-density zone, ω_a increases, and when it enters a low-density zone, ω_a decreases;

2) *global connectivity*: let N_c be the neighbor set of a node c in the overlay. The avatar of c must be located inside the convex hull of the set formed by avatars of N_c. Figure 5.8 illustrates this property. On the left, c is located inside the convex hull and on the right, after a move of the avatar, the property is broken. This property ensures that the avatar does not neglect a part of its playing area, causing (i) inconsistent

rendering and, (ii) possibly partitioning the Solipsis overlay graph (and, therefore, making parts of the virtual world unreachable).

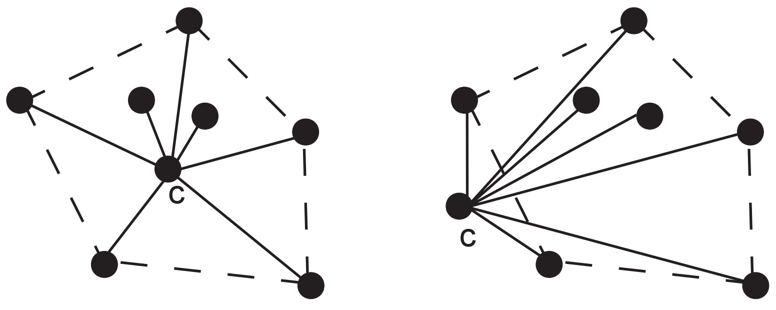


Figure 5.8. *While a peer moves its convex hull has to be reconstructed*

To ensure these rules, Solipsis implements a mechanism called *spontaneous collaboration*. At each moment, thanks to periodic updates, a node sends its coordinates and the size of its knowledge areas to its neighbors. When a node detects that one of its neighbors enters the knowledge area of another of its neighbors, it sends a message to both entities to warn them that the local awareness rule is about to be broken.

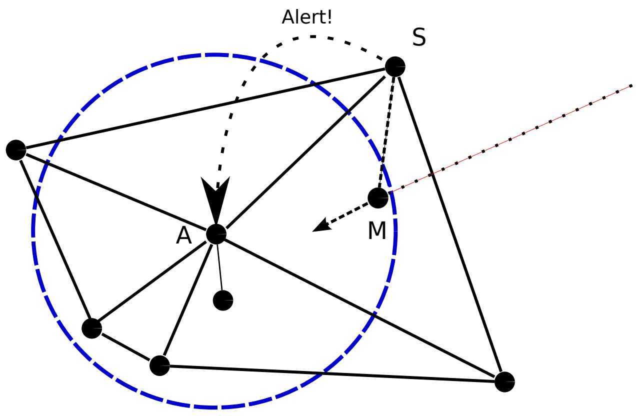


Figure 5.9. *Solipsis spontaneous collaboration mechanism*

This algorithm is illustrated in Figure 5.9. A and M are initially in the knowledge area of S. S sends an alert to node A and M to warn them when it detects that M enters the knowledge area of A. As they receive that message, the two entities (A and M in Figure 5.9) become neighbors. Simulations showed that this technique is very efficient: most of the time, a node receives a warning message and does not have to initiate a costly new-neighbor query. The global connectivity rule ensures that a node is always surrounded by its neighbor set, making spontaneous collaboration more efficient.

To conclude, if the local awareness rule is violated for a node n, it means that an avatar has arrived in the playing area of n and is not yet included to the local knowledge of n, causing a transient failure. If the global connectivity rule is violated for a node n, it means that n is not surrounded by its neighbor set. It will then not receive spontaneous data updates for a part of its playing area, which will necessarily lead to transient failures.

An avatar keeps breaking fundamental rules as long as it moves because the spontaneous collaboration mechanism is not always able to react on time. For that reason, a more efficient anticipation mechanism is required.

5.5.2. *Anticipating avatar mobility to pro-actively adapt the overlay*

Blue Banana [LEG 10] is an extension of Solipsis that pro-actively adds new neighbors "ahead" of the player by anticipating its movements. For this purpose, it finds nodes in the direction of the avatar's movement and add them to a new set called the prefeteched set. Once the moving avatar approaches a prefetched node, the prefetched node is added to the regular neighbor set of Solipsis that defines the overlay topology. Hence, Blue Banana substantially helps native algorithms in Solipsis to maintain the fundamental rules, minimizing resulting transient failures.

Algorithm description

Technically, if the algorithm observes that the avatar of a node B (for Blue Banana) is moving fast and straight, and if the prefetched neighbor set is not full, B starts searching for new prefetched neighbors in the movement direction. It sends a message to its neighbor that is closest to its *movement vector* as illustrated by Figure 5.10. The message contains the number of prefetched neighbors that B is willing to retrieve (called the TTL), the position of B, its direction, and its speed.

As B is moving during the message transfer, upon receipt of a prefetching request on a node R (for Receptor), R estimates the probable position p of B when B will receive a response from R. For this purpose, R estimates the network latency and uses the initial position and the speed of the avatar. It then builds a probability cone that begins at the apex p and that has the direction of B. The shape of the cone decreases with B's speed. R responds to B with all its neighbors (itself included) that are in this cone. R then sufficiently decreases the TTL and if the TTL remains greater than zero, R forwards the request to its neighbor closest to B's vector movement.

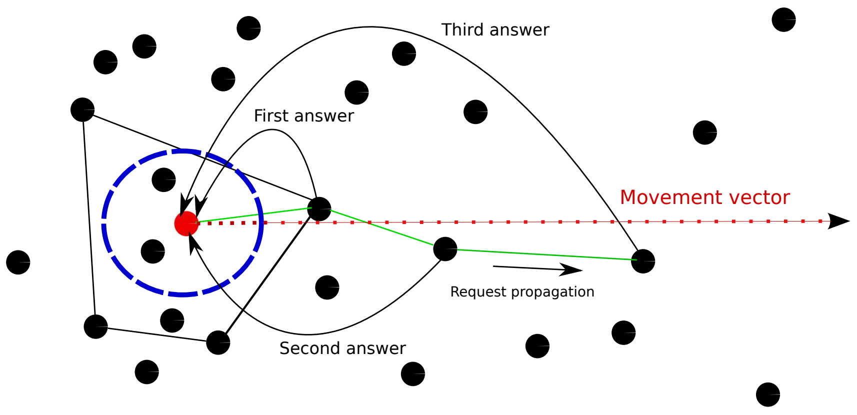


Figure 5.10. *Blue Banana prefetching algorithm*

Network overhead

Blue Banana does not interfere with the maintenance protocol of Solipsis: the prefetched neighbors are not placed in the regular Solipsis neighbor set, but in a separate one. Therefore, Solipsis does not use network resources on maintaining links with prefetched neighbors. Blue Banana does not spend network resources to maintain a link with a node that is useless *in the present* as it is not yet in the playing area.

As a consequence, once inserted in the prefetched neighbor set, the position of a node's avatar is not updated, while it can move outside the probability cone. Blue Banana automatically removes useless prefetched neighbors (i) when they have been overtaken by the moving avatar, (ii) when the avatar changes its direction or (iii) when the avatar slows down. It is possible to consider another policy by periodically updating the state of the prefetched neighbors. However, there is a risk of spending network resources to update possibly useless nodes.

In order to compensate the small network overhead, Blue Banana nodes take advantage of the high predictability of the avatar movement in desert zones. In Solipsis, a node periodically propagates the coordinates of its avatar to all the members of its neighbor set, so the neighbor-nodes can update their view of the virtual environment. Blue Banana doubles the period of such updates for nodes when it detects the avatar is moving fast and straight. The neighbors of that node simply predict the position of the avatar between two updates by using its initial position and its speed. This technique is a simple form of *dead reckoning* [2], but it could easily be enhanced with more sophisticated mechanisms widely used in online gaming [BHA 08, PAN 02, PAN 07].

2. Recall that dead reckoning is the process of estimating one's current position based upon a previously determined position.

To summarize, life-simulation games do not exhibits hard constraints in term of game reactivity. However, the number of players involved in this kind of game can reach hundreds of thousands. The scalability therefore becomes a problem and a DHT becomes inefficient. Overlays tailored for MMOGs attempt to solve this scalability problem by (i) building knowledge areas that correspond to player areas and by (ii) using node's elders as neighbors in the network. These topologies require frequent updates when the avatars move and, therefore, require the addition of neighbor prefetching mechanisms to increase the gaming experience.

5.6. Conclusion

Nowadays game applications become important both in terms of number of the users and in term of the financial impact. Successful games are now multi-player: hundreds of thousands of players are spread all over the world [LAB 11, ENT 11]. Building the software infrastructure for these applications is still challenging.

The currently deployed systems are based on the classical "client-server" paradigm and do not scale with the number of nodes. Additional rules have been introduced to limit the number of players interacting on the same set of servers at a given time (e.g. Second Life "islands" or World of Warcraft "realms" [PIT 07]).

Classical overlays built for file-sharing applications also do not scale with the number of players. However, FPS games that exhibit real-time constraints that are successfully built on DHT.

Only really recent research work has proposed overlays tailored for MMOGs. These overlays try to minimize the latencies between nodes and their elders by moving them closer in the overlay. For this purpose, they identify the knowledge and the playing areas. These overlays react to player movements. Observing avatar movements to prefetch neighbors in the movement direction, which minimizes the reaction time.

Open challenges

New overlays tailored for MMOGs minimize the communication latencies but these systems only provide *best effort* "guarantees". Therefore, how are cheaters dealt with in such environments? How is players collusion avoided? It is mandatory to ensure (i) privacy: unauthorized information must not be obtain about another avatar; (ii) integrity: the game rules have to be respected; (iii) availability: the system has to remain responsive. Few recent research works focus on these issues, however, these are still open issues [NEU 07].

We believe that the next generation of peer-to-peer overlays for large-scale game applications will be multi/complex overlays: malleable overlays such as Solipsis or, such as those based on Voronoi tessellations are well adapted for local interactions, however they are not suitable for global search or publish/subscribe. However, such "global" operations are likely to be needed: e.g. to maintain virtual bank accounts, to count player "credits", or more simply, to find information within a virtual environment.

Research in this field is still very active; new systems will emerge in the coming years.

5.7. Bibliography

[ALV 07] ALVES R. T., ROQUE L., "Because players pay: the business model influence on MMOG design", AKIRA B., Ed., *Situated Play: Proc. of the 2007 Digital Games Research Association Conference*, Tokyo, The University of Tokyo, p. 658–663, September 2007.

[BEA 07a] BEAUMONT O., KERMARREC A.-M., MARCHAL L., RIVIERE E., "VoroNet: a scalable object network based on Voronoi tessellations", *21th International Parallel and Distributed Proc. Symposium (IPDPS 2007), Long Beach, USA*, p. 26-30, March 2007.

[BEA 07b] BEAUMONT O., KERMARREC A.-M., RIVIERE E., "Peer to Peer Multidimensional Overlays: Approximating Complex Structures", TOVAR E., TSIGAS P., FOUCHAL H., Eds., *OPODIS*, vol. 4878 of *LNCS*, Springer, p. 315-328, 2007.

[BHA 04] BHARAMBE A. R., AGRAWAL M., SESHAN S., "Mercury: supporting scalable multi-attribute range queries", *SIGCOMM Comput. Commun. Rev.*, vol. 34, num. 4, p. 353–366, ACM, 2004.

[BHA 06] BHARAMBE A., PANG J., SESHAN S., "Colyseus: a distributed architecture for online multiplayer games", *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, Berkeley, CA, USA, USENIX Association, p. 155-168, 2006.

[BHA 08] BHARAMBE A. R., DOUCEUR J. R., LORCH J. R., MOSCIBRODA T., PANG J., SESHAN S., ZHUANG X., "Donnybrook: enabling large-scale, high-speed, peer-to-peer games", BAHL V., WETHERALL D., SAVAGE S., STOICA I., Eds., *SIGCOMM*, ACM, p. 389-400, 2008.

[BRU 02] BRUNET R., "Lignes de force de l'espace Européen", *Mappemonde*, vol. 66, p. 14-19, 2002.

[CHE 06] CHECHKIN A., GONCHAR V., KLAFTER J., METZLER R., "Fundamentals of Lévy flight processes", *Advances in chemical physics*, vol. 133B, p. 439-496, 2006.

[ENT 11] BLIZZARD ENTERTAINMENT INC, "World of Warcraft", http://us.battle.net/wow/en/, January 2011.

[FRE 08] FREY D., ROYAN J., PIEGAY R., KERMARREC A., ANCEAUME E., FESSANT F. L., "Solipsis: a decentralized architecture for virtual environments", *The Second International Workshop on Massively Multiuser Virtual Environments at IEEE Virtual Reality (MMVE' 09)*, Lafayette, USA, March 2008.

[HU 04] HU S.-Y., LIAO G.-M., “Scalable peer-to-peer networked virtual environment”, *NetGames ’04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, New York, NY, USA, ACM, p. 129–133, 2004.

[HU 06] HU S.-Y., CHEN J.-F., CHEN T.-H., “VON: a scalable peer-to-peer network for virtual environments”, *IEEE Network*, vol. 20, num. 4, p. 22-31, Jully 2006.

[IIM 04] IIMURA T., HAZEYAMA H., KADOBAYASHI Y., “Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games”, *NetGames ’04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, New York, NY, USA, ACM, p. 116–120, 2004.

[KEL 03] KELLER J., SIMON G., “Solipsis: a massively multi-participant virtual world”, *International Conference on Parallel and Distributed Processing Techniques and Applications – PDPTA*, p. 262-268, 2003.

[KUM 08] KUMAR S., CHHUGANI J., KIM C., KIM D., NGUYEN A., DUBEY P., BIENIA C., KIM Y., “Second Life and the new generation of virtual worlds”, *Computer*, vol. 41, num. 9, p. 46-53, IEEE Computer Society, 2008.

[LA 08] LA C.-A., MICHIARDI P., “Characterizing user mobility in Second Life”, *SIGCOMM 2008, ACM Workshop on Online Social Networks, August 18-22, 2008, Seattle, USA*, August 2008.

[LAB 11] LAB L., “Second Life”, http://secondlife.com/, January 2011.

[LEG 10] LEGTCHENKO S., MONNET S., THOMAS G., “Blue Banana: resilience to avatar mobility in distributed MMOGs”, *The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, Chicago, USA, July 2010.

[LIA 08] LIANG H., TAY I., NEO M. F., OOI W. T., MOTANI M., “Avatar mobility in networked virtual environments: measurements, analysis, and implications”, *CoRR*, vol. abs/0807.2328, 2008.

[MIL 09] MILLER J. L., CROWCROFT J., “Avatar movement in world of warcraft battlegrounds”, *Netgames 2009*, IEEE, November 2009.

[MON 06] MONNET S., MORALES R., ANTONIU G., GUPTA I., “MOve: Design of An Application-Malleable Overlay”, *Symposium on Reliable Distributed Systems 2006 (SRDS 2006)*, Leeds, UK, p. 355-364, October 2006.

[NEU 07] NEUMANN C., PRIGENT N., VARVELLO M., SUH K., “Challenges in peer-to-peer gaming”, *SIGCOMM Comput. Commun. Rev.*, vol. 37, num. 1, p. 79–82, ACM, 2007.

[ORA 01] ORAM A., “*Peer-to-Peer: harnessing the power of disruptive technologies*”, Chapter Gnutella, p. 94-122, O’Reilly, May 2001.

[PAN 02] PANTEL L., WOLF L. C., “On the suitability of dead reckoning schemes for games”, WOLF L. C., Ed., *NETGAMES*, ACM, p. 79-84, 2002.

[PAN 07] PANG J., UYEDA F., LORCH J. R., “Scaling peer-to-peer games in low-bandwidth environments”, *IPTPS ’07: Proc. of the 6th International Workshop on Peer-to-Peer Systems*, February 2007.

[PIT 07] PITTMAN D., GAUTHIERDICKEY C., "A measurement study of virtual populations in massively multiplayer online games", *NetGames '07: Proc. of the 6th ACM SIGCOMM workshop on Network and system support for games*, New York, NY, USA, ACM, p. 25–30, 2007.

[RHE 08] RHEE I., SHIN M., HONG S., LEE K., CHONG S., "On the Levy-Walk nature of human mobility", *INFOCOM*, IEEE, p. 924-932, 2008.

[STO 01] STOICA I., MORRIS R., KARGER D., KAASHOEK F., BALAKRISHNAN H., "Chord: a scalable peer-to-peer lookup service for internet applications", *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM '01)*, San Diego, USA, p. 149-160, August 2001.

[VAR 09a] VARVELLO M., DIOT C., BIERSACK E. W., "A walkable Kademlia network for virtual worlds", *Infocom 2009, 28th IEEE Conference on Computer Communications, April 19-25, 2009, Rio de Janeiro, Brazil*, 04 2009.

[VAR 09b] VARVELLO M., DIOT C., BIERSACK E. W., "P2P Second Life: experimental validation using Kad", *Infocom 2009, 28th IEEE Conference on Computer Communications*, Rio de Janeiro, Brazil, p. 19-25, April 2009.

[VOU 04] VOULGARIS S., KERMARREC A. M., MASSOULIE L., VAN STEEN M., "Exploiting semantic proximity in peer-to-peer content searching", *10th International Workshop on Future Trends in Distributed Computing Systems (FTDCS 2004)*, Suzhou, China, May 2004.

[VOU 06] VOULGARIS S., RIVIERE E., KERMARREC A.-M., VAN STEEN M., "SUB-2-SUB: self-organizing content-based publish and subscribe for dynamic and large scale collborative networks", *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS)*, Santa Barbara, USA, February 2006.

SECOND PART

Distributed, Embedded and Real-Time Systems

Chapter 6

Introduction to Distributed Embedded and Real-time Systems

Building distributed real-time embedded systems requires a stringent methodology, from early requirement capture to final program implementation. Real-time systems must enforce strict timing constraints such as a deadline. Embedded systems come with strict resource constraints, such as a restricted memory usage. But similar to most complex systems, real-time embedded systems are inherently distributed systems. Moreover, they have to deal with fault tolerance issues and distribution can help handle such a system requirement. However, we shall see that distribution adds significant complexity to the design of real-time embedded systems.

Safety-critical systems are typical examples of distributed real-time embedded systems. They are used in many domains, such as aerospace, avionics, or medicine. Control systems deployed on air and space platforms represent one of the most safety-critical categories of software. There are stringent standards of code review and certification [RTC 92] that must be met before deployment onboard the platform. In this chapter, we briefly depict a typical safety-critical architecture for avionics systems and, more specifically, how distribution introduces difficult challenges.

The design of such safety-critical system architectures involves several steps and at each level the system must be carefully specified, analyzed, implemented, and certified. We show how this development process has to be enriched to address the inherent complexity introduced by distribution. This gives us the opportunity to highlight the contributions of the next chapters.

Chapter written by Laurent PAUTET.

Distributed Systems: Design and Algorithms Edited by Serge Haddad, Fabrice Kordon, Laurent Pautet and Laure Petrucci

6.1. Distributed real-time embedded systems

6.1.1. *Real-time systems*

A real-time system is a system that has to respond to externally generated input stimuli within a finite and specified delay [BUR 01]. The input stimuli come periodically or aperiodically from sensors, such as pressure sensors. These hardware devices collect data from the environment and send them to the system for processing and then reaction. As a consequence, the system changes the environment through actuators such as engines. A major concern of these systems is ensuring a deterministic behavior which is a prerequisite to guarantee timing constraints.

6.1.2. *Embedded systems*

There is no single definition reflecting all kinds of embedded systems. We may define an embedded system as a special-purpose computer system designed to perform one or a few dedicated functions, often with real-time computing constraints [WIK 10]. As embedded systems are dedicated to specific tasks, design engineers can optimize them, reducing their size and cost, or increasing their reliability and performance. Generally speaking, an embedded system also needs to check resource usage. Typically, a strict determinism on memory usage may prevent the use of memory dynamic allocation.

6.1.3. *Distribution in Real-Time Embedded (DRE) systems*

Distribution enables the improvement of the performance or the redundancy of real-time embedded systems. Distributed architectures are required to satisfy real-time constraints (by increasing the computation capabilities), as well as to take into account the geographical localization of the resources (sensor/actuator, computation, memory) of real-time embedded systems. However, distribution makes it more difficult to guarantee the timing and resource constraints specified by real-time embedded systems.

Safety-critical systems represent a typical category of these DRE systems. Transportation systems, such as avionics systems, represent one of the most safety-critical categories of software. They perform critical functions and contain classified data; therefore they must be safe, reliable, and secure.

For these reasons, their design process has to conform to strict standards such as DO-178B [RTC 92] for aircraft transport, CENELEC 50128/50129 [CEN 00, CEN 01] for railway transport or ECSS-E40A [ESA 03] for the space industry.

Conforming to such standards requires the definition of a rigorous design process as described in Chapter 9. Guaranteeing the schedulability and safety properties of the application is one of the costliest aspects of this process and Chapter 7 details the different issues in this area.

6.2. Safety critical systems as examples of DRE systems

Avionics systems represent one of the most safety-critical categories of software. To illustrate this chapter, we briefly introduce the main issues to address in this specific domain.

6.2.1. *Avionics safety-critical systems*

A typical avionics system architecture is designed as a federated architecture of dedicated boxes. The applications are physically distributed from one another in such a way that failures of one or more applications do not affect others. However, such architectures are expensive to build in terms of space, weight, and power requirements.

6.2.2. *IMA*

To overcome the disadvantages induced by physical separation, a new avionics architecture model, known as integrated modular avionics (IMA), has been developed. IMA reduces the space, weight, and power requirements of the aircraft, reduces spares holding, and therefore, reduces complexity, costs, and time of development.

However, if the number of computing modules has been decreased from 37 units to six or seven units, the global architecture remains distributed. To illustrate the impact of distribution, the architecture adopted by Airbus for the new A380 generation consists of the utilization of AFDX [ENG 03], a solution based on switched Ethernet, which eliminates the inherent indeterminism of this technology.

6.2.3. *DO-178B*

In order to provide an effective IMA system, standards have to be followed to allow for methodical development, validation, and testing, as well as providing a standards-based application programming interface (API) to allow for software portability and modularity.

DO-178B [RTC 92] provides guidelines for the production of software for airborne systems and equipment. The objective of the guidelines is to ensure that the software performs its intended function with a level of confidence in safety that complies with airworthiness requirements.

6.2.4. *ARINC 653*

ARINC 653 [ENG 97] is a standard defining operating system service to support avionics software. ARINC 653 operating systems isolate software applications in terms of space and time and avoid error propagation across them. These software applications or partitions are configured and executed as if they were running on an independent processor so that a partition at a given criticality level cannot impact partitions that run at a different criticality level.

ARINC 653 specification forces applications to be isolated, ARINC 653 conformance can be a step toward DO-178B certification and provides a framework for building an operating system environment to support IMA.

6.2.5. *ARINC 653 Principles*

ARINC 653 outlines specifications for a system execution environment where isolated applications can run independently of one another, each in its own virtual container called a partition. ARINC 653 partitions are isolated in terms of space and time.

Space isolation means that each partition cannot read/write data from/to other partition memories in order to preserve data integrity and confidentiality.

Time isolation means that the system schedules partitions periodically. During their execution, partitions manage their resources and schedule their tasks.

An example of temporal partitioning would be a single core CPU (central processing unit) that must be shared between partitions by giving each partition a deterministic time-slice of the overall CPU schedule. An example of spatial partitioning would be main memory that must be divided amongst the partitions (with hardware restrictions to prevent unauthorized access to a partition's allocated memory by unrelated partitions).

As a result, the presence of a partition does not affect the performance of another unrelated partition and faults within a partition are isolated from other partitions.

Avionics systems require elaborate and thus expensive certification. The ARINC 653 standard eliminates the need to reexamine unchanged applications on an IMA system because the guaranteed isolation it provides limits certification efforts to only the modified partition. Certification is only required on the modified partitions because faults are contained within partitions.

Each partition is assigned a DO-178B criticality level from level A (highest) to level E (lowest). The higher the criticality level, the higher the certification requirements are. As a consequence, the ARINC 653 executive platform must be certified to at least the highest level of criticality of the system partitions.

The ARINC 653 standard divides the partitioning environment into three layers (illustrated in Figure 6.1): hardware (processor, memory, I/O devices, etc.), the board support package (interfacing with the hardware components), and operating system.

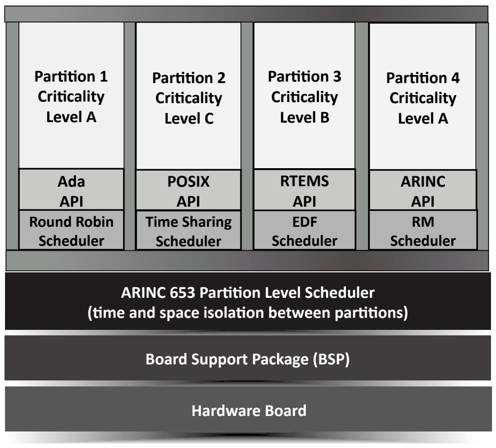


Figure 6.1. *Hierarchical ARINC 653 architecture*

ARINC 653 services are structured as follows:

– partition service to manage partition mode;

– process service to manage processes inside partitions;

– time service to manage timed events and periodic processes;

– interpartition communication service to enable communication between partitions through fixed-sized messages (sampling ports) and variable-sized messages (queuing ports);

– intrapartition communication service to enable interprocess communication through unqueued messages (blackboards) and queued messages (buffers) but also take over interprocess synchronization through notification (events) and control access to shared resources (semaphores);

– health monitoring service to catch errors and to attach a recovering policy at the process, partition, or kernel level.

The processor allocation is made according to a two-level hierarchical scheduling:

– the partitions are cyclically activated. This partition scheduling is usually off-line;

– the second scheduling level is related to the task scheduling inside a partition. This scheduling may be online.

6.3. Design process of DRE systems

Building DRE systems involves many tightly coupled steps, from requirements capture (number of tasks and their interactions, non-functional attributes) to validation (feasibility of scheduling) down to implementation and testing.

There are numerous mature results as well as outgoing works to help the user in one design process. In the context of safety-critical systems, we provide an overview of the possible approaches, some of them being detailed in the following chapters.

6.3.1. *Modeling*

The distance between requirements and implementation must be kept minimal all along the process: non-functional attributes have to be carefully respected when implementing tasks; any change in the specification has to be carefully propagated at the implementation level; interactions between entities have to be mapped onto run-time entities in a safe manner (deadlock free, no starvation, no overrun, etc.).

Therefore, developers and system architects need common interchange models to work cooperatively on their requirements and concerns. Several modeling languages help to represent safety-critical systems architectures. Some popular ones are UML and, in particular, the MARTE profile [OMG 08], SysML [OMG 07], or AADL [SAE 08].

In the context of our safety-critical system example, AADL offers an interesting solution. The AADL (*Architecture Analysis and Design Language*) recently appeared as an architecture description language suitable to describe systems, from high-level concerns down to implementation.

First, it proposes a unique representation when other modeling languages may require the use of several profiles but also model transformations between these profiles. Second, AADL can be used as a backbone, meaning that it is involved in all the different steps of the process (modeling, analysis, code generation). Third, AADL comes with an ARINC 653 modeling annex, which proposes guidelines to represent ARINC 653 systems. In particular, this aids model inter-communications, intra-communications, and space/time isolation.

6.3.2. *Analysis*

Following the modeling phase, the safety-critical avionics system previously described has to be analyzed in order to enforce the expected functional and non-functional properties. One approach may be to run schedulability tests to check that the system has a correct temporal behavior.

Many schedulability tests have been created [LIU 73]. A schedulability test requires that the target system fulfills a set of specific assumptions. Therefore, it may be difficult for a designer to choose the relevant analytical method. Moreover, only few modeling languages or engineering tools allow the designer to apply the results of real-time scheduling.

Coupling modeling and analysis tools requires that both ends strictly comply with the same semantic definition of the exchanged model. As far as our safety-critical system is concerned, the Cheddar scheduling analysis tool can help us by automatically checking scheduling feasability on each partition and on each processor as it reads AADL models [SIN 07]. In a context of non-hierachical scheduling, a similar approach would have been possible with MAST [HAR 01] using MARTE as the modeling language.

6.3.3. *Implementation*

The differences between the system model, analysis models, and the generated code introduce a semantic gap, reducing the confidence in the whole process. Differences are due to:

1) the different levels of abstraction between the initial model, intermediate analysis models, and the generated code (or implementation model) and

2) the resources of the execution support (middleware) not taken into account in relevant analysis stage.

In [BOR 05], Bordin and Vardanega state that the goal of automatic code generation is precisely to reduce the gap between the system as it has been modeled and analysed and the system as it is produced. Such a process has been successfully implemented and described in [HUG 08]. Returning to the safety-critical system domain, this automatic code generation can also lead to the configuration of executive platforms and, in the context of our avionics system example, to the configuration of the ARINC 653 kernel as described in [DEL 11].

6.3.4. *Certification*

As for code production, the configuration and execution of avionics systems need to be certified against certification standards [RTC 92]. Most of the time, the certification process is manually achieved and consists of ensuring that system implementation is compliant with its specification. Due to the time required to inspect and track each requirement from a text document (specifications) to the corresponding piece of code, this is a very costly and error-prone.

Of course, one important trend in the design of safety-critical systems consists of automating the design process where possible and integrating this automated process to some of the steps required by the certification. For instance, the authors of [DEL 11] integrate the ability to perform code coverage technics in their design process thanks to the GNAT toolchain [BOR 10] and to the emulator QEMU [BEL 05].

6.4. Objectives of Part 2

The goal of this part is to highlight some well-known or innovative approches to deal with distribution in the context of real-time embedded systems. The following presentations mainly focus on safety-critical systems and aim to address one or several steps of the design process in depth.

This part contains three chapters.

First, Chapter 7 presents holistic analysis, a well-known method used to compute the schedulability of DRE systems. This chapter requires no specific background about scheduling theory. It starts by presenting static priority preemptive scheduling analysis. The authors present scheduling algorithms that guarantee schedulability for a particular set of threads with or without jitters. Then, the approach is extended to systems in which tasks with arbitrary deadlines communicate via messages over a communication network. The basic idea is to interpret the message delay induced by the communication system as a release jitter of the receiver task. This first contribution tackles the analysis and execution steps described in the current chapter.

Second, Chapter 8 presents the design of adaptative real-time embedded systems. These adaptation requirements may help to implement fault-tolerance mechanisms or represent different phases of the system life cycle. The chapter focuses on mode switches as a possible implementation of these adaptation requirements. If this chapter gives some complementary schedulability results to Chapter 7, it also details some fundamental insights in the design process of adaptative real-time embedded systems. In particular, the chapter comes with the modeling, the analysis, and the implementation of a robot, which can operate either in automatic or manual modes. This second contribution addresses the modeling and analysis steps described in the current chapter.

Finally, Chapter 9 presents an innovative approach to designing the new generation of space systems. This approach captures the system architecture with a modeling language, defines the data model, proceeds to a possible schedulability analysis, enforces optimized code generation, and finally, combines applicative and middleware components into an homogenous software application. This approach is supported by a toolset in order to make the process as automated as possible. This work takes advantage of several standards (AADL, ASN.1, etc.) and the current state of the toolset is close to a commercial product. This last contribution covers all the steps of the design process described in the current chapter.

6.5. Bibliography

[BEL 05] BELLARD F., "QEMU, a fast and portable dynamic translator", *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, USENIX Association, p. 41–41, 2005.

[BOR 05] BORDIN M., VARDANEGA T., "Automated Model-Based Generation of Ravenscar-Compliant Source Code", *ECRTS'05: Proceedings of the 17th Euromicro Conference on Real-Time Systems*, Washington, DC, USA, IEEE Computer Society, p. 59–67, 2005.

[BOR 10] BORDIN M., GASPERONI F., "Towards verifying model compilers", *ERTSS'10: Proceeding of the Embedded Real-Time Systems and Software Conference*, Toulouse, France, SEE, May 2010.

[BUR 01] BURNS A., WELLINGS A. J., *Real-time systems and programming languages: Ada 95, Real-Time Java and Real-Time POSIX*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2001.

[CEN 00] CENELEC, *Railway Applications - communication, signalling and processing systems - software for railway control and protection systems, EN 50128*, 2000.

[CEN 01] CENELEC, *Railway applications - communication, signalling and processing systems - safety related electronic systems for signalling, EN 50129*, 2001.

[DEL 11] DELANGE J., PAUTET L., KORDON F., "A model-based approach to configure and reconfigure avionics systems", KHALGUI M., HANISCH H.-M., Eds., *Reconfigurable Embedded Control Systems: Applications for Flexibility and Agility*, IGI, p. 509-541, 2011.

[ENG 97] AIRLINES ELECTRONIC ENGINEERING, "ARINC Specification 653: Avionics Application Software Standard Interface", Annapolis, MD, Aeronautical Radio, Inc, January 1997.

[ENG 03] AIRLINES ELECTRONIC ENGINEERING, "ARINC Specification 664: Aircraft data network, part 7-avionics full duplex switched ethernet (afdx) network", Annapolis, MD, Aeronautical Radio, Inc, January 2003.

[ESA 03] ESA, "Part 1: Principles and requirements", *ECSS-E-40 Parts 1B : Space engineering - Software*, November 2003.

[HAR 01] HARBOUR M. G., GARCIA J. G., NCIA GUTIERREZ J. P., MOYANO J. D., "MAST: modeling and analysis suite for real time applications", *ECRTS'01 : Proceeding of the 13th Euromicro Conference on Real-Time Systems*, Delft, Netherlands, IEEE Computer Society, p. 125–134, June 2001.

[HUG 08] HUGUES J., ZALILA B., PAUTET L., KORDON F., "From the prototype to the final embedded system using the Ocarina AADL tool suite", *ACM Transactions in Embedded Computing Systems (TECS)*, vol. 7, num. 4, p. 1-25, July 2008.

[LIU 73] LIU C., LAYLAND J., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, vol. 20, num. 1, p. 46–61, 1973.

[OMG 07] OMG, "Systems modeling language (SysML)", 2007.

[OMG 08] OMG, "UML Profile for MARTE: modelling and analysis of real-time and embedded systems", *OMG document: ptc/08-06-09*, June 2008.

[RTC 92] RTCA, EUROCAE, DO-178B, Software considerations in airborne systems and equipment certification, 1992.

[SAE 08] SAE AS-2C ARCHITECTURE DESCRIPTION LANGUAGE SUBCOMMITTEE, *Architecture analysis and design language (AADL) v2 - Draft v1. 6, SAE AS5506*, SAE Aerospace, January 2008.

[SIN 07] SINGHOFF F., PLANTEC A., "AADL modeling and analysis of hierarchical schedulers", *SIGAda '07: Proceedings of the 2007 ACM International Conference on SIGAda Annual International Conference*, New York, NY, USA, ACM, p. 41–50, 2007.

[WIK 10] WIKIPEDIA, "Embedded system", http://en.wikipedia.org/wiki/Embedded_system, 2010.

Chapter 7

Scheduling in Distributed Real-Time Systems

7.1. Introduction

This chapter introduces temporal validation for distributed real-time systems. It assumes some basic knowledge about operating systems, but no background knowledge of scheduling theory is needed. It is mainly a commonly used method for validating distributed systems: the holistic analysis. It is intended to give the reader the basic knowledge required to deal with temporal validation and scheduling, as well as an introduction to more advanced topics.

Temporal validation is an important step in the validation of a time-critical system. Temporal validation usually takes place at the very end of the software development process, but more and more research studies address the conception stage, offering methods, such as the sensitivity analysis [VES 94, BIN 08, DOR 10], allowing a designer to choose, at an early stage, valid temporal parameters for his system. These methods rely on the fundamental theory presented in this chapter. This theory is based on temporal and structural abstractions (models) of the concurrent software running on the processors.

Different types of implementation are presented in section 7.2 in order to relate them to the models used for temporal validation in section 7.3, and then introduce the principles of holistic analysis. The holistic analysis is a conjoint validation of the tasks and the messages, based on their WCRT analysis. Section 7.4 shows how to compute the WCRT of the tasks for fixed-priority scheduling policies and deadline-driven scheduling, then section 7.5 focuses on the WCRT of the messages on a network.

Chapter written by Emmanuel GROLLEAU, Michaël RICHARD, and Pascal RICHARD.

Distributed Systems: Design and Algorithms Edited by Serge Haddad, Fabrice Kordon, Laurent Pautet and Laure Petrucci
© 2011 ISTE Ltd. Published 2011 by ISTE Ltd.

Section 7.6 uses these techniques in order to validate a case study. The conclusion discusses the limits and extensions of such a method.

7.2. Generalities about real-time systems

This section presents the fundamental links between the real-world (the application) and the models used for the temporal validation. Indeed, the first step in the validation process is to choose an accurate model for an application.

7.2.1. *Real-time systems*

A command and control system typically has to ensure the control of a real-world physical process. It obtains information about the state of the process using sensors (e.g. temperature sensor, speed sensor, attitude sensor, etc.), and acts on the physical process using actuators (e.g. electromechanical valve, engine, switch, etc.). Once provided with the knowledge of the instantaneous state of the system, the main role of this kind of system is to react to it in order for the controlled process to behave in a desired manner. The physical process can be as simple as a toy robot, or as complex as a car or an airplane.

The main difference between a classic software and a command and control software is that the time used by the system between the moment the sensors are read and the moment it effects the actuators has an impact on the process behavior. Therefore, when a system is critical, in other words when catastrophic consequences occur in case of a failure, it is often also time-critical, which means that the designer has to ensure that the duration between sensing and actuating is bounded. In this case, the command and control system is called a real-time system.

In some cases, the command and control system is embedded into the physical process, we then call it an embedded system. This kind of system has severe constraints of size, weight, energy consumption, physical robustness, etc. Therefore, the processors used in embedded systems have a very low computing power compared to industrial or personal desktop or laptop computers.

Table 7.2.1 gives some examples of applications and their usual classifications:

In the following sections, we consider real-time systems, for which a temporal validation is required.

7.2.2. *Synchronous or asynchronous system*

The implementation of a system has an impact on the way it is abstracted as a temporal model. The first question to consider is the parallelism of the actions. A

Application	Time-critical	Embedded	Distributed
Car control system	*yes*	*yes*	*yes*
Airplane control system	*yes*	*yes*	*yes*
UAV control system	*yes/no*	*yes*	*yes/no*
Cellphone	*no*	*yes*	*no*
Test bench instrumentation	*no*	*no*	*yes/no*
Power plant supervision & control	*yes*	*no*	*yes*

Table 7.1. *Applications and their classifications*

command and control system has to deal with information at different rates. For example, an UAV control system (see Figure 7.1) has to ensure several functions at different rates [TRA 06]:

– yaw, pitch, and speed control, using feedback loops at a frequency of 20 to 50 Hz, which use the data of an inertial measurement unit and a speed sensor as input, and control the flight control surfaces, as well as the engine throughput, in order to conform to an attitude (yaw and pitch) and speed command;

– sending telemetry data to a ground station using a wireless modem, a possible rate would be 10 Hz;

– receiving commands from the ground station (mode change, waypoints, etc.), at a typical rate of 10 Hz;

– supervision of the system: self-inspection in order to detect failures, at a possible rate of 5 Hz;

– navigation when following waypoints, using a geographic positioning system (GPS) in order to calculate an attitude and speed command such that the UAV is navigating toward a waypoint, at a typical rate of 2 to 4 Hz.

For a control and command system, two kinds of implementation can be used to ensure this parallelism: multitasking (also called multi-threading), and reactive implementation. Multitasking is referred to as asynchronous implementation, while a reactive implementation is usually a synchronous implementation.

7.2.2.1. *Asynchronous implementation*

Regarding scheduling point of view, a task is a thread executing a function in parallel with the rest of the program. Most operating systems, and real-time operating systems (RTOS), can handle several tasks in a process. A task can be seen as a function that will occur for a processor at a certain rate. The rate is said to be time-driven if the task release is driven by the internal clock of the system (e.g. using the internal clock in order to send telemetry every 100 ms). The rate is said to be event-driven if the task release is driven by an external event (e.g. every time data are received from

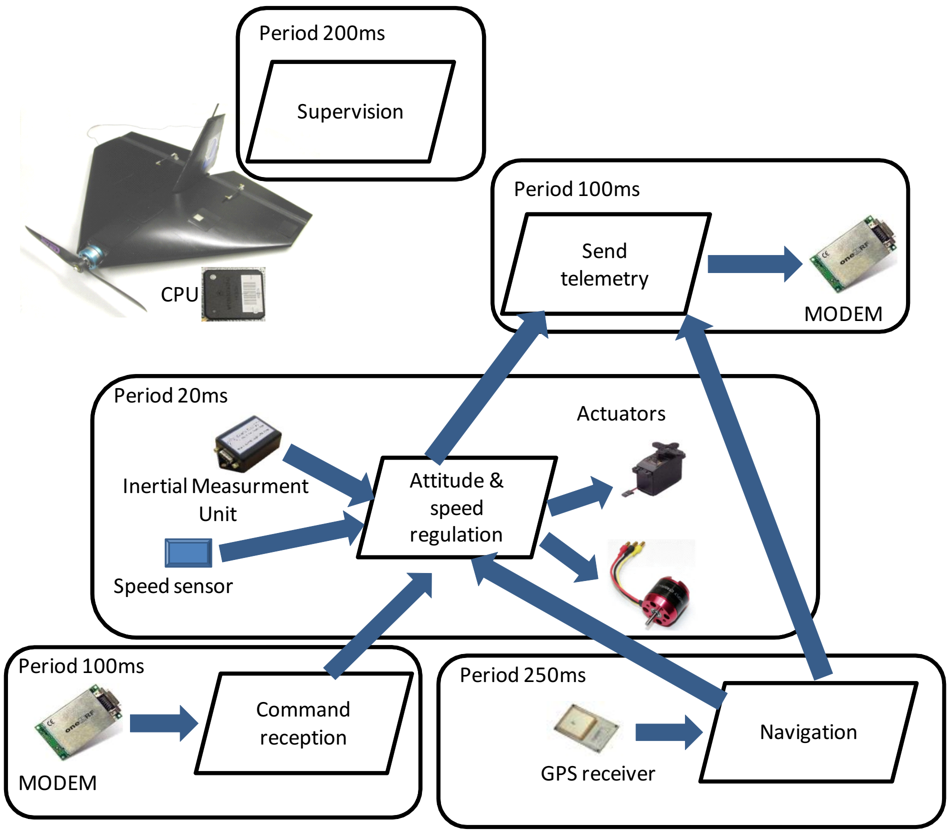


Figure 7.1. *UAV control system*

the inertial measurement unit, an interrupt is triggered on the processor, releasing the attitude control task).

Figure 7.2 illustrates a real-time system from the point of view of the designer and programmer. A RTOS is the interface between the application and the hardware. The RTOS controls the state of the tasks (running, ready, waiting, etc.), and its kernel is in charge of scheduling the system. Scheduling a system consists of choosing a ready task and assigning it to the processor. The scheduling decisions rely on the knowledge of the state of the tasks, and on the events expected by non-ready tasks, for example:

– the running task state changes (e.g. waiting for an event, a semaphore, a duration, a specific time, etc.),

– an internal event changes the state of a non-ready task into ready (such as the internal clock, the release of a semaphore, an internal message is sent to a task waiting for it, etc.),

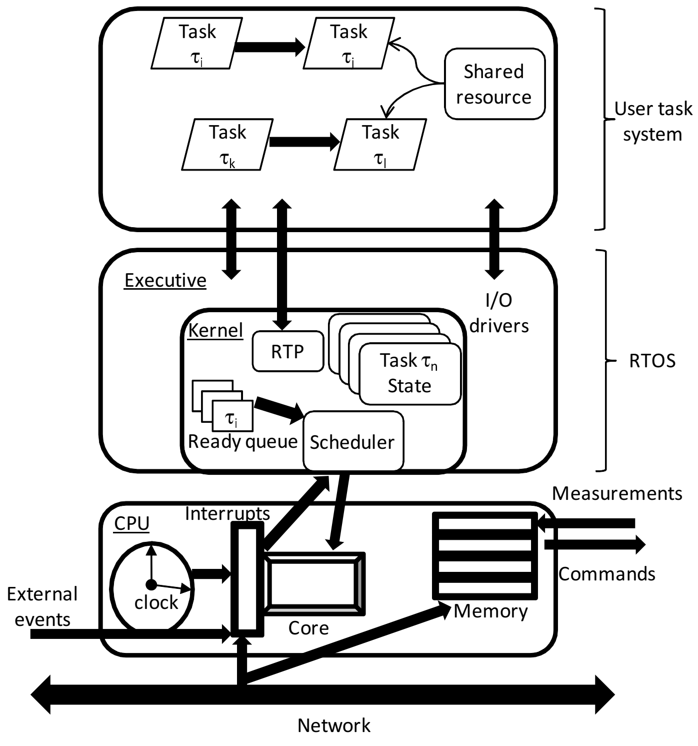


Figure 7.2. *Hardware and software layers of a command and control system*

– an external event triggers a task (e.g. an interrupt, such as the arrival of a message on the network is releasing a task, making it ready).

The kernel of a RTOS (proprietary RTOS like VxWorks and RTEMS, or conforming to standards like POSIX, OSEK/VDX, ARINC 653, Ada, etc.) is usually priority based, therefore, the task chosen by the kernel in order to use the processor is the ready task with the highest priority. Most of the RTOS are preemptive: when a high-priority task is released, the kernel is putting a lower priority running task in the ready queue, in order for the high-priority task to be executed.

7.2.2.2. *Synchronous implementation*

A synchronous implementation does not have to rely on a RTOS, because no priority-based scheduler is required. If the computing unit is "fast enough" compared to the dynamics of the system, then the processing time required for each functionality is very small compared to the inter-arrival time between two successive events. In this case, we can assume the treatment of an event to be instantaneous, so the system reacts synchronously to the event. Nevertheless, in embedded systems, the processors are chosen in order to fulfill the computing needs of the system for obvious economical and power consumption reasons. So in reality, the synchronous hypothesis is rarely applied in this way. However, it is possible to see the functionalities of a system as several finite automata (e.g. for the UAV presented in Figure 7.1, there would be an automaton for every function described in section 7.2.2), then to synchronize them into a single functionality, which is run cyclically or periodically on the processor. The synchronous hypothesis is valid if the delay implied by the cycle duration is "short enough" compared to the dynamics of the controlled system (i.e. such that the desired rates are matched), but the reactivity of a synchronous system to the occurrence of external events is as low as the cycle time of the system.

A synchronous implementation, even if less reactive than an event-driven asynchronous implementation, presents some advantages: it is less subject to the specification approximations. For example, we show in the following sections that the validation of an asynchronous task system relies on the description of a worst-case behavior, but the actual on-line (when the system is running) execution order of the tasks is unknown, and depends on the actual execution duration of the tasks.

In the following sections, we consider that the system is executed on a RTOS, with an asynchronous implementation, so we consider multitask systems, either time-driven, or event-driven. The sections concerning CPU (central processing unit) scheduling are useful for asynchronous systems, while the sections concerning worst-case message delay on different kinds of network concern both synchronous and asynchronous systems.

7.3. Temporal correctness

The main characteristic of a real-time system is that the time given to the system to react to an internal or external event is bounded in time. For example, an engineer in aerodynamics can ask that the yaw and pitch regulation takes place every 20 ms. This kind of time constraint is an end-to-end (from the sensor to the actuator) time constraint. If it is violated, then the stability of the aircraft cannot be ensured. A task reading the data bytes coming from a GPS receiver can be expected, when data are available, to read a byte of data in less than 170 microseconds. If the task cannot meet this time constraint, then the byte is lost, therefore, the frame containing the positioning information is lost. In this case, the task deadline is directly deriving from a constraint.

In the models used to validate the temporal behavior of a system, the timing constraints are represented by a deadline for every task of the system. Therefore, end-to-end deadlines are translated into deadlines applied to each task implied in the chain of treatment from one end to the other.

There are two kinds of timing constraint: hard (must never be violated) and firm (meeting the constraint is increasing the global quality of service). In the rest of the chapter, we consider hard deadline systems: a late result is considered to be a fault. The main question addressed is how to prove that the tasks of a distributed system always meet their respective deadline, in other words if the system is feasible.

7.3.1. *Feasibility*

As we have to deal with temporal constraints, every task has to be characterized with a worst-case execution time (WCET), giving the longest time necessary to its computation on the processor where it is assigned, and by a rate (usually a periodicity) giving its worst-case arrival pattern.

THEOREM 7.1 *Feasibility: a system of tasks is feasible with a scheduling algorithm if, and only if, for any duration of the tasks smaller than their WCET, and for every activation pattern corresponding to the rates of the tasks, all the tasks are always completely executed prior to their deadlines.*

Example 7.1 Suppose a task system composed of four tasks, τ_1 and τ_2 are executed on processor 1, while τ_3 and τ_4 are executed on processor 2. A network is shared between the two processors to pass a message between τ_1 and τ_4, and a message between τ_3 and τ_2. We suppose that the messages sent on the network are not preemptive (in practice, every message requires only one frame). The schedule of the tasks on their respective processor and of the messages on the network are shown in Figure 7.3. The up arrows represent task activations, and deadlines are symbolized by down arrows. A task, once activated, is sent to the ready queue, and has to be completely executed before its deadline. Part (a) of Figure 7.3 shows a simulation of the system, where the deadlines are met. Part (b) shows a simulation of the system where τ_3 is shorter, and where the deadline of τ_4 is not met.

This example illustrates that, in general, a simulation cannot be used to validate a distributed system, even when using the WCET. The observed phenomenon is called a scheduling anomaly. In most non-trivial systems, scheduling anomalies can occur, therefore, the validation steps consist of:

– defining worst-case scenarios for a task system;

– trying to show that the system is feasible by showing that even under these unfavorable conditions, the deadlines are met. In order to do so, the literature provides low computational complexity (polynomial or pseudo-polynomial) feasibility tests in order for them to be scalable to large systems. Nevertheless, for non-trivial systems,

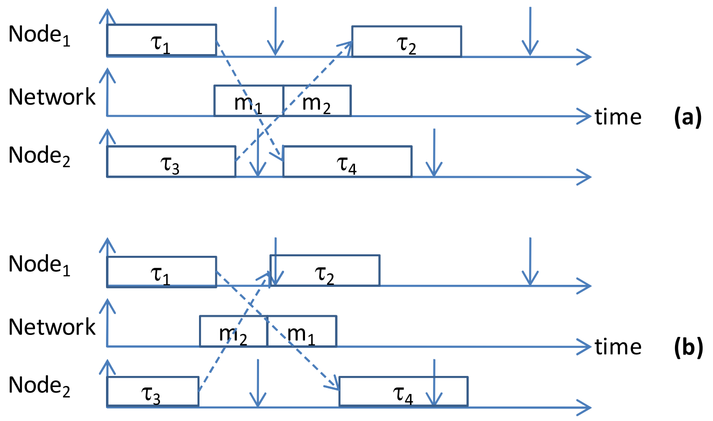


Figure 7.3. *A non-feasible system: a) tasks with an execution time equal to their WCET; b) a task is shorter than its WCET*

even on a single processor, the feasibility problem is NP-hard in the strong sense (there is no polynomial or pseudo-polynomial time algorithm to solve it). As a result, the worst-case validation can be pessimistic, which is the price paid in order to tackle intractability.

The holistic analysis is a method showing that a distributed system is feasible using pseudo-polynomial time algorithms.

7.3.2. *Principles of the holistic analysis*

The holistic analysis, introduced in [TIN 94a, TIN 94b], analyzes both the scheduling of the tasks and the messages. The underlying model is made of event-driven asynchronous tasks: a task on a CPU (node) waiting for a message is activated periodically but has to wait for the expected message to arrive on the node before it can be sent to the ready queue. This is based on the calculation of the WCRTs of the tasks and the messages. The response time of a task is the difference between its release time and the end of its execution. The WCRT is the longest response time achievable by a task, depending on the context.

Suppose, as in Figure 7.4, that a task τ_i sends a message m over a shared network to a task τ_j. For the task τ_j waiting for the message, the best case occurs when the message is available when the task is supposed to be released (i.e. if τ_j has a duration of 0, and the message is transmitted instantaneously), while the worst-case occurs when the message suffers the longest possible delay. Depending on what happens between the best and the worst-case, the task τ_j is sent into the ready queue at some time between its release date and the worst-case arrival time of the message. This incertitude is modeled as a release jitter.

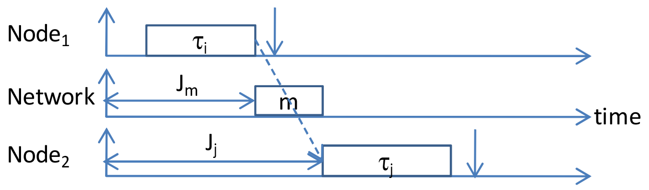


Figure 7.4. *Release jitter of a task and a message*

The fact that a task can suffer from a release jitter on a processor has an impact on the WCRT of this task, but also on the other tasks sharing the same processor, which can delay some messages sent on the shared network, and so on. As a result, the holistic analysis is a recurring process, taking into account the inter-dependences between the WCRT of the tasks and the WCRT of the messages by adding or increasing a release jitter to the tasks and messages, until a fixed-point is reached. This fixed point provides an upper bound of the WCRT of the tasks and the messages.

In the following sections, when a task τ_i is waiting for a message m sent by a task τ_j, we say that τ_j precedes m and that m precedes τ_i. In the holistic analysis, tasks and messages are both considered as tasks sharing a computing resource: a task is using a processor over a certain duration, and a message is using a network over a certain duration. If the system has to be validated, then both task and message durations must be bounded in time (a task has a worst-case execution time, and a message has a worst-case transmission time).

In the following sections, we assume some hypothesis on the system in order to validate it:

– the clocks are perfectly synchronized on the processors of the system;

– the network is perfect, there is no transmission error;

– the tasks are assigned to a processor and do not migrate;

– messages are read at the beginning of the tasks, and sent at the end;

– the communicating tasks are released at the same time on the different processors, and are executed at the same rate;

– the precedence graph implied by the inter-task communications does not have a circuit;

– all the communications are loosely coupled (i.e. there is no synchronous barrier or rendez-vous);

– tasks are non-reentrant (only one instance of a task can be ready at once) and sequential (they cannot be parallelized).

Suppose that n is the total number of tasks and messages in the system. We denote $i = 1..k, k \leq n$ the tasks, and $i = k + 1..n$ the messages of the system. Suppose that we have a function $WCRT$ computing the WCRT of a task or a message $i \in 1..n$. Giving the example shown on Figure 7.4, this function has to take into account the release jitter of the other tasks and messages (depending on the $WCRT$ of the message m, the $WCRT$ of the task τ_j is affected, and can delay tasks sharing the same CPU as τ_j and output messages). Therefore, the $WCRT$ of a task or a message, taking the worst-case jitter of every task and message, is given by the fixed point of the following equation.

$$\begin{array}{ll} J^{(0)} & = \{0\, \forall i \in 1..n\} \\ WCRT(i)^{(1)} & = WCRT(i, J^{(0)}) \\ J^{(m+1)} & = \{\max_{j \text{ precedes } i} \left(WCRT(j)^{(m)}\right)\ \forall i \in 1..n\} \\ WCRT(i)^{(m+1)} & = WCRT(i, J^{(m)}) \end{array} \tag{7.1}$$

where $J^{(m)}$ is the set of the worst-case release jitters of the tasks and messages at the step n of the fixed-point lookup (it can be seen as a vector of n jitters). The function $WCRT(i, J^{(m)})$ depends on the worst-case release jitters of the tasks and messages, on the local scheduling for a task, and on the network type for a message. The sequel of the chapter gives the formulas used to compute this function in different contexts, for CPU and network scheduling.

The convergence of the fixed-point lookup is dependent on the $WCRT$ functions: they must be monotonically increasing functions of the release jitters. Because of this required property, a holistic analysis is a worst-case analysis, giving an upper bound of the WCRT of the tasks and messages of the system. This method can be pessimistic (see [BAT 98] for an illustration) and can be improved by reducing the release jitter in calculating the best-case response time [HEN 01].

7.4. WCRT of the tasks

The previous section showed that in order to study the feasibility of a distributed system, we need to compute the WCRT of tasks with a release jitter. The release jitter is accounting for the worst-case arrival time of the messages. Depending on the context (scheduling algorithm, task activation scenarios), several methods exist. We focus here on two commonly used scheduling algorithms: the fixed-priority policies (see section 7.4.3), and the deadline-driven scheduling algorithm (Earliest Deadline First) in section 7.4.4. Regarding the tasks, we focus on event-driven tasks, which represent a worst-case for time-driven tasks (i.e. the WCRT of a task assuming it is event-driven is a WCRT for the same task if it was time-driven). In the chosen contexts, the WCRT problem is co-NP-hard in the weak sense for the case of synchronous periodic task systems with a relative deadline shorter than the period [EIS 08, EIS 10] (i.e. there are pseudo-polynomial time algorithms solving the problem).

7.4.1. *Scheduling algorithms*

A schedule is feasible if the WCRT of every job of every task is smaller than the task's relative deadline, in other words if all the deadlines are met during the life of the system. Most researchers propose scheduling policies or feasibility tests, or consider some quality of service or cope with more complex task models. Except for basic problems, the feasibility problem is NP-hard, thus there are two choices: being exact at an exponential cost, or being pessimistic at a polynomial or pseudo-polynomial cost. Of course, it is better not to be optimistic when talking about feasibility. There are two families of scheduling techniques for asynchronous real-time systems:

– on-line scheduling (or priority-driven scheduling): during the execution of the system, a simple scheduling policy is used by the RTOS in order to choose the highest priority job in the set of ready jobs. This algorithm (usually called policy) is used when the executed job is finished or when it is blocked, or when another job is released, or even sometimes at every time unit (quantum based scheduling). In this case, the literature proposes efficient schedulability tests (polynomials or pseudo-polynomials) that can be used off-line (i.e. in order to validate a scheduling policy for a system before its actual execution). As soon as some non-preemptive parts are involved in the system, there is no optimal online scheduling algorithm [MOK 83]. The same kind of negative result can be found in scheduling theory for tasks including some practical factors, like tasks with self-suspension [RID 04]. A scheduling algorithm A is optimal if for any task system, either it is not feasible with any scheduling algorithm or it is feasible with the algorithm A;

– off-line scheduling (or time-driven scheduling) techniques, using model-based or branch and bound or meta-heuristic algorithms, create a feasible schedule that can be executed endlessly by a dispatcher. In this case, techniques deal with the state explosion problem. A major drawback of off-line scheduling is that a schedule has to be computed every time the system is modified during the coding stage.

In this chapter, we deal with on-line scheduling algorithms. They are flexible, and widely used in industrial processes. The WCRT analysis depends on the nature of the scheduling algorithm:

– fixed-priority policies (FPP) assign a fixed priority to the tasks of the system. The scheduler assigns the highest priority task to the processor. The designer has to choose the priority to assign to the tasks. Rate monotonic is optimal in the class of FPP [SER 72, LIU 73] (optimality restrained to the comparison with other FPP algorithms) for independent (fully preemptive) periodic tasks with an implicit deadline (a task has to be executed before its next release). It assigns a higher priority to the tasks with a shorter period. The most widely used policy is deadline monotonic, which is an adaptation of rate monotonic to the case where the deadline of a task can be shorter to its period: Deadline Monotonic gives a higher priority to the tasks with a higher relative urgency (i.e. modeled as relative deadline), and is optimal in the class of FPP for independent periodic tasks with a deadline prior to their period [LEU 82]. Audsley's algorithm [AUD 91] for priority assignment is optimal when some tasks are never released simultaneously (i.e. for some time-driven task systems), or when the relative deadline can be larger than the periods; this algorithm has an exponential time complexity;

– dynamic priority scheduling algorithms compute the priorities of the task online. The most interesting algorithm of this family is the deadline-driven scheduling algorithm [JAC 55, DER 74, LIU 73], also called earliest deadline first (EDF), which assigns a priority to a task depending on its urgency: the closer its deadline, the higher its priority. This algorithm is optimal for independent task systems;

– in the case of a multiprocessor, a recent family of algorithms, called PFair, has been investigated [DAV 09]. Some algorithms of this family are optimal for multiprocessor systems. In this chapter, we assume the distributed system to be compounded of uniprocessor systems, so we do not describe WCRT analysis, which is still an open problem for non-trivial cases in the multiprocessor case.

7.4.2. *Modeling the tasks*

Most of the treatments of a command and control system are periodic, and are run for a potentially infinite duration. Therefore, the common way to represent a task, denoted τ_i, is based on the Liu and Layland [LIU 73] periodic task model:

– the WCET C_i is the worst possible computing time of a task τ_i (see [WIL 08] for specific information about the WCET). It should be noted that, in general, the actual computation time of the tasks is variable between 0 and the WCET;

– the smallest task period T_i is the smallest interval separating two successive activations of τ_i, we note $\tau_{i,j}$ the j^{th} job of the task τ_i;

– the relative deadline D_i represents the temporal constraint of τ_i, i.e. if the job $\tau_{i,j}$ is released at the time t, then it must be finished at the time $t + D_i$ (note that in the original model of [LIU 73], the deadlines are implicit: $D_i = T_i$), the deadlines can be arbitrary (i.e. we can have $D_i > T_i$);

– the release jitter J_i represents the worst-case duration between the release of a task and the time it is sent into the ready queue. This parameter is accounted for the incertitude about the availability of an expected message.

If the system is time-driven (see 7.2.2.1), we say that the tasks are concrete. The release date r_i of the first job $\tau_{i,1}$ is known; moreover, in this case, the tasks are strictly periodic, and the subsequent releases of the jobs are known, as well as the corresponding deadlines:

$$r_{i,j} = r_i + (j-1)T_i$$
$$d_{i,j} = r_{i,j} + D_i$$

A job $\tau_{i,j}$ can be ready at any time in the interval:

$$[r_{i,j}, r_{i,j} + J_i]$$

It is important to keep in mind that the release time $r_{i,j}$ and the ready time are different when dealing with release jitters. Figure 7.5 represents a time-driven task: the up arrows represent job activations (i.e. instants when the task becomes ready in the system), while the down arrows represent job deadlines.

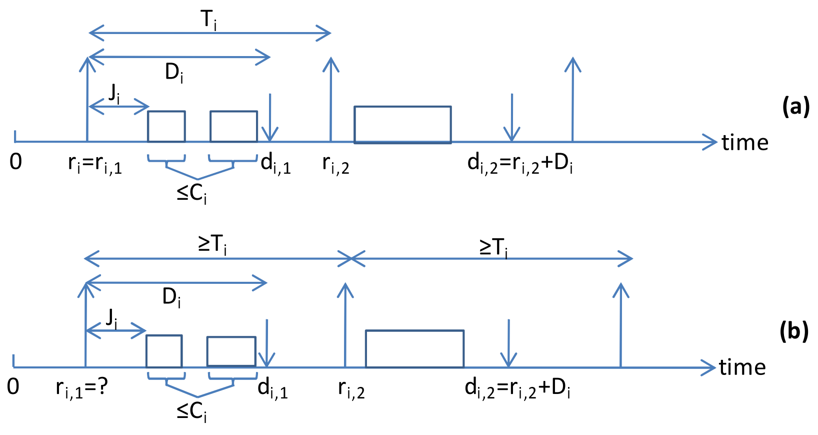


Figure 7.5. *a) A time-driven (concrete) task, b) an event-driven (non-concrete) task*

If the system is event-driven, then the tasks are said non-concrete (i.e. their release dates are unknown). Usually for this task model, the tasks are event-driven, and their actual period during system execution can vary between their period and $+\infty$ (they are called sporadic in the literature).

Giving these possible task models, the scheduling theory focuses on how to define the worst-case scenario for a task (the conditions issuing the worst possible response time). This scenario depends on the scheduling algorithm.

7.4.3. *WCRT for fixed priority scheduling*

We have seen that the delay introduced by the messages is taken into account with release jitter. We first show how to calculate the WCRT for tasks without jitter, and will show how jitters are taken into account.

The most widely used scheduling policy is based on fixed priorities. These scheduling algorithms are called fixed priority policies (FPP). The reason can be that most commercial off-the-shelf RTOS offer only FPP (even if the deadline-driven scheduling algorithm appeared in an annex of the Ada 2005 standard). The most important concepts to understand are the critical instant concept (for non-concrete systems and synchronous concrete systems) presented in section 7.4.3.1 and the busy period concept introduced in section 7.4.3.2. Section 7.4.3.3 shows how to establish the WCRT without taking release jitters into account, then section 7.4.3.4 shows how to integrate jitters into the formulas. Finally section 7.4.3.5 shows how to integrate practical factors such as mutual exclusions into the WCRT calculation.

7.4.3.1. *Critical instant*

As the durations, and for the sporadic tasks the periods, and for non concrete systems the first release date, may all vary, it is important to study the worst-case behavior of the tasks. The worst-case scenario for a task is called a critical instant.

THEOREM 7.2 *Critical instant theorem [LIU 73, BAR 90]: for independent task systems, where the tasks are non-concrete or where the tasks are concrete and synchronous (i.e. for every task τ_i, $r_i = 0$), the critical instant for a task τ_i, leading to its WCRT, occurs when τ_i is released simultaneously with all the higher priority tasks.*

Figure 7.6 illustrates this theorem, when tasks are subject to release jitters [TIN 94a]: the WCRT of τ_2 occurs when it is released at the same time as τ_1. A task is delayed by higher priority tasks releases. It should be noted that this theorem concerns only independent task systems. In section 7.4.3.5, we shall describe how to take mutual exclusion into account.

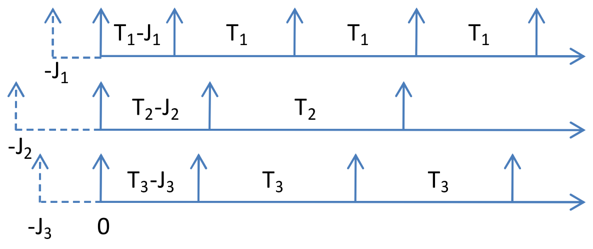


Figure 7.6. *Illustration of the critical instant*

7.4.3.2. *Busy period*

A level-i busy period is a time period where the processor is kept busy by tasks with a priority higher than or equal to $priority(\tau_i)$, where there is no idle point. An idle point corresponds to a point where the $Time\ Demand\ Function$ meets the $Time\ line$ (it corresponds to a point where all the previous requests of this priority level have been completed). Figure 7.7 shows the "classic view" of a busy period: initially, τ_1 and τ_2 are released; therefore, the processor has to compute $C_1 + C_2$ (with $C1 = 1$ and $C2 = 2$) time units, and each time a task is released ($T_1 = 4$ and $T2 = 14$), the processor demand increases. The processing power is given on the diagonal: the CPU can process one time unit of work per time unit. When the time demand function crosses the time (line $Time\ demand = Time$), it is the end of a busy period. When there is no demand, the CPU remains idle until the next release, which is the beginning of the next busy period. It is important not to confuse the end of a busy period (idle slot) and idle time: in Figure 7.7, the first idle slot takes place in the time interval $[27, 28]$, but the end of the busy period occurs at time $t = 14$ (an idle point is enough to end a busy period).

THEOREM 7.3 *[AUD 91, AUD 93] The WCRT for a task i occurs during the longest level-i busy period. The longest busy period is initiated by the critical instant.*

Therefore, we know the worst-case for a task τ_i for the case of non-concrete task systems and concrete synchronous task systems: we consider the critical instant, build the first level-i busy period, study all the jobs of τ_i occurring in the busy period, and claim the WCRT of these jobs as the WCRT of τ_i. A busy period ends if, and only if, for a system of n tasks, the processor utilization ratio $U = \sum_{i=1}^{n} Ci/Ti \leq 1$, which is a trivial but necessary schedulability condition on a single processor.

7.4.3.3. *Worst-case response time without release jitter*

7.4.3.3.1. Case of constrained deadlines ($D_i \leq T_i$)

[JOS 86] gives an exact pseudo-polynomial test when only one job of a task can be found in busy period (the authors suppose that $D_i \leq T_i$, thus if 2 jobs of a task τ_i are in the busy period, the system is not feasible with the chosen FPP). This test is studying the tasks one by one. The exact duration of the busy period, if it is smaller than the task's period T_i, is given by the smallest fixed point of the equation:

$$\begin{aligned} R_i^{(0)} &= C_i \\ R_i^{(n+1)} &= C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(n)}}{T_j} \right\rceil C_j \end{aligned} \tag{7.2}$$

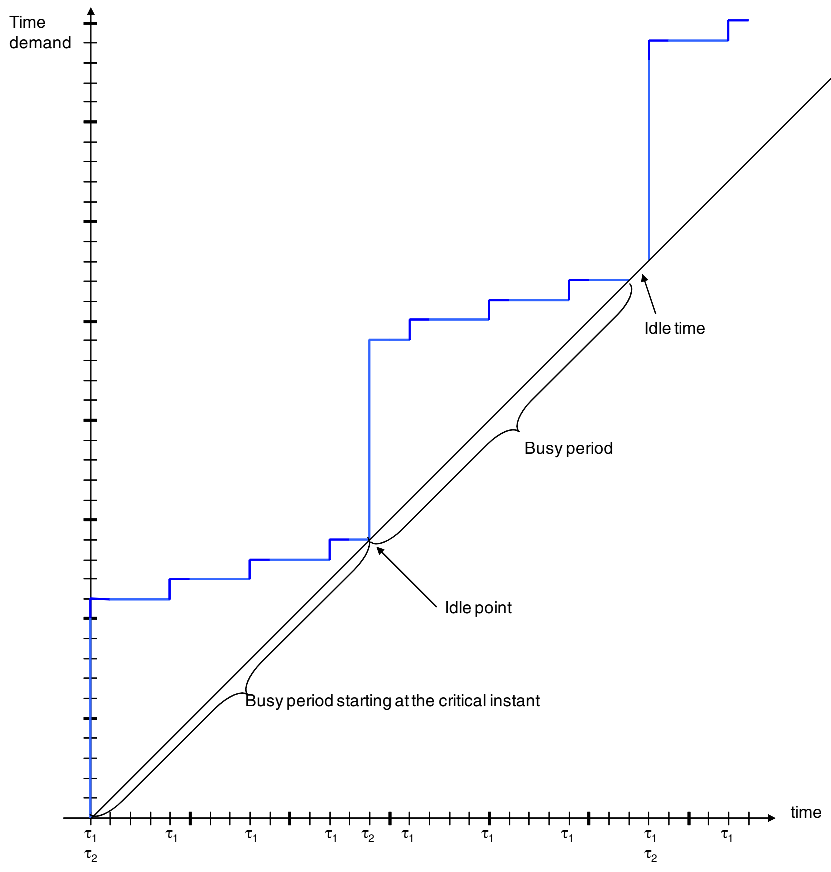


Figure 7.7. *Illustration of a busy period*

with $hp(i)$ the set of indexes of higher priority tasks than τ_i. The part of the equation corresponding to the function $W_i(t) = \sum_{i \in hp(i)} \lceil t/T_i \rceil C_i$ is called the processor demand function of priority level i: it represents the CPU time requested by tasks with a priority greater or equal to $priority(i)$ in the interval $[0, t)$ (an interval $[a, b)$ is closed on the left, opened on the right). Using this notation, $R_i^{(*)}$ is the smallest fixed-point of the equation $t = C_i + W_i(t)$.

This equation consists of taking C_i as the shortest possible busy period $R_i^{(0)}$. For the next step, we consider that the higher priority jobs released in the interval $[0..R_i^{(0)})$ will increase the busy period by their WCET. We carry on until all the jobs in the busy period have been taken into account. If $R_i^{(*)} \leq T_i$ (thus τ_i does not occur more than once in the busy period), following the critical instant theorem, and theorem 7.3, we can conclude that $R_i^{(*)}$ is the longest busy period, and that the WCRT of τ_i occurs in this busy period. As its release time r_i was assumed to be 0, the WCRT of τ_i is $R_i^{(*)}$.

What is really interesting in this test is the fact that the priority order of higher priority jobs has no influence on the response time of τ_i.

7.4.3.3.2. Case of arbitrary deadlines

Nevertheless, if $R_i^{(*)}$ is greater than T_i, the busy period is not over, as τ_i is released at least a second time. We thus have to continue with the test taking the following jobs of τ_i into account. This is exactly what is proposed in [LEH 90, LEH]: k represents the number of occurrences of τ_i in the busy period. Starting with $k = 1$ (obtaining exactly the same test as in equation (7.2)):

$$\begin{aligned} R_i^{(0)}(k) &= kC_i \\ R_i^{(n+1)}(k) &= kC_i + \textstyle\sum_{j \in hp(i)} \left\lceil \frac{R_i^{(n)}}{T_j} \right\rceil C_j \end{aligned} \tag{7.3}$$

If $R_i^{(*)}(1) > T_i$, then the busy period initiated by the critical instant contains at least two occurrences of τ_i, therefore, the test has to be carried out for $k = 2$. If $R_i^{(*)}(2) > 2T_i$, we have to carry on for $k = 3$ and so on until $R_i^{(*)}(k) \leq kT_i$. The WCRT of τ_i is found in this busy period, but it is not necessarily the first job's response-time. The response time of the job $\tau_{i,k}$ (k starting at 1) is $R_i^{(*)}(k) - (k-1)T_i$ (date of its termination minus date of its release).

We can remark that an accurate choice of the initial value $R_i^{(n)}(k)$ can reduce the number of iterations needed to reach the fixed-point [DAV 08, SJO 98, BRI 03].

Let K be the smallest integer such that $R_i^{(*)}(K) \leq KT_i$, then the WCRT of a task τ_i is given in the general case by:

$$WCRT(\tau_i) = \max_{k=1..K} (R_i^{(*)}(k) - (k-1)T_i)$$

7.4.3.4. *WCRT with release jitter*

As the network communications are taken into account by a release jitter for the tasks, we now see how to take them into account in the WCRT computation. First, we need to establish the worst-case scenario for a task under analysis.

THEOREM 7.4 *Critical instant theorem [TIN 94a]: for independent task systems, where the tasks are non-concrete or where the tasks are concrete and synchronous (i.e. for every task τ_i, $r_i = 0$), the critical instant for a task τ_i, leading to its WCRT, occurs when τ_i is released, after suffering its maximal jitter, simultaneously with all the higher priority tasks, which are delayed by their maximal jitter.*

In order to maximize the interference of higher priority tasks on a task under analysis, we need to consider the time instant where the higher priority tasks are about to have the largest amount of requests. Moreover, the worst-case for the task under analysis occurs when it is delayed by its maximum jitter, as the time remaining between its arrival in the ready queue and its deadline is minimal. Figure 7.8 illustrates this theorem: the interference of a task τ_i on lower priority tasks can be seen as a stair function, where each stair has a height of the task WCET C_i, and the delay between two successive stairs is the task period T_i. To maximize the interference of τ_i when τ_i has a release jitter J_i, we consider that τ_i is initally released at the date $-J_i$: this is the release date maximizing the height of the stair function (i.e. the interference) at any time t. In Figure 7.8 the dashed line interference function is always higher than any other possible interference function.

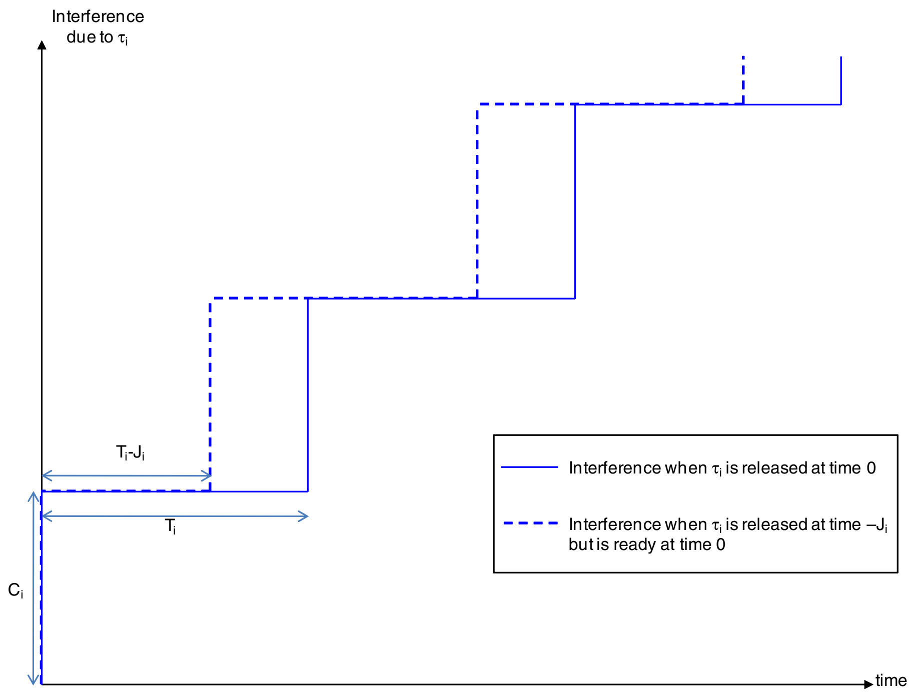


Figure 7.8. *Illustration of the critical instant with jitter*

If we set the time origin at the critical instant, every task τ_j is released at the date $-J_j$. In such a configuration, the worst-case interference of a higher priority task τ_j in the time window $[0, t)$ is $\left\lceil \frac{J_j+t}{T_j} \right\rceil C_j$. Thus, the longest level-i busy period is given by the smallest fixed-point of the equation:

$$\begin{aligned} R_i^{(0)}(k) &= kC_i \\ R_i^{(n+1)}(k) &= kC_i + \sum_{j \in hp(i)} \left\lceil \frac{J_j + R_i^{(n)}}{T_j} \right\rceil C_j \end{aligned} \tag{7.4}$$

Note that, since the origin of time is taking place J_i time units after the release of the task under analysis τ_i, if $R_i^{(*)}(k) > kT_i - J_i$, then the next job of τ_i is part of the same busy period and has to be studied. Let K be the smallest integer value such that $R_i^{(*)}(K) \leq KT_i - J_i$, then the WCRT of τ_i is given by:

$$WCRT(\tau_i) = \max_{k=1..K} (R_i^{(*)}(k) + J_i - (k-1)T_i) \tag{7.5}$$

It is important to realize that this bound is exact in the case of sporadic tasks, but that it is an upper bound for concrete synchronous task systems because the critical instant is not always realistic in this case. In fact, forbidding the occurrence of critical instant can be interesting in order to increase the schedulability of a system that would otherwise not be feasible. There are two main problems: choosing the right release dates to avoid the critical instant (offset free systems), and the WCRT analysis. Even without release jitter, this is a difficult problem. For example, if two tasks τ_i and τ_j should never be released at the same time, there are $gcd(T_i, T_j) - 1$ (where gcd is the greatest common divisor) possible integer values for their relative offset [GOO 03]. Then choosing the release times wisely may improve schedulability; moreover, [AUD 91] proposes an optimal priority assignment for such systems by testing $O(n^2)$ priorities (n being the number of tasks). Nevertheless, testing the feasibility of a priority assignment for asynchronous independent task systems is NP-hard [LEU 82, BAR 90].

Therefore, in this case, we have to study every busy period contained in a simulation duration. Depending on the task system, the length of the simulation duration is going to be, at least $lcm(T_i)$, and at most up to $\max(r_i) + 2 \times lcm(T_i)$ [LEU 80, CHO 04] where lcm is the least common multiple.

7.4.3.5. *Critical sections*

Except for deadlock potential problems, mutual exclusion introduces new problems in real-time scheduling: scheduling anomalies and priority inversion. A scheduling anomaly occurs, for example, when reducing a task duration compared to its WCET is increasing a WCRT (see Figure 7.3). Therefore, even if a system employing a scheduling algorithm seems to be feasible during simulation, it may not be feasible for any possible duration of the tasks (remark: it would be feasible if we were using an off-line schedule in a dispatcher). As soon as non-preemptive parts are involved (and a critical section implies non-preemptivity for some parts of the tasks), then scheduling anomalies can occur.

Moreover, a specific phenomenon can occur in priority-driven systems: the priority inversion. A priority inversion occurs when a low priority task reatins a resource. When a high-priority task is released, and requires the same resource, the high-priority task is blocked until the end of the low-priority task's critical section. In the meantime, any release of an intermediate-priority task can delay the low-priority task; therefore, the high-priority task is delayed while waiting for the end of the critical section.

An intuitive way to avoid the priority inversion is to use the Priority Inheritance Protocol (PIP) [SHA 90]: a task retains a resource that is blocking a higher priority task inherits the higher priority task's priority until it frees the resource. The PIP avoids any priority inversion, but it does not reduce the number of times a task can be blocked when trying to enter a critical section. Studying a graph of resource usage, we can compute how many resources can block a job for a system, and how long the longest critical section would be. We can deduce a blocking factor B_i of a job. Note that during a level-i busy period, a task can be blocked at most once (when some low-level priority task commenced a critical section before the release of the task under analysis); thus, the WCRT of a task τ_i is found in the longest level-i busy period computed with:

$$\begin{aligned} R_i^{(0)}(k) &= B_i + kC_i \\ R_i^{(n+1)}(k) &= B_i + kC_i + \textstyle\sum_{j \in hp(i)} \left\lceil \frac{J_j + R_i^{(n)}}{T_j} \right\rceil C_j \end{aligned} \tag{7.6}$$

In this formula, we assume the worst-case scenario as being an instant when all the higher priority jobs are released at the critical instant, while all the lower priority jobs have just started their longest critical section, implying the longest blocking time. Note that when using this protocol, a task can be delayed by a lower priority task even if it is not sharing a resource with it. This is called indirect blocking and is due to priority inheritance of a lower priority task up to a higher priority than the priority of the task under analysis.

Sha *et al.* [SHA 90] use PIP as an intuitive protocol but they show its inefficiency compared to the Priority Ceiling Protocol (PCP). In PCP each resource R has a ceiling Π_R, defined as the highest priority among the tasks using it. The system ceiling is defined as $\Pi_S = \max_{\forall\ resource\ in\ use\ R}(\Pi_R)$. The protocol operates exactly like the PIP, with an additional resource access rule: a task can access a resource if its priority is strictly higher than the system ceiling or if it is itself the cause of the value of the system ceiling. PCP avoids any priority inversion (like PIP), moreover, a task can be blocked only once per busy period, even if it is using several resources. A blocking time cannot exceed the length of one critical section. This is due to the rule introduced by PCP: if there is a critical section using a resource R_1 required by a task τ_i (thus, $\Pi_{R_1} \geq priority(\tau_i)$ and $\Pi_S \geq R_1$), then no other task can enter in critical section unless its priority is strictly greater than the priority of τ_i (because $\Pi_S \geq priority(\tau_i)$). An interesting side effect of PCP is that no deadlock can occur.

While PIP cannot be implemented efficiently and has a poor behavior regarding the value of B_i, PCP can be implemented efficiently in its immediate version (having the behavior of the super priority protocol proposed in [KAI 82]). The exact same worst-case behavior takes place when the inheritance occurs as soon as a task enters in a critical section. As a result, Immediate PCP is the most widely used protocol in commercial off-the-shelf real-time executives (e.g. POSIX, OSEK/VDX, Ada standards). The difference between the existing resource access protocols is the way in which the blocking factor of a task B_i is computed. The longest busy period is computed by equation (7.6).

Non-preemptible tasks are a particular case of task sharing resources (we can consider that all the tasks share the same resource); thus scheduling anomalies can also occur (even if, of course, priority inversion cannot occur). Validating a non-preemptible task system is NP-hard [LEN 77, JEF 91]. However, their behavior is closer to the non-preemptible critical section [MOK 83].

7.4.4. *WCRT analysis for deadline-driven scheduling*

The other well-known on-line scheduling algorithm is the deadline-driven (also called EDF) policy. It is considered to be a dynamic priority policy; contrary to fixed-priority policies. For EDF, the closer the task's deadline, the higher its priority. This scheduling algorithm is not yet available in many commercial off-the-shelf RTOS for several reasons such as the domino effect (when a task misses its deadline, several subsequent tasks are prone to miss their deadline also), and the overhead introduced by the dynamic attribution of the priorities. Nevertheless, this algorithm is optimal for independent task systems. [LIU 73] shows that for independent tasks with an implicit deadline ($D_i = T_i$), a system of n tasks is feasible with EDF if, and only if, $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$; therefore, it is an optimal algorithm in this context. In other words, if a system is feasible with any scheduling algorithm, then a schedule produced by EDF is feasible.

In order to conduct a holistic analysis, we need to show how to compute the WCRT of a task with release jitter. The question is how to find a scenario leading to the WCRT for a task under analysis.

7.4.4.1. *Busy period*

If the tasks are concrete and synchronous, the longest busy period is initiated by a synchronous release of the tasks [JEF 93]. The difference, compared to the level-i busy period computed for the WCRT of the tasks scheduled by a FPP, is that the busy period here takes all the tasks into account at once (it is equivalent to the level-n busy period in FPP, n being the lowest priority level). EDF, like the FPP algorithms, is a conservative algorithm (i.e. the processor is never left idle if there are ready tasks). As a consequence, the processor is idle at the exact same moments for EDF as for any FPP scheduling algorithm. So, similar to FPP, theorem 7.3 holds true for the longest busy period: the longest busy period is found after a critical instant initiated by a synchronous release of all the tasks (when release jitter is present in the system, the critical instant is taken as a synchronous arrival of all the tasks in the ready queue, after experiencing their maximum release jitter). Therefore, for a system of n tasks, the longest busy period, denoted L is obtained as the fixed point of the equation:

$$\begin{aligned} L^{(0)} &= \sum_{i=1}^{n} C_i \\ L^{(n+1)} &= \sum_{i=1}^{n} \left\lceil \frac{J_j + L^{(n)}}{T_j} \right\rceil C_i \end{aligned} \tag{7.7}$$

Regarding the set of deadlines that will occur in the busy period, the critical instant corresponding to a synchronous release of the tasks is a critical instant for the system. Nevertheless, the critical instant of the system does not correspond to the critical instant for a task under analysis:

THEOREM 7.5 *[SPU 96a]: the WCRT of a task occurs in a busy period initiated by a synchronous release of all the other tasks.*

This theorem means that while the other tasks are synchronously released, the release of the task under analysis has to vary in order to identify its WCRT. As the priorities are a function of the next deadline, the fact that the deadline of a job of the task under analysis occurs before or after the deadline of another job, depending on the difference of release dates of the tasks, will give a higher or a lower priority to the job of the task under analysis compared to the other jobs. We set the time origin as the simultaneous arrival of the tasks, other than the task τ_i under analysis. So the release occurs at the date $-J_j$ for every task $\tau_j, j \neq i$. Note that as J_j can be greater than T_j, several jobs of τ_j can arrive in the ready queue at the time origin. We assume that one of the release times of the analyzed task τ_i happens at a date a (we will see later how to accurately choose the set of the test dates a in order to compute the WCRT), and we first describe how to compute the WCRT of τ_i when it is released at this date. This computation is based on the deadline-d busy period.

DEFINITION 7.6 *[GEO 96] The deadline-d busy period of a deadline d is a time interval where the processor is executing jobs with deadlines that are less than or equal to d.*

If we are studying the job τ_{i,r_a}, which is released at the date a, we can see that its first deadline occurs at the date $a + D_i$ (we always consider that ties are unfavorable to the task under analysis), and the only jobs able to delay τ_{i,r_a} are the jobs with a deadline occurring before or on the date $a + D_i$ situated in the same deadline-d busy period because EDF gives a higher priority to urgent jobs. These jobs are the jobs interfering with τ_{i,r_a}, and can either be due to other tasks than τ_i, or be jobs from the task τ_i released in its previous periods. In order to compute this interference, we need a function that calculates the processor required by the jobs of a higher priority than the job with the deadline $a + D_i$ in a time interval $[0, t)$.

$$Interf(a,t) = \sum_{\substack{j \neq i \\ D_j \leq a+J_j+D_i}} \min\left(\left\lceil \frac{t+J_j}{T_j} \right\rceil, 1 + \left\lfloor \frac{a+J_j+D_i-D_j}{T_j} \right\rfloor\right) \times C_j \tag{7.8}$$

This interference represents the maximal interference applied to τ_{i,r_a} by other tasks than τ_i, taking into account the jobs having a deadline prior to $a + D_i$. Note that we assume a critical instant where each task $\tau_{j,j \neq i}$ is put in the ready queue at the time origin, implying that its first release time occurs at the date $-J_j$. Therefore, in order for at least one job to be included in the deadline-d busy period of τ_{i,r_a}, the task has to have at least one job included, therefore we must have $D_j \leq a + J_j + D_i$. For every task τ_j having at least one job with a deadline not later than $a+D_i$, the $Interference$ has to take into account the amount of processor requested jobs able to interfere with τ_{i,r_a} in $[0, t)$. It cannot be higher than the request implied by the releases of τ_j in the interval $[-J_j, t)$, therefore it cannot be higher than $C_j \times \left\lceil \frac{t+J_j}{T_j} \right\rceil$ (the first argument of the min function), and it cannot be higher than C_j times the number of deadlines encountered in the interval $[0, a + D_i]$ which is given by the second argument of the min function.

The size of the deadline-d busy period will be obtained as the fixed-point of an equation, taking into account the interference of other tasks (see equation (7.8)), and the interference of the task under analysis itself. As $a \geq 0$, several jobs of τ_i can take place before the date a. Any job of τ_i released in the interval $[-J_i, a]$ can be part of the deadline-d busy period of τ_{i,r_a}. The release date of the first job of τ_i able to take part in the busy period occurs at the date $s_i(a)$ where:

$$s_i(a) = a + J_i - \left\lfloor \frac{a + J_i}{T_i} \right\rfloor T_i \tag{7.9}$$

The self-interference introduced by τ_i in the deadline-d busy period cannot be greater than the requests in $[s_i(a), t]$, and cannot be greater than the requests having a deadline prior to $a + D_i$. Therefore, it is given by the function of time:

$$Self(a,t) = \begin{cases} 0 & \text{if } t \leq s_i(a) \\ \min\left(\left\lceil \frac{t - s_i(a) + J_i}{T_i} \right\rceil, 1 + \left\lfloor \frac{a + J_i}{T_i} \right\rfloor\right) \times C_i & \text{else} \end{cases} \tag{7.10}$$

Summing up these two sources of workload in a deadline-d busy period for τ_{i,r_a}, we obtain the workload function giving the work that needs to be processed with deadlines that occur before or at $a + D_i$ in an interval $[0, t)$:

$$Workload(a,t) = Interf(a,t) + Self(a,t) \tag{7.11}$$

The calculation of the length of the deadline-d busy period of τ_{i,r_a} is the smallest fixed-point of the equation:

$$I_i = 1 \text{ if } s_i(a) = 0, \ 0 \text{ else} \tag{7.12}$$

$$\begin{cases} L_i^{(0)}(a) = & I_i \times C_i + \sum_{D_j \leq a + D_i + J_j}^{i \neq j} C_j \\ L_i^{(n+1)}(a) = & Workload(a, L_i^{(n)}(a)) \end{cases} \tag{7.13}$$

$L_i^{(*)}(a)$ gives the length of the deadline-d busy period containing only a workload with a priority not lower than the priority of τ_{i,r_a}; therefore, this is the WCRT of τ_i when a job is released at the date a. As in the case where τ_i is the only task in the busy period, equation (7.12) gives 0, the corresponding WCRT is:

$$WCRT(a) = \max(L_i^{(*)}(a) - a, C_i + J_i) \tag{7.14}$$

7.4.4.2. *WCRT*

Now, giving a release date a of a job under study, we have a method by which we can find the corresponding WCRT. The question that now arises concerns the different release dates to test for τ_i to calculate its WCRT. The release dates tested are part of the interval $[-J_i, L - J_i - C_i]$ (L is the longest busy period of the system, given by the smallest fixed point of equation (7.7)). However, we cannot test every date if we consider non-integer values for the temporal parameters. Moreover, this interval has an exponential size of the task system parameters. Nevertheless, [SPU 96b] notes that the local maxima of the function $WCRT(a)$ occur when the deadline of τ_{i,r_a} coincides with another task's deadline, or when $s_i(a) = 0$. We then obtain the set A of all the release dates to test for τ_i:

$$\begin{aligned} A = \ & \textstyle\bigcup_{\forall\ task\ \tau_j} \{a | a + D_i = kT_j - J_j + D_j, \\ & k \in \mathbb{N} \text{ and } a \in [-J_i..L - J_i - C_i]\} \end{aligned} \tag{7.15}$$

The WCRT of a task τ_i is thus the maximum of the WCRT obtained for every local maximum of $WCRT(a)$:

$$WCRT(\tau_i) = \max_{a \in A}(WCRT(a)) \text{ applied on } \tau_i \tag{7.16}$$

7.4.4.3. *Critical sections*

As in the case of FPP scheduling policies, the EDF scheduling algorithm has been modified to avoid priority inversion and to limit the blocking duration of a task due to a resource access or a (dynamic) priority inheritance [CHE 90, BAK 91]. If a resource management protocol is used, such as in FPP, a task can be delayed by a lower priority task once per deadline-d busy period by a blocking factor B_i. In this case, equation (7.12) is modified as follows:

$$I_i = 1 \text{ if } s_i(a) = 0,\ 0 \text{ else} \tag{7.17}$$

$$\begin{cases} L_i^{(0)}(a) = & I_i \times C_i + \sum_{D_j \leq a + D_i + J_j}^{i \neq j} C_j \\ L_i^{(n+1)}(a) = & Workload(a, L_i^{(n)}(a)) + B_i \end{cases} \tag{7.18}$$

Moreover, equation (7.14) is modified as follows:

$$WCRT(a) = \max(L_i^{(*)}(a) - a, C_i + J_i + B_i) \tag{7.19}$$

7.5. WCRT of the messages

This section presents two typical WCRT message analyses. Section 7.5.1 introduces the analysis of a shared bus: a controlled area network (CAN) network, using the analysis proposed in [DAV 07], which corrects [TIN 94c, TIN 95b]. Then, 7.5.2 presents the analysis of a commuted network, using the example of WCRT analysis of an ATM network [ERM 97]. The reader will find response time analysis for other kinds of deterministic networks in:

– a simple timed token passing [TIN 95a];

– a real-time priority broadcast bus [TIN 95a];

– a timed token protocol [SPU 96b];

– FDDI [MAL 93, ZHA 95];

– ARINC 629 [AUD 97];

– AFDX (ARINC 664) [BAU 10];

– FIP [PED 97].

7.5.1. *CAN*

Regarding scheduling, the CAN can be seen as a non-preemptive computing resource, scheduled by a FPP, where messages are tasks. The propagation time of a message on the network is assimilated to the WCET of a task. If τ_{bit} is the propagation time of a single bit of the network, then the propagation time of a frame of n bytes ($n \leq 8$) is given by [DAV 07]:

$$C_f = \tau_{bit} \times \left(\left\lceil \frac{g + 8n - 1}{4} \right\rceil + g + 13 + 8n \right) \tag{7.20}$$

Where g is 34 for the standard format (11 bits identifier), or 54 for the extended format (29 bits identifier).

We consider a set of non-preemptive periodic messages, characterized by a propagation time seen as a WCET. Note that if the messages are sporadic, as in the sporadic task model, we consider the worst arrival pattern, which is the smallest inter-arrival time between two consecutive identical messages. We consider that there is no transmission error (or at least that the application is loosely fault-tolerant, e.g. an erroneous message is lost for the current period).

Scheduling non-preemptive messages (or tasks) with FPP is similar to scheduling preemptive tasks with a blocking factor due to resource sharing (see section 7.4.3.5). The critical instant for a message τ_i corresponds, without release jitter, to a synchronous emission of all the higher priority messages (release of all the higher priority tasks) with the message under analysis, when the longest lower priority message has just started to use the network [GEO 96]. In this case, the message under analysis has to suffer the interference of all the higher priority messages, plus the interference of the longest message with a lower priority as messages cannot be preempted.

Nevertheless, specificity occurs with the use of a network: the transmission delay. Indeed, assuming that the bus is idle and there is no higher priority message to send, we can start to emit the message, but if within the duration of the transmission of a bit τ_{bit}, another node did not receive the first bit of the message, and starts to emit a higher priority message, our message will have to be delayed. Therefore, we cannot just observe an interval $[0, bp]$ (where bp is the length of a busy period in a non-preemptive processor) to find out whether there is no higher priority message and whether the message under analysis can be sent; we have to observe an interval $[0, bp + \tau_{bit})$ in order to make sure that every node of the network received the first part of the message under analysis, and that they cannot start to send a message because they see that the bus is not idle.

When taking release jitters into account, the critical instant occurs for a message under analysis τ_i when all the higher priority messages are ready to be sent synchronously after suffering their worst release jitter, while τ_i was delayed by its worst release jitter before being ready to be sent. The release jitter will take into account the WCRT of the task sending the message. The message under analysis can send the first part after waiting for an interval $[-J_i, bp + \tau_{bit})$.

Let C_b be the longest duration of a lower priority message, then the longest level-i busy period during which the message under analysis $\tau_{i,k}$ might have to wait is the smallest fixed-point of the equation (starting with $k = 1$):

$$\begin{aligned} R_i^{(0)}(k) &= C_b + \sum_{j \in hp(i)} C_j \\ R_i^{(n+1)}(k) &= C_b + (k-1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{J_j + R_i^{(n)} + \tau_{bit}}{T_j} \right\rceil C_j \end{aligned} \tag{7.21}$$

Where $hp(i)$ is the set of the indexes of the higher priority messages. Giving K the smallest integer value such that $R_i^{(*)}(K) \leq C_i + KT_i - J_i$, the WCRT of a message is given by:

$$WCRT(\tau_i) = \max_{k=1..K} (C_i + R_i^{(*)}(k) + J_i - (k-1)T_i) \tag{7.22}$$

7.5.2. *Asynchronous transfer mode (ATM) network*

In an ATM network, a message follows a virtual circuit going from a source node to a destination node. The message is decomposed into cells (a cell has a 5-byte header and a 48-byte payload), and every cell uses the same virtual circuit. A virtual circuit defines the links carrying the cells between adjacent nodes. A node is a source node, a destination node, or an internal switch. On the output of the source node, and on every output of the switches, the cells waiting for emission on the output link are stored in a waiting queue. We assume that the priority-ordered queues use a first in first out (FIFO) within priorities order, i.e. cells are ordered by priority. Cells with the same priority are stored in FIFO order (a possible implementation uses a FIFO queue per priority level). Figure 7.9 shows a virtual circuit going from a source node to a destination node, passing through switches. A virtual circuit carries a stream of cells, and is characterized by a bit rate contract. In this section, we consider a constant bit rate contract based on the work of [ERM 97]. Note that the same authors generalized the technique to specific traffic shapes (for example, in [SJO 99], the authors consider multimedia messages using a multiframe model). A detailed version of the method presented in this section can be found in [SJO 00].

Figure 7.9 shows (in bold) a virtual circuit carrying a flow of cells $flow_i$ containing a message sent on a virtual circuit (we consider that this circuit is static, but the method could be used for dynamic virtual circuits, if the WCRT computation was conducted on-line) from a source node to a destination node. Each flow has a priority (which is not necessary unique), and on the source node, several messages can be transmitted

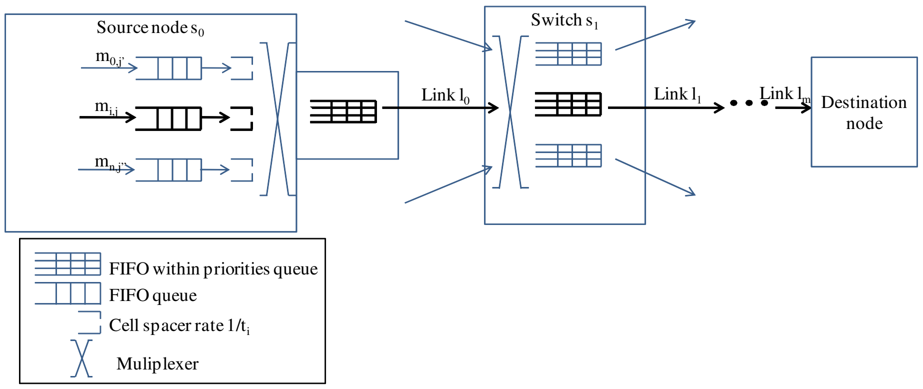


Figure 7.9. *Representation of a stream on an ATM network*

using the same link from the node to a switch. Thus, a priority queue is used at the output of the source node in order to sort the cells sent by the source node to the first switch. Each flow is shaped using a cell spacer: the role of a cell spacer is to guarantee that the bandwidth used by a flow conforms to its contract, by delaying every successive cell of a flow by a period, similar to a leaky bucket. The local task, executed on the source node, can send a whole message $m_{i,j}$ at once in the message queue, this message is decomposed into cells, which are shaped by the cell spacer before being sent into the output queue. The cells stored in the output queue are sent at the bit rate of the link. In a switch, an incoming cell is stored into the priority queue corresponding to the output of the virtual circuit, and sent to the next switch, until it reaches the destination node.

The WCRT of a message represents the longest duration from when the message is sent into the message queue, to when the last cell of the message arrives in the destination node. It depends on:

– the time spent in the source node before emission on the output link:
  - time spent by the last cell of the message into the cell spacer,
  - delay in the output queue;

– duration of the transmission of the cell over the output link;

– for every switch crossed:
  - time spent to send the cell to the right message queue of the switch,
  - delay in the output queue,
  - transmission duration on the output link.

In order to compute the worst-case delay in the priority queues, we need to know all the contacts concerning the flows of a higher or the same priority as the studied flow. Note that the lower priority flows do not have any impact on the response time of the message, so this method can be used to validate the real-time traffic of a general purpose ATM network, assuming that the real-time flows have a higher priority than the non-real-time flows.

Given the specificity of an ATM network, we need some specific additional notations for this section:

– N_i is the maximal amount of ATM cells used to carry a message $m_{i,j}$ of the studied flow $flow_i$. If a message is at most x bytes long, then $N_i = \lceil x/48 \rceil$ cells;

– T_i is, as in the other sections, the period of the task required to send the messages of $flow_i$ in the message queue, but here represents the arrival period of a message in the message queue. As it relies on a task, which is subject to local scheduling, J_i represents the jitter of the message (accounting for the WCRT of the task sending the messages);

– t_i is the cell spacer period, and represents the minimal delay between the transfer of a cell from the cell spacer to the output queue of the source node, we assume that $0 < t_i \leq T_i/N_i$;

– j_i is the network jitter of the stream $stream_i$. Initially 0, this parameter increases along the path of the stream. This parameter is crucial to the analysis, as we can see in Figure 7.8 that the interference due to periodic occurrences of a higher priority stream (task on the figure) is higher when a interference has jitter. If an interference has a period T and jitter J, the interference appears as if it was early by J;

– b_l is the bandwidth of a link l; C_l the time to send one cell on a link is given by $C_l = 424/b_l$.

We assume that the real-time traffic uses less than the whole bandwidth, i.e.:

$$\sum_{\forall stream\ stream_i} \sum_{\forall link\ l\ used\ by\ stream_i} \frac{N_i C_l}{T_i} < 1 \tag{7.23}$$

7.5.2.1. *Delay in the source node*

For every stream, we have to compute the earliest and the latest response time, in order to compute the jitter, which has an impact on the lower and same priority streams.

If when a message was sent in the message queue, the cell spacer was empty (i.e. all the previous messages have been completely sent), then the longest time spent by a cell would be given by the time spent by the last cell of a message of the studied stream. Nevertheless, because of the local scheduling of the task sending the message, a message has jitter J_i. So even if $t_i \leq T_i/N_i$ (i.e. there is enough time in a message period to send all the cells of a message), it is possible that the second message arrives "early" at the date $T_i - J_i$. There is no hypothesis concerning J_i, that might be arbitrary long; therefore, the message suffering the longest delay in the spacer (assuming that the first message of the stream $stream_i$ is $m_{i,0}$) is $m_{i,n}$ where:

$$n = \left\lceil \frac{t_i N_i + J_i}{T_i} - 1 \right\rceil$$

which is a message arriving in the message queue when there is the largest amount of cells still in the message queue, which belong to the previous messages, delayed by J_i. Therefore, the message $m_{i,n}$ suffers the longest delay in the cell spacer. We compute its earliest arrival time $E_{i,spacer}$ in the output queue (i.e. earliest passage through the cell spacer), as well as its latest arrival time $L_{i,spacer}$. These lower and upper bounds are used to compute the maximal jitter $j_{i,spacer}$ applied to $stream_i$ at the cell spacer.

$$\begin{aligned} E_{i,spacer} &= max(0, nT_i - J_i) \\ L_{i,spacer} &= ((n+1)N_i - 1)t_i \\ j_{i,spacer} &= L_{i,spacer} - E_{i,spacer} \end{aligned} \quad (7.24)$$

7.5.2.2. *Delay in a FIFO within priorities queue*

After passing the cell spacer, as well as in every switch of the path, a cell is stored in a FIFO within priorities queue. We thus need to know the worst possible amount of higher priority cells in the queue arriving before the cell is sent on the output link, as well as the amount of cells of same priority already in the ready queue when then cell arrives in the queue. [SJO 00] uses two functions to compute this information: $Arrival(i, k)$ computes the earliest arrival date of the cell k (the first cell of a stream is labeled by 0) of a stream $stream_i$ in the queue, and $Arrived(i, t)$ computes the maximal amount of cells belonging to $stream_i$ arrived before or at the time t.

$$\begin{aligned} Arrival(i, k) &= \max(0, \left\lfloor \tfrac{k}{N_i} \right\rfloor T_i - J_i - j_i + (k\%N_i)t_i) \\ Arrived(i, t) &= \left\lfloor \tfrac{t+J_i+j_i}{T_i} \right\rfloor N_i + \min(N_i, (t + J_i + j_i)\%t_i + 1) \end{aligned} \quad (7.25)$$

where $a\%b$ is the remainder of the integer division. The function $Arrived(i, t)$ gives the worst interference pattern of $stream_i$ in a switch when its jitter at this switch is j_i. Note that [SJO 00] proposes more complex formula, which are less pessimistic, for example, in a switch, $Arrival(i, k)$ cannot be higher than $Arrival(i, k-1)+C_l$ where

l is the link used by $stream_i$ to be sent to the switch. It is the same for $Arrived(i,t)$, which has to be less than $\left\lfloor \frac{t}{C_{in}} \right\rfloor + 1$ where C_{in} is the lowest bandwidth of the links used by $stream_i$ from its source node to the considered switch.

Using these functions, we can compute the worst-case queuing delay Q_i^k of a cell k belonging to a stream $stream_i$ in a FIFO within priorities queue (either on the source node, or in a switch).

$$\begin{array}{ll} Q_i^k & = Dequeue(i,k) - Arrival(i,k) \\ Dequeue(i,k) & = C_{out} \times (1 + k \\ & + \sum_{j \in sp(i)} Arrived(j, Arrival(i,k)) \\ & + \sum_{j \in hp(i)} Arrived(j, Dequeue(i,k))) \end{array} \tag{7.26}$$

where C_{out} is the time needed to send a message on the output link of the queue, $sp(i)$ (respectively $hp(i)$) is the set of streams of same (respectively higher) priority as $stream_i$. Note that $Dequeue(i,k)$ is obtained with a fix-point lookup, initialized by 0. The way $Dequeue$ is obtained relies on the fact that the worst interference on a cell is given by at most one lower priority cell that was just starting to be sent when the cell arrives in the queue, plus the preceding cells of the same stream, plus the maximum amount of same priority cells in the queue when the cell arrives, plus the interference due to the arrival of higher priority cells during the waiting time.

Finally, the worst-case queuing time for any cell of a stream is given by the maximum queuing time for any cell between the first cell and the cell number n (see equation (7.23)):

$$Q_i = \max_{k=0..n} Q_i^k \tag{7.27}$$

7.5.2.3. *Summing up to obtain the WCRT of a message*

The WCRT of a message is then obtained starting from the highest priority stream down to the lowest priority stream. For a $stream_i$, following a virtual circuit starting from the source node, passing through a link l_0 going to s_1, etc. outputting the cells to a link l_{m-1} to a switch s_m, linked by l_m to a destination node, the WCRT is given by:

$$WCRT(stream_i) = L_{i,spacer} + Q_i(source) + C_{l_0} + \sum_{p=1..m} \left(Q_i(s_p) + C_{l_p}\right)$$

where $Q_i(s_p)$ is the application of equation (7.27) to the queue of the switch or source node, and C_{l_p} is the transmission duration of a cell on its output link. Note that in the process, the jitter of every stream has to be updated for every encountered node on the virtual circuit associated to the stream:

$$\begin{aligned} j_i(s_0) &= 0 \\ j_i(s_p) &= j_i(s_{p-1}) + Q_i(s_{p-1}) \end{aligned} \qquad (7.28)$$

$j_i(s_p)$ is the jitter associated with switch s_p, and s_0 is the source node.

7.5.2.4. *Conclusion and remarks*

The WCRT calculation can be used either to validate a static system or to perform an on-line acceptation test: when negotiating a real-time stream, we can use this acceptance test in order to accept or reject a new stream. We accept a new real-time stream if it can meet its deadline, and does not make the other accepted real-time streams miss their respective deadline. This Call Admission Control (CAC) is more efficient than other classic methods [ERM 97]. For an on-line utilization of CAC, several optimizations are made on the formulas and on the way they can be applied efficiently [SJO 00].

The way the jitter is used at every FIFO within priorities queue is classic in this kind of switched network. We can note that the WCRTs are not always realistic, but they are safe, in the sense that they give an upper value of the actual WCRT of the messages. As a global analysis is not possible, the jitter computed at each switch is a way to account for the incertitude concerning the interference of a stream on other lower priority streams as well as on same-priority streams.

7.6. Case study

In order to illustrate the holistic analysis, we consider a numerical example of a real-time system distributed on three processors communicating on a CAN network, presented in Figure 7.10. On $Processor_A$ and $Processor_C$, we use a preemptive FPP algorithm. The priority assignment for the processors using a FPP and the CAN network are given in Figure 7.10, where $\tau_1 > \tau_2$ indicates that the priority of τ_1 is higher than the priority of τ_2. We note that the release jitter formulas can be used also for local message passing: this will be the case for $Processor_C$ where we assume that a synchronization mechanism (similar to a private semaphore) is used to ensure the precedence constraint. It should be noted that the priority of τ_9 is higher than the priority of τ_8 in order to avoid scheduling anomalies. On $Processor_B$, an EDF scheduling policy is used, therefore, the priority assignment is dynamic, and is not given in Figure 7.10.

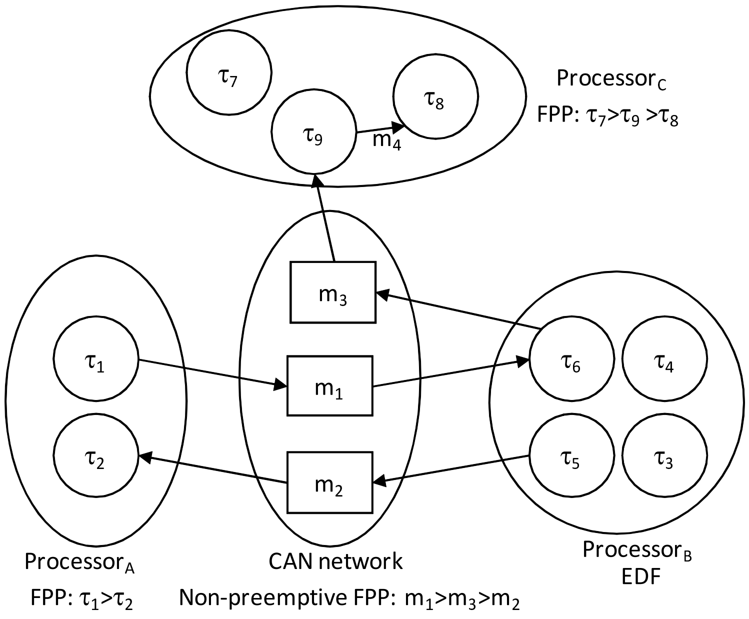


Figure 7.10. *A case study*

The principles of the holistic analysis were presented in section 7.3.2: it consists of computing the WCRT of the tasks and messages, and, for a task or a message i preceding a task or a message j, to inject the WCRT of i as a release jitter of j (note that if a task or message has several predecessors, then its release jitter becomes the maximal value of the WCRT of the preceding tasks or messages). We then iterate, modifying the release jitters of the tasks, which can increase the WCET, that will increase some release jitters, etc. until a fixed-point is reached, i.e. until the WCRTs are the same between one iteration and the next (see equation (7.1)).

The fixed-point of the holistic analysis is not dependent on the order used to compute the WCRT; nevertheless, taking the tasks and messages dependencies into account can help to reduce the number of computation steps before a fixed-point in the release jitter computation is reached. The dependencies are shown in Figure 7.11: the plain arrows represent the dependencies due to jitter (e.g. a modification of the WCRT of the message m_2 might have an impact on the task τ_2 through the value of the release jitter), and the dashed arrows represent the dependencies due to local scheduling and the priorities. For example, on $Processor_C$ as the priority of τ_7 is the highest, if no change is made to the release jitter of the other tasks, it can have an impact on

τ_7, while on $Processor_B$, as the priorities are dynamic, all the tasks are interdependent. Note that on $Processor_B$ all the tasks are interdependent but they were not all represented by the dashed arrows in order to improve readability.

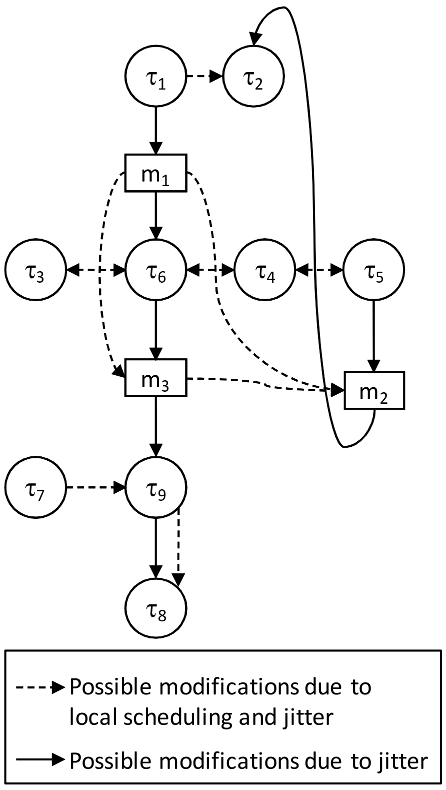

τ1
τ2
m1
τ3
τ6
τ4
τ5
m3
m2
τ7
τ9
τ8
Possible modifications due to local scheduling and jitter
Possible modifications due to jitter

Figure 7.11. *Dependencies due to the jitter and the scheduling*

The parameters of the tasks and messages are given in Table 7.6. Note that for any message dependency, the involved tasks and messages must have the same period. This constraint could be removed using generalized precedence constraints as in [FOR 10] or in [RIC 01b].

As there are loops in the precedencing graph (see Figure 7.11), we decided to study the WCRT of the tasks in this order: $\tau_1, m_1, \tau_3, \tau_4, \tau_5, \tau_6, m_3, m_2, \tau_7, \tau_9, \tau_8, \tau_2$ which does not introduce any dependency loop. As a consequence, the holistic analysis should converge in one pass only (we will nevertheless undertake a second pass in order to make sure that we reach the fixed-point).

$Task$	C_i	D_i	T_i	$Task$	C_i	D_i	T_i
τ_1	52	200	100	τ_7	28	60	60
τ_2	52	280	160	τ_8	25	320	100
τ_3	10	60	40	τ_9	14	250	100
τ_4	20	85	60	m_1	3		100
τ_5	52	150	160	m_2	3		160
τ_6	52	220	100	m_3	3		100

Table 7.2. *Initial parameters of tasks and messages*

$Task$	J_i^1	$WCRT^1$	$Task$	J_i^1	$WCRT^1$
τ_1	0	52	τ_7	0	28
τ_2	110	266	τ_8	219	337
τ_3	0	10	τ_9	177	219
τ_4	0	33	m_1	52	58
τ_5	0	98	m_2	98	110
τ_6	58	168	m_3	168	177

Table 7.3. *First pass*

7.6.1. *First pass of the holistic analysis*

For the first pass, we obtain the WCRTs given in Table 7.6.1 and we use them to introduce release jitters on the preceded tasks or messages. For example, the WCRT of τ_1 is introduced as a release jitter for m_1.

For τ_1, τ_2, τ_7, τ_8, and τ_9 we will use equation (7.6) in order to compute the duration of their respective level-i busy periods, and then deduce their WCRT using equation (7.5). For m_1, m_2 and m_3 we will consider that the τ_{bit} is 0.02 time units and use the equations (7.21) and (7.22). For tasks τ_3, τ_4, τ_5, and τ_6, we will use equation (7.7) to compute the length of the longest busy period. Then equation (**??** is used to compute the set of points to test, and for each point we use equation (7.12) to compute the deadline-d busy period duration, and then deduce the corresponding response time using equation (7.19). The WCRT of a task will then be given by equation (7.16).

We detail the calculation made to obtain the first pass results. We trivially obtain $WCRT(\tau_1) = 52$ as it is the highest priority task on $Processor_A$. For the message m_1, the jitter is thus 52 (the WCRT of τ_1), and $WCRT(m_1) = 58$ is obtained immediately also, with a blocking time duration equal to $C_b = 3$, corresponding to the message m_2 or the message m_3 sent on the network.

For the $Processor_B$, we first update $J_6 = WCRT(m_1) = 58$, and we compute the longest busy period $L = 240$. For the task τ_3, we compute set A (see equation (7.15)) and we get $A = \{0, 25, 40, 80, 85, 90, 102, 120, 145, 160, 200, 202, 205\}$. Then, using the equation (7.14) for every value of this set, we find a WCRT of 10, obtained for $a = 0$. For τ_4 and τ_5 we find, respectively, a WCRT of 33 for $a = 77$ and 98 for $a = 12$. For τ_6, we find a WCET of 168 for $a = -58$. We recall that we can have $a \in [-J_i, , L - J_i - C_i]$ (see section 7.4.4.2).

We can update the release jitter of m_2 (sent by τ_5) and m_3 sent by τ_6. Then we use equation (7.22) to compute the WCRT of m_2 and m_3. The WCRT of the first instance of m_2 in the synchronous busy period is 110, which is smaller than the period of the message, therefore, we investigate only for $k = 1$. Then we can update the release jitter of τ_2 which is waiting for the message m_2. For the message m_3, we assume that its blocking factor is $C_b = C_{m_2} = 3$. The WCRT of the first instance of the message is 177 $> T_{m_3} = 100$; therefore, we have to study the following instance ($k = 2$), for which we obtain a WCRT of 80, which is less than T_{m_2}; therefore, we do not have to study the following instances. Thus, the WCRT of m_3 is 177, enabling us to update the release jitter of τ_9.

A FPP is used to schedule $Processor_C$. First we compute the WCRT of τ_9 using equation (7.6): for $k = 1$ we obtain a WCRT of 219, which is higher than $T_9 = 100$; therefore we compute the WCRT for $k = 2$ and find a WCRT of 133, which is still higher than T_9. We then investigate the third job of τ_9 in the busy period ($k = 3$) and find a WCRT of 75 which is lower than T_9. Therefore, the WCRT of τ_9 is $\max(219, 133, 75) = 219$. This response time gives a release jitter for the task τ_8, as this task is waiting for an internal message sent by τ_9. For τ_8, with $k = 1$ (i.e. first job in the synchronous busy period), we find a WCET of 323 time units (note that this is clearly an unrealistic upper bound because τ_9 is counted twice: once as an interference, and once as delaying τ_8 with a release jitter). We use equation (7.6) for $k = 2$ and find a longer response time (337), which is an illustration of the fact that the first job of a busy period does not always have the longest response time. We then check for $k = 3, 4, 5, 6, 7, 8, 9$ where we find, respectively, a response time of $304, 271, 224, 191, 158, 111, 78$. So the WCRT of τ_8 is 337, which is the maximum of these values (corresponding to $k = 2$). The WCRT of τ_7 equals its WCET as it is the highest priority task and it does not suffer any release jitter.

Finally, the WCRT of τ_2 is obtained: its release jitter is 110, the WCRT of m_2. Using equation (7.6), we find for $k = 1$ a WCRT equal to 266. For $k = 2$, we obtain a WCRT of 210, and for $k = 3$ a WCRT of 154, which is smaller than the period $T_2 = 160$ and allows us to stop. Therefore, we get the WCET of τ_2, which is the maximum value of 266, 210 and 154.

7.6.2. *Second pass of the holistic analysis*

We picked the order in which we applied the WCRT computations wisely, but as there are dependency loops, we need to compute the WCRT with the release jitters that we computed. The second pass gives the same response time for each task; therefore, we reached the fixed-point of the holistic analysis, and the WCRT that we computed during the first pass are the actual WCRTs. As they are lower than the respective relative deadlines of the task, we can conclude that the system is feasible.

It should be noted that if we had not chosen the order in function of the task and message dependencies, we would probably have needed more passes before reaching the fixed-point of the holistic analysis.

7.7. Conclusion

We presented the holistic analysis and the scheduling methods used to compute the WCRT of the tasks (with a FPP, and with a deadline-driven scheduling algorithm) and messages on a CAN network, as well as the references for other kinds of networks. This method has a relatively low complexity and is scalable to large distributed systems.

However, all the nodes of the network are supposed to start synchronously. If this is not a realistic hypothesis, then it should be possible to add a release jitter to all the messages, to account for the lag between the nodes of the network. The WCRT computation methods are locally optimal (i.e. exact) for sporadic (non concrete) task systems, but on a global system scale, they provide the upper bounds of the WCRT as the worst-case scenario is not aways realistic. For strictly periodic, concrete task systems, the holistic analysis and the WCRT computation methods presented in this chapter are an upper bound on the real WCRT, because the critical instants taken into account are not necessarily happening in the system. As a result, a negative answer to the feasibility problem does not mean that the system is not feasible. Nevertheless, we have validated systems with an average processor load of 70% with a holistic analysis [RIC 02].

Some methods use the relations among the tasks in order to eliminate some non-realistic scenarios during the analysis. As an example, [PEL 05, PEL 07] takes into account the offset between the tasks and messages in order to propose more accurate tests.

As the holistic analysis has a relatively low computational complexity, it is possible to use it for system dimensioning as in [DOR 08] or to help a designer to assign priorities to tasks and messages [RIC 01a]. Moreover, polynomial approximation algorithms (fully polynomial-time approximation schemes) have been studied to reduce the time needed for WCRT computation [BIN 09, NGU 09].

7.8. Bibliography

[AUD 91] AUDSLEY N. C., DD Y., "Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times", December 1991.

[AUD 93] AUDSLEY N., BURNS A., RICHARDSON M., TINDELL K., WELLINGS A., "Applying new scheduling theory to static priority pre-emptive scheduling", *IEEE Software Engineering*, vol. 8, p. 284-292, 1993.

[AUD 97] AUDSLEY N., GRIGG A., "Timing analysis of the ARINC 629 databus for real-time application", *Microprocessors and Microsystems*, vol. 21, p. 55–61, 1997.

[BAK 91] BAKER T. P., "Stack-based scheduling for realtime processes", *Real-Time Systems*, vol. 3, p. 67–99, April 1991.

[BAR 90] BARUAH S. K., HOWELL R. R., ROSIER L., "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor", *Real-Time Systems*, vol. 2, p. 301-324, 1990.

[BAT 98] BATE I., BURNS A., "Investigation of the pessimism in distributed systems timing analysis", *10th Euromicro Conference on Real-Time Systems*, p. 107-114, June 1998.

[BAU 10] BAUER H., SCHARBARG J.-L., FRABOUL C., "Improving the worst-case delay analysis of an AFDX network using an optimized trajectory approach", *IEEE Transactions on Industrial Informatics*, vol. 6, p. 521–533, 2010.

[BIN 08] BINI E., NATALE M. D., BUTTAZZO G. C., "Sensitivity analysis for fixed-priority real-time systems", *Real-Time Systems*, vol. 39, num. 1-3, p. 5–30, 2008.

[BIN 09] BINI E., NGUYEN T. H. C., RICHARD P., BARUAH S. K., "A response-time bound in fixed-priority scheduling with arbitrary deadlines", *IEEE Trans. Computers*, vol. 58, num. 2, p. 279-286, 2009.

[BRI 03] BRIL R., VERHAEGH W., POL E.-J., "Initial values for online response time calculations", *15th Euromicro Conference on Real-Time Systems*, p. 13 - 22, 2003.

[CHE 90] CHEN M.-I., LIN K.-J., "Dynamic priority ceilings: a concurrency control protocol for real-time", *Real-Time Systems*, vol. 2, num. 4, p. 325–346, 1990.

[CHO 04] CHOQUET-GENIET A., GROLLEAU E., "Minimal schedulability interval for real-time systems of periodic tasks with offsets", *Theoretical Computer Science*, vol. 310, num. 1–3, p. 117–134, January 2004.

[DAV 07] DAVIS R. I., BURNS A., BRIL R. J., LUKKIEN J. J., "Controller area network (CAN) schedulability analysis: refuted, revisited and revised", *Real-Time Systems*, vol. 35, num. 3, p. 239–272, 2007.

[DAV 08] DAVIS R. I., ZABOS A., BURNS A., "Efficient exact schedulability tests for fixed priority real-time systems", *IEEE Trans. Computers*, vol. 57, num. 9, p. 1261–1276, 2008.

[DAV 09] DAVIS R., BURNS A., A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems, Report num. YCS-2009-443, University of York, Department of Computer Science, 2009.

[DER 74] DERTOUZOS M., "Control robotics: the procedural control of physical processes", *IFIP Congress*, p. 807-813, 1974.

[DOR 08] DORIN F., RICHARD M., GROLLEAU E., RICHARD P., “Minimizing the number of processors for real-time distributed systems”, *Real-Time and Network Systems*, Rennes, France, October 2008.

[DOR 10] DORIN F., RICHARD P., RICHARD M., GOOSSENS J., “Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities”, *Real-Time Systems*, vol. 46, num. 3, p. 305–331, 2010.

[EIS 08] EISENBRAND F., ROTHVOSS T., “Static-priority real-time scheduling: response time computation is NP-hard”, *Real-Time Systems Symposium*, Barcelona, December 2008.

[EIS 10] EISENBRAND F., ROTHVOSS T., “EDF-schedulability of synchronous periodic task systems is coNP-hard”, *Symposium on Discrete Algorithms*, Austin, TX, January 2010.

[ERM 97] ERMEDAHL A., HANSSON H., SJODIN M., “Response-time guarantees in ATM networks”, *18th IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, p. 274–284, 1997.

[FOR 10] FORGET J., BONIOL F., GROLLEAU E., LESENS D., PAGETTI C., “Scheduling dependent periodic tasks without synchronization mechanisms”, *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, Stockolm, Sweden, April 12-15th 2010.

[GEO 96] GEORGE L., RIVIERRE N., SPURI M., Preemptive and non-preemptive real-time uniprocessor scheduling, Report num. RR-2966, INRIA, 1996.

[GOO 03] GOOSSENS J., “Scheduling of offset free systems”, *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 24, num. 2, p. 239–258, 2003.

[HEN 01] HENDERSON W., KENDALL D., ROBSON A., “Improving the accuracy of scheduling analysis applied to distributed systems computing minimal response times and reducing jitter”, *Real-Time Systems*, vol. 20, num. 1, p. 5–25, 2001.

[JAC 55] JACKSON J., Scheduling a production line to minimize maximum tardiness, Report, University of California, Los Angeles, 1955.

[JEF 91] JEFFAY K., STANAT D., MARTEL C., “On non-preemptive scheduling of periodic and sporadic tasks”, *12th IEEE Real-Time Systems Symposium*, San Antonio, TX, p. 129-139, December 1991.

[JEF 93] JEFFAY K., STONE D., “Accounting for interrupt handling costs in dynamic priority task systems”, *IEEE Real-Time Systems Symposium*, Raleigh-Durham, NC, USA, 1993.

[JOS 86] JOSEPH M., PANDYA P., “Finding response times in real-time system”, *The Computer Journal*, vol. 29, p. 390-395, 1986.

[KAI 82] KAISER C., “Exclusion mutuelle et ordonnancement par priorité”, *Technique et Science Informatiques*, vol. 1, p. 59-68, 1982.

[LEH] LEHOCZKY J., SHA L., STROSNIDER J., TOKUDA H., “Fixed priority scheduling theory for hard real-time systems”, *Foundations of Real-Time Computing: Scheduling and resource management*.

[LEH 90] LEHOCZKY J., “Fixed priority scheduling of periodic task sets with arbitrary deadlines”, *IEEE Real-Time Systems Symposium*, p. 201-213, 1990.

[LEN 77] LENSTRA J., RINNOOY KAN A., P. B., “Complexity of machine scheduling problems”, *Ann. Discrete Math.*, vol. 1, p. 343-362, 1977.

[LEU 80] LEUNG J. Y.-T., MERRILL M. L., “A note on preemptive scheduling of periodic, real-time tasks”, *Information Processing Letters*, vol. 11, num. 3, p. 115–118, November 1980.

[LEU 82] LEUNG J., WHITEHEAD J., “On the complexity of fixed-priority scheduling of periodic real-time tasks”, *Performance Evaluation*, p. 237–250, 1982.

[LIU 73] LIU C., LAYLAND J., “Scheduling algorithms for multiprogramming in a hard-real-time environment”, *Journal of the ACM*, vol. 20, num. 1, p. 46–61, 1973.

[MAL 93] MALCOLM N., ZHAO W., “Guaranteeing synchronous messages with arbitrary deadline constraints in an FDDI network”, *IEEE Conf. Local Computer Networks*, p. 186–195, 1993.

[MOK 83] MOK A., Fundamental design problems of distributed systems for the hard-real-time environment, PhD thesis, Massachusetts Institute of Technology, 1983.

[NGU 09] NGUYEN T. H. C., RICHARD P., BINI E., “Approximation techniques for response-time analysis of static-priority tasks”, *Real-Time Systems*, vol. 43, num. 2, p. 147-176, 2009.

[PED 97] PEDRO P., BURNS A., “Worst case response time analysis of hard real-time sporadic traffic in FIP networks”, *9th Euromicro Workshop on Real-time Systems*, p. 3–10, 1997.

[PEL 05] PELLIZZONI R., LIPARI G., “Improved schedulability analysis of real-time transactions with earliest deadline scheduling”, *Real-Time and Embedded Technology and Applications Symposium, IEEE*, p. 66-75, IEEE Computer Society, 2005.

[PEL 07] PELLIZZONI R., LIPARI G., “Holistic analysis of asynchronous real-time transactions with earliest deadline scheduling”, *Journal of Computer and System Sciences*, vol. 73, num. 2, p. 186 - 206, 2007, Special Issue: Real-time and Embedded Systems.

[RIC 01a] RICHARD M., RICHARD P., COTTET F., “Task and message priority assignment in automotive systems”, *4th FeT IFAC Conference on Fieldbus Systems and their Applications*, p. 105–112, November 2001.

[RIC 01b] RICHARD P., COTTET F., RICHARD M., “On-line scheduling of real-time distributed computers with complex communication constraints”, *ICECCS’2001*, Skövde, Sweden, p. 26–34, June 2001.

[RIC 02] RICHARD M., Contribution à la Validation des Systèmes Temps Réel Distribués : Ordonnancement à Priorités Fixes et Placement, PhD thesis, University of Poitiers, 2002.

[RID 04] RIDOUARD F., RICHARD P., COTTET F., “Negative results for scheduling independent hard real-time tasks with self-suspensions”, *25th IEEE International Real-Time Systems Symposium*, December 2004.

[SER 72] SERLIN O., “Scheduling of Time Critical Processes”, *Spring Joint Computer Conference*, AFIPS, p. 925–932, 1972.

[SHA 90] SHA L., RAJKUMAR R., LEHOCZKY J., “Priority inheritance protocols : an approach to real-time synchronization”, *IEEE Transactions on Computers*, vol. 39, p. 1175-1185, 1990.

[SJO 98] SJODIN M., HANSSON H., "Improved response-time analysis calculations", *IEEE Real-Time Systems Symposium*, p. 399-408, 1998.

[SJO 99] SJODIN M., HANSSON H., "Analysing multimedia traffic in real-time ATM networks", *5th Real-Time Technology and Applications Symposium*, Vancouver, Canada, IEEE Computer Society, August 1999.

[SJO 00] SJODIN M., Predictable high-speed communications for distributed real-time systems, PhD thesis, Uppsala University, Information Technology, 2000.

[SPU 96a] SPURI M., Analysis of deadline scheduled real-time systems, Report num. RR-2772, INRIA, 1996.

[SPU 96b] SPURI M., Holistic analysis for deadline Scheduled real-time distributed systems, Report num. RR-2873, INRIA, 1996.

[TIN 94a] TINDELL K., Adding time-offsets to schedulability analysis, Report num. YCS-221, University of York, 1994.

[TIN 94b] TINDELL K., CLARK J., "Holistic schedulability analysis for distributed hard real-time systems", *Microprocessing and Microprogramming*, vol. 40, 1994.

[TIN 94c] TINDELL K., HANSSON H., WELLINGS A., "Analysing real-time communications: controller area network (CAN)", *Real-Time Systems Symposium*, San Juan , Puerto Rico, IEEE Computer Society Press, p. 259–263, 1994.

[TIN 95a] TINDELL K., BURNS A., WELLINGS A. J., "Analysis of hard real-time communications", *Real-Time Systems*, vol. 9, p. 147-171, Springer Netherlands, 1995.

[TIN 95b] TINDELL K., BURNS A., WELLINGS A. J., "Calculating controller area network (CAN) message response times", *Control Engineering Practice*, vol. 3, num. 8, p. 1163-1169, August 1995.

[TRA 06] TRAORÉ K., GROLLEAU E., COTTET F., "Schedulability analysis of serial transactions", *Real-Time and Network Systems, RTNS'06*, Poitiers, France, p. 141-149, May 30-31 2006.

[VES 94] VESTAL S., "Fixed-Priority sensitivity analysis for linear compute time models", *IEEE Transactions on Software Engineering*, vol. 20, num. 4, p. 308-317, 1994.

[WIL 08] WILHELM R., ENGBLOM J., ERMEDAHL A., HOLSTI N., THESING S., WHALLEY D. B., BERNAT G., FERDINAND C., HECKMANN R., MITRA T., MUELLER F., PUAUT I., PUSCHNER P. P., STASCHULAT J., STENSTRÖM P., "The worst-case execution-time problem: overview of methods and survey of tools", *ACM Trans. Embedded Comput. Syst.*, vol. 7, num. 3, 2008.

[ZHA 95] ZHANG S., BURNS A., "Guaranteeing synchronous message sets in FDDI networks", *13th IFAC Workshop on Distributed Computer Control Systems*, p. 107–112, 1995.

Chapter 8

Software Engineering for Adaptative Embedded Systems

8.1. Introduction

Most real-time embedded software applications are designed to control a physical system evolving in an environment where characteristics may vary. In order to adapt to these variations, the system might have to modify its own behavior, which is controlled by the embedded software application. Thus, sometimes this embedded software application has to adapt to the the modifications of the system's environment.

The objective of this chapter is to explain and illustrate the principles that lead the design of an adaptative distributed real-time embedded system (ADRES).

Pursuing this objective, we describe the problems raised by the design of ADRESs, the theoretical solutions highlighted by researchers to tackle these problems and the technical solutions those using theoretical results. We illustrate this presentation with the design of an ADRES extracted from the field of robotics.

This chapter is organized as follows: section 8.2 presents an overview of the challenges raised by the design of ADRESs. In section 8.3, we present in more detail the theoretical problems raised by such systems, and we present some of the existing solutions. In section 8.4, we describe and compare the existing technologies to those using these theoretical results. Section 8.5 is dedicated to the presentation of the robotic example to illustrate the design of an ADRES. In section 8.6, we present the

Chapter written by Etienne BORDE.

Distributed Systems: Design and Algorithms Edited by Serge Haddad, Fabrice Kordon, Laurent Pautet and Laure Petrucci
© 2011 ISTE Ltd. Published 2011 by ISTE Ltd.

corresponding design, and discuss the reasons for the specification choices. Finally, section 8.8 concludes this chapter.

8.2. Adaptation, an additional complexity factor

Considering the increasing complexity of embedded systems, software engineering activities related to embedded systems advocate splitting a system into gradually smaller pieces which are easier to implement, analyze, and maintain. Following this "divide and conquer" strategy, a system is split into subsystems, themselves split into components, subcomponents, and eventually subprograms (i.e. operations). In this context, two interconnected layers of specification are clearly identifiable: the system architecture and software architecture.

Orthogonally to undertake this process, the adaptation mechanisms of a system may be specified at any of these layers. At system layer, the possible abstract behaviors of a system are grouped into sets of functionalities (often referred as operational modes) to which requirements are associated. At the software layer, components represent the specific behaviors of the software applications that control the system: these have to be modified in cases of adaptation of the system's behavior.

Considering this very synthetic presentation, it is easily understand that dealing with adaptation brings additional complexity to appreciate the traditional design process of embedded systems.

We illustrate this additional complexity factor all along the remainder of this chapter, going through abstract and general considerations, theoretical problems and solutions, as well as practical solutions that help in addressing this complexity.

8.2.1. *Adaptation, a step towards autonomy*

Presenting the emergence of *autonomic computing* in the management systems of information technology [GAN 03] introduces adaptation as the last step towards autonomic systems.

In order to create an autonomic system, an adaptation control loop relies on (i) a set of sensors that monitor the state of the system and it's environment, (ii) a supervisor that aggregates and analyzes the data provided by the sensors, (iii) a planner that produces the set of actions that need to be undertaken in order to adapt the behavior of the system, and (iv) an executor that applies the different adaptations on the system [COM 06]. Figure 8.1 illustrates the different steps of adaptation in an autonomic system.

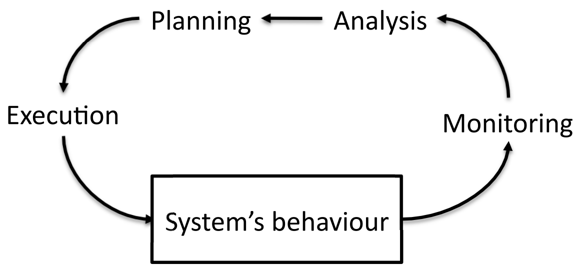


Figure 8.1. *Basic control loop for autonomous systems*

The level of autonomy of an adaptative system lies in the capacity of the analyzer to model and synthesize the behavior of the system and the state of its environment. The level of autonomy of a system also lies in the capacity of the planner to decide what the most suitable behavior to adopt is, and how to implement this behavior.

What is adaptation

In [ORE 99], a "self-adaptive software modifies its own behavior in response to changes in its operating environment, [i.e.] anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation." In this chapter, we consider adaptation as the capacity of a system to modify its own behavior.

Different types of adaptation

Depending on its objectives, adaptation will occur at different moments of a system's life cycle. This led the research community studying adaptation to define three types of adaptation:

– static adaptation: to proceed to adaptation, this system stops completely. This type of adaptation aims to provide mechanisms that tune the system under development. Static adaptation is mainly used for the maintenance of systems for which the execution can be stopped [CHA 01];

– pseudo-dynamic adaptation is executed while the system is running but all the possible adaptations are exhaustively known at the design stage [KE 07, BOR 09]. This type of adaptation excludes the maintenance activity (which cannot be provided at design time) but is dedicated to the implementation of adaptation mechanisms for critical embedded systems (systems that must be analyzed in depth at design time);

– dynamic adaptation is also executed while the system is running, but corresponds to adaptations that cannot be foreseen during the design stage (corrective and evolutive maintenance) [BRI 05, DAV 09, GRO 06]. This type of adaptation is very important for the maintenance of real-time systems as most of them cannot be completely stopped, but must continue to provide the services they were designed for.

Rationale for adaptation

Bringing an additional factor of complexity, adaptation enables the implementation of crucial requirements of the design and management of software systems, such as:

– evolution and maintenance [BRI 05, DAV 09], for instance to correct bugs while the system is running;

– resource allocation management to ensure quality of service attributes [IVA 99]; or to implement fault-tolerance mechanisms [DAI 06];

– implement different phases of the system life-cycle, for instance different steps of the mission of an unmanned vehicle.

Regarding real-time systems, an important consideration is that the transition from one behavior to another must often be realized (i) without switching off the system; (ii) within a given time limit, and (iii) reducing the perturbation of the system due to the adaptation process where possible. Those two last requirements are contradictory: reducing the effect on the system will increase the adaptation time and vice versa [BRI 05].

8.2.2. *Autonomy versus predictability*

In the context of embedded systems controlling physical systems, predictability may be a requirement of prime importance. Depending on the importance of the behavior of a system, this requirement drastically influences to what extent the system can be autonomic.

Indeed, autonomy and predictability are two characteristics opposed to one another insofar as the first allows the system to decide by itself the behavior that best answers its new operating environment, while the second that all the possible behaviors of the system and their characteristics are predicted at the time of the design of the system.

To illustrate this, we can consider the two following extreme situations. On the one hand, an extremely critical system will have only one possible behavior, configured and analyzed by the system designer. This system is thus easier to analyze. On the other hand, a totally autonomous system will be able to learn from previous situations and decide by itself what new behavior to adopt.

Beyond these extreme situations, and considering the complexity of predicting the behavior of a software system, it can be seen that the criticality of a system will limit its adaptation mechanisms in order to improve the analysability of the system (e.g. in terms of adaptation time, memory sizing, etc.). For instance, a critical adaptative system will privilege static or pseudo-dynamic adaptation mechanisms to dynamic ones. Conversely, autonomy relies on dynamic mechanisms in order (i) to decide behavior to adopt, and (ii) to apply these modifications to the application while it is executing.

In the introduction to this section, we have presented adaptation as a step towards autonomy. We have then explained that the level of autonomy will mainly impact the supervision and planing activities. As an answer to the trade-off between predictability and autonomy, different types of supervision and planning policy have been proposed. Among these, we can distinguish three complexity levels in the policy:

– the *condition/action-based* policy: the supervision activity is described as a set of conditions that a source state of the system must ensure in order to transit to a target state. Following the same idea, the planning activity is defined as a set of actions that have to be realized on the system [DAV 09];

– the *goal-based* policy: in this case, the supervision activity is similar to that explained above (i.e. a set of rules on the source states). The planning activity is slightly different as the actions executed are deduced from the specification of the target behavior [BOR 09];

– the *utility-based* policy: considering that a level of utility and resources utilisation is associated to each target state, a utility function computes the more suitable target state (analysis) from which the adaptation action executed is deduced (planning) [GRO 06].

Obviously, the level of autonomy increases from the first to the third policy while the behavior of the system becomes more difficult to predict and to analyze.

From the general concerns we presented in this section, several theoretical problems and solutions have been highlighted in studies related to the design of ADRESs. We present the most important in next section.

8.3. Theoretical aspects of adaptation management

In this section, we present the main issues related to an adaptation management process in ADRES. These issues deal with the coordination of distributed adaptation processes, the definition of a system state that allows adaptation to occur in a safe manner, and the specificities of adaptation in real-time systems (considering timing requirements).

8.3.1. *Coordination of control loops*

In a distributed real-time system, the adaptation control loop needs to be distributed in order to control the behavior of all the computation units involved in the adaptation process. Thus, these different control loops need to be coordinated in order to adapt the overall behavior of the system. In this section, we present different coordination strategies.

Centralized coordination

In a centralized coordination, the overall behavior of the system is controlled by a unique supervisor which has a global knowledge of the system's state, its environment and its possible behaviors. Thus, this supervision entity is the central adaptation unit of the system (i.e. the unique feedback and command unit). This coordination eases the design as the adaptation process is clearly identified and is easy to manage (local computation only). However, this central entity constitutes *de facto* a *bottleneck* reducing the scalability (in terms of performance) of this approach. From a dependability perspective, this centralized supervisor may also constitute a single point of failure.

Decentralized Coordination

Contrasting the centralized coordination, a decentralized coordination is made of a set of autonomous but interacting units that control their adaptation process by themselves [WOL 05]. This approach solves the scalability problem and offers much more flexibility as the different entities of the system can be designed and used independently. The main issue of a decentralized coordination is to assess the system's global behavior. As a consequence, this approach requires the implementation of an important number of interactions in order to maintain the overall consistency of the system [WOL 05].

Hierarchical coordination

An alternative to the centralized or decentralized coordination is the hierarchical coordination. Different proposals for the hierarchical coordination of control loops have already been published [BRO 03, DAI 06, ROY 07, ARM 03]. To summarize, a hierarchical adaptation is organized as a set of control loops in which the adaptation entity of higher hierarchical level controls the adaptation of the system; either by controlling the entities of lower hierarchical level [DAI 06, ARM 03] or by controlling the adaptation process in priority [BRO 03, ROY 07]. The hierarchical adaptation process enables the implementation of a trade-off between the difficulties of implementing a decentralized adaptation, and the scalability issues raised by a centralized adaptation.

In an adaptative system, the adaptation control loops are dedicated to the election of the system's behavior. In these loops, the feedback inputs represent the state of the system (and its environment) while the output is a set of actions that have to be undertaken in order to modify the system's behavior. When designing the software application that is embedded in a system in order to control its functionalities, an important question is to determine when the software application is ready to undertake these adaptation actions. This state of the system, in which the adaptation process can really begin, is often referred to as the system's *"safe state"*.

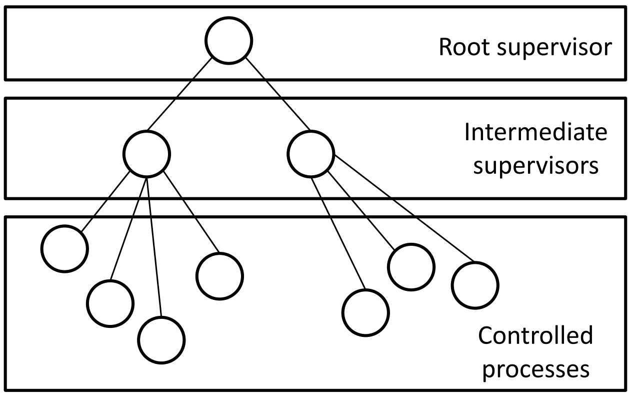


Figure 8.2. *Hierarchical coordination of adaptation control loops*

8.3.2. *The safe state problem*

In a software application, adaptation is a process led by the software to modify itself. It is thus a risky process that requires that the entities impacted (i.e. modified) by the adaptation are isolated from the rest of the application. An initial proposition for safe state was defined in [KRA 90] as the *«quiescent state»*, considering the interactions among software components as transactions. A transaction is the exchange of one or more messages between two components. In this model, a transaction is assumed (i) to be limited in time and (ii) to ensure that the initiator of a transaction is aware of its completion. As a consequence of being involved in a transaction, a component may initiate transactions with other components. In the meanwhile, the initial transaction remains active. In [KRA 90], a component is in the quiescent state when (i) it is not currently involved in a transaction, (ii) it is not waiting for the result of an outgoing transaction, and (iii) none of its operations that are likely to produce (directly or through intermediate entities) an incoming transaction are executing.

Due to this definition and the complexity of interactions among software components, it is not possible to ensure that an adaptation requirement would be met during runtime, thus producing an *adaptation starvation*. The «quiescent state» hypothesis was relaxed in [VAN 06], but the associated protocol also does not prevent the adaptation starvation (i.e. the adaptation might never occur).

In [Hof93], the developer of the application decides and directly implements the definition of the system's safe state. As a consequence, it is very difficult to ensure *a priori* the correctness of the adaptation.

The different definitions of safe states given in this section raise important issues regarding their usage in a real-time and embedded system: how to limit the adaptation time and how to evaluate the worst-case adaptation time. To answer these questions, the adaptation process of real-time systems has been studied using the application of mode switch protocols.

8.3.3. *Mode switch protocols*

A mode is usually defined as an abstract representation of a set of functionalities that the system must provide when the considered mode is active. On switching from one mode to another, the system modifies the current set of provided functionalities and thus self-modifies its own behavior (see the definition of adaptation, section 8.2.1).

Since the late 1980s, mode switch mechanisms have been intensively studied by researchers in real-time systems. Focusing mainly on the timing requirements in terms of the promptness of the mode switch and schedulability of the adaptative system, these studies consider the tasks of a system as the main adaptable entities. In [REA 04], the authors propose an exhaustive survey of modes switch protocols. This survey proposes a categorization of protocols, as well as a set of requirements to analyze them.

The requirements introduced are:

1) schedulability: the different tasks controlling the system must always be executed within a given deadline, and this must remain true during the mode switch. This requirement implies that it must be possible to easily verify the schedulability;

2) periodicity: the pattern of activation of periodic tasks should remain the same, even in presence of a mode switch;

3) promptness: in certain situations, a mode switch must be executed as fast as possible. The promptness represents the capacity of a system to switch from one mode to another rapidly;

4) consistency: if data are shared among different tasks, their consistency must be preserved. This property must remain true during a mode switch.

In order to answer those different (sometimes contradictory) requirements, different mode switch protocols have been proposed. Those protocols are characterized by the subset of requirements they privilege. In the remainder of this section, we provide a brief summary of the previously mentioned survey [REA 04].

Idle time protocol [TIN 92]

This protocol is a synchronous protocol: all the active tasks in the source mode, modified by the mode switch, must be terminated before the actual mode switch occurs.

The main idea of this protocol is to wait until all the tasks modified by the mode switch have finished before proceeding to the mode switch. As a consequence, (i) traditional techniques for schedulability analysis can be applied to such a protocol, (ii) the periodicity is preserved, and (iii) the computation of the worst case mode switch time (WCMST) is very easy (see equation (8.1)).

$$WCMST = \sum_{j \in Im} \lceil \frac{WCMST}{T_j} \rceil C_j + \sum_{j \in Un\ t.q.\ P_j \geq P_R} \lceil \frac{WCMST}{Tj} \rceil C_j \qquad (8.1)$$

In this equation, C_j, P_j, and T_j, represent, respectively, the worst case execution time (WCET), the priority, and the period of a task T_j. Im represents the set of tasks impacted by the mode switch, Un represents the set of tasks unchanged by the mode switch, and P_R represents the lowest priority among the priorities of the tasks impacted by the mode switch.

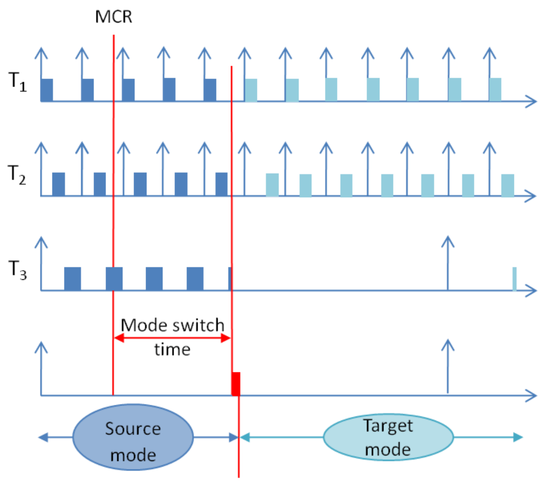


Figure 8.3. *Idle time protocol*

Figure 8.3 illustrates the behavior of a set of tasks in presence of a mode change request (MCR), the mode switch being configured with the idle time protocol.

As shown in this figure, this protocol prioritizes the schedulability to the detriment of the promptness: the WCMST can be very long as it allows multiple execution of the impacted tasks (see T_1 in Figure 8.3, executed three times between the mode switch request and the moment the target mode is reached).

Minimum single offset with or without periodicity [REA 04]

In order to tackle rapidness requirements, [REA 04] proposes another mode switch protocol that prioritizes the promptness to the detriment of periodicity. The idea of this protocol is that during the mode switch, only the active tasks that are impacted can finish their current release but no further release is permitted until the target mode is reached.

In this case, the schedulability analysis is also very simple (see equation (8.2)).

$$WCMST = \sum_{j \in Im} C_j \tag{8.2}$$

Figure 8.4 illustrates the behavior of a set of tasks in the presence of a MCR, the mode switch being configured with the minimum single offset protocol (without periodicity).

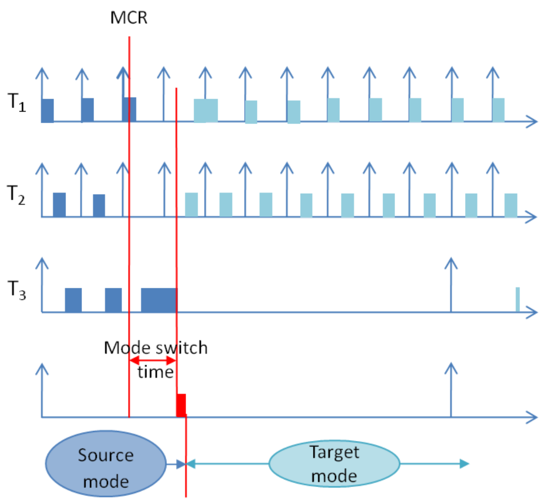


Figure 8.4. *Minimum single offset protocol*

In the figure, the minimum offset protocol without periodicity prioritizes the promptness to the detriment of the periodicity: the WCMST is shorter than with the idle time protocol but one release of task T_2 has been missed.

This protocol can be easily extended to preserve the periodicity of non-impacted tasks [REA 04]. Considering requirements other than periodicity, the evaluation of this protocol with periodicity is very close to the evaluation given when the periodicity is not ensured.

In order to again accelerate the mode switch, asynchronous protocols have been proposed. Contrary to the synchronous protocols, asynchronous protocols allow the simultaneous execution of tasks in the source mode and tasks in the target mode. Making the mode transitions faster, these protocols are much more difficult to analyze regarding their schedulability.

Utilization based [SHA 89]

The utilization-based protocol is an asynchronous protocol based on the processor capacity and usage: using the traditional schedulability analysis techniques, it is possible to determine whether the processor capacity can, depending on its current usage, proceed to a mode switch. In this protocol, tasks are internally modified by a mode switch: their WCET varies from one mode to another. A mode switch will thus modify the processor capacity and usage.

The main disadvantages of this solution are (i) the necessity to keep track of the processor usage (which induces an overhead in the runtime environment of tasks), and (ii) the difficulty in analyzing the schedulability of this protocol [TIN 92].

Asynchronous protocols [TIN 92, PED 98, REA 04]

Different asynchronous protocols have been proposed in published studies about mode switches [TIN 92, PED 98, REA 04]. These protocols result in a faster mode switch but are much more complex to analyze with regards to the WCMST and the schedulability. Indeed, the expression of the WCMST depends on the instant when the mode switch request is received. The corresponding value is considered as the time separating the last release of an impacted task with the moment of the arrival of the mode switch request. Those different protocols do not provide any method to calculate the instant corresponding to the occurrence of a mode switch request that leads to a worst-case scenario. As a consequence, the analysis must be launched considering different possible values, which limits its usability in critical real-time systems.

Sampling preserving mode switch [BOR 09]

Data sampling is another important aspect of real-time embedded systems. For instance, in order to produce the speed of a mobile system, several consecutive values of a position variable can be used. Knowing the interval separating the measure of these positions, the computation of the speed profile is easy. When the mode switch does not substantially modify the way values are produced, the protocols with periodicity presented earlier preserve the integrity of the data set: the position values produced in the source mode can be used together with the position values produced in the target mode. The data set obtained is consistent. When the values produced are substantially different in the source mode and in the target mode, then the computation of the speed might use an inconsistent value set. To avoid this problem, [BOR 09] extends the periodic protocols presented above: when a set of data is shared by synchronized tasks, then the mode switch occurs at the hyper-period of the set of synchronised tasks.

Figure 8.5 illustrates the behavior of a set of tasks in presence of a MCR, the mode switch being configured with the sampling preserving mode switch protocol.

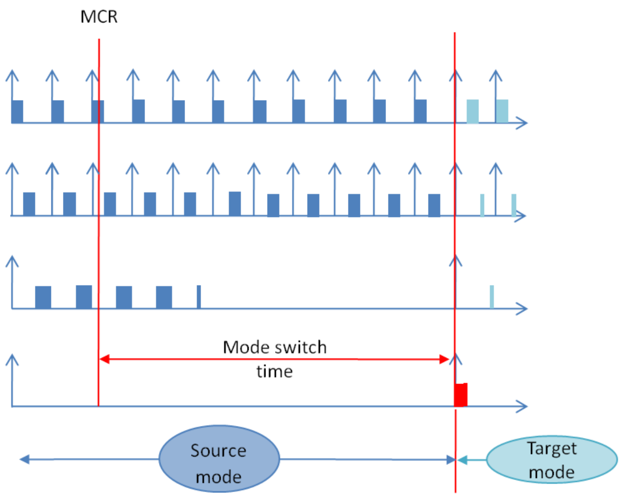


Figure 8.5. *Sampling preserving protocol*

Obviously, depending on the hyper-period of the set of synchronised tasks, the WCMST might be long. In this case, it is important to consider if the system is robust to the absence of updated data during one hyper-period. If it is robust, it is better to reset the data set and to switch mode with a faster protocol. If the system is not robust in this case, then it is necessary to wait until the hyper-period.

Time-triggered mode switch [FAR 06]

In a distributed real-time system, the coordination of mode switches must be done considering the global state of the system. In [FAR 06], authors consider *distributed* architectures deployed on a time-triggered bus. Taking advantage of this hypothesis, the mode switch request can be globally synchronized: it is typically performed at the end of a major cycle on the bus. The main drawback of this approach lies in the degradation of the promptness of the mode switch: the mode switch is propagated only once in the major cycle.

Different practical solutions have been proposed to deal with the additional complexity resulting from the dynamic adaptation of embedded systems. We present some of these solutions in the next section.

8.4. Technical solutions for the design of adaptative embedded systems

In the previous sections of this chapter, we have presented the different problems and theoretical solutions to address the design of ADRES. We have shown that the implementation of an ADRES requires complex problems to be solved and wise choices to be made. In this section, we present the techniques and practical solutions used to build ADRESs; this includes specific runtime infrastructures and dedicated design technologies.

8.4.1. *Infrastructure dedicated to adaptation*

Design patterns for adaptation

Usage of existing design patterns enables the implementation of the adaptation mechanisms to be structured the using a well-defined architectural style.

In [BOI 00], the authors rely on the *strategy pattern* (see Figure 8.6) to modify the implementation of a component. In this pattern, a context class provides a strategy interface that can be implemented in different classes (*Strategy_A* and *Strategy_B* in Figure 8.6). The idea of this pattern is that every method *m* provided by the interface *Strategy* is implemented in the *Context* class as follows: *currentStrategy.m()*. Thus, when calling *m()* on a *Context* object, the actual implementation is decided at runtime depending on the implementation of the *currentStrategy* interface.

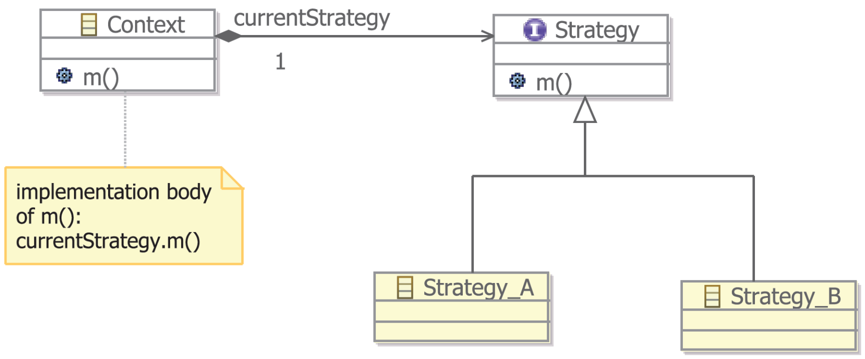


Figure 8.6. *Strategy design pattern*

The main problem with using such a technique is that it does not ensure that the conditions of a "safe-state" are met when the adaptation occurs because of its construction.

Middleware dedicated to adaptation.

The middleware of an application is a software layer that interfaces the applicative code with its underlying execution platform. As such, the middleware is usually responsible for ensuring the activation of components, as well as their interactions. Thus, the middleware can keep track of the activations/communications among components in order to ensure they are in a "safe state" when the adaptation process begins. In [BRI 05], the authors not only implement some safe adaptation mechanisms in the middleware, but also implement different adaptation policies to keep manage the trade-off between adaptation time and unavailability of services.

Software reconfiguration in component based architectures

Component-based software engineering (CBSE) has been used many times in the context of adaptative systems. Taking advantage of the clear separation between the different entities that constitutes the software application, CBSE enables the set of modification necessary to implement an adaptation to be clearly identified. Following this idea, [DAV 09] propose a script-based language that enables the behavior of a component-based application to be modified by applying a set of clearly identified modifications. The process applying these modifications, which aims the software configuration to dynamically change, is thus called reconfiguration. The main issue when dealing with dynamic reconfiguration in ADRESs is to enable the usual requirements and constraints of such systems, such as strict timing requirements, limited computation resources, and determinism to be met.

Software reconfiguration and architecture description languages

In order to provide ADRES designers with a better support for the design, implementation, and analysis of their products, certain architecture description languages (ADL) enable the description of features dedicated to adaptation.

Darwin [MAG 95] was one of the first ADL that included the specification of dynamic reconfigurations. The creation of this language was based on previous work that defined the reconfiguration semantics. This semantics is mainly based on:

– the notion of "quiescent state" presented in section 8.3.2;

– the definition of three possible states for a component:

- active (the component can initiate, accept, and service transactions),

- passive (the component continues to answer new transactions but it is not currently involved in a transaction and it does not initiate new transactions),

- and frozen (the component state will not be modified by a incoming requests).

```
component navigation {
  require position ;
}

component localization {
  provides position ;
}

component system {
  provides pos <dyn>;
  inst
    Nav:navigation ; // component instantiation
  bind
    pos —— dyn localization ;
    Nav.position —— localization . position ; // connection of components
}
```

Listing 8.1: Example of binding specification in Darwin

Besides, Darwin enables us to describe reconfiguration actions, for instance:

– the modification of components connections;

– the instantiation of a new component;

– the removal of an existing component.

Listing 8.1 illustrates the definition of a component (*system*) that dynamically instantiates a component of type *localization* when receiving a request on its port *pos*. In order to go further in the description of a dynamic behavior, Darwin enables more complex controls of the instantiation process to be specified. To do so, **forall** (iterator) and **when** (conditional) operators are available to describe the instantiation of subcomponents. However, the scope of the reconfiguration actions is limited to the dynamic creation and binding of components, which is insufficient to control the adaptation of an ADRES.

More recently, ADLs dedicated to the real-time systems have included features to describe adaptation mechanisms. In the architecture analysis and design language (AADL) [FEI 06], adaptation is modeled as a result of:

1) a set of operational modes;

2) a set of transitions between operational modes, with rules that enable to describe when a mode transition must be triggered;

3) a set of software configurations associated with each operational mode.

Figure 8.7 illustrates the definition of an adaptative system using the AADL. In this figure, *P* is the enclosing entity; a process containing one task *T*. *P* defines two operational modes *M1* and *M2*. In *M1*, *T* executes every 12 ms, while it executes every 25 ms in *M2*. Finally, the definition of the mode switches define that the process goes from *M1* to *M2* when receiving an event on port *p1*.

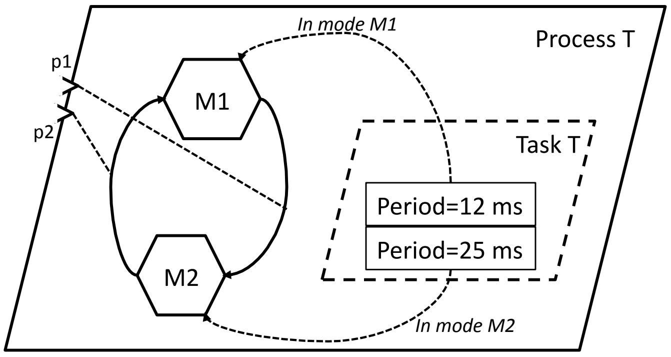


Figure 8.7. *Adaptation specification in AADL*

Inspired by the definition of modes and software reconfiguration proposed by the AADL, the component-oriented architecture language (COAL) [BOR 09] proposes the definition of the adaptation mechanisms of ADRESs by reconfiguration of generic software components.COAL also enables the mode switch automata to be configured in order to implement either of the *synchronous mode switch protocols* presented in 8.3.3. This feature is not explicit when using AADL mode switches.

In the different solutions presented in this section, reconfiguration is used to implement adaptation. In the latter solutions, reconfiguration is combined with mode switch mechanisms. Mode switches can be implemented without reconfiguration and vice versa. Used together, these artifacts enable the way the system adapts to variations of its environment to be specified. Describing the adaptative behavior of the system early in the design process gives the opportunity to detect design errors. It also consumes fewer resources as the adaptation mechanisms can be produced according to the specifications, thus avoiding the overhead of a generic implementation.

However, such an approach does not address unexpected adaptations (e.g. maintenance needs). As stated in the introduction of this chapter, our objective is to describe how to design ADRESs, which led us to focus on the early stages of the design process of such systems. We will thus focus on technologies that take advantage of the simultaneous specification of mode switches and reconfiguration to ease the design of ADRES.

8.4.2. *Existing technologies for the adaptation of embedded real-time embedded systems*

In this section, we present and compare different technologies that aids an ADRES designer. As the presence of adaptivity adds complexity, this kind of technology play a major role in the management of this complexity.

Time-triggered synchronous technologies

In order to improve the determinism of distributed real-time systems, time-triggered architectures enable (or limit the de-synchronization of) the distributed processors to be synchronized by sharing a global clock. Taking advantage of this hypothesis, the framework GIOTTO [HEN 01] automates the production of distributed ADRES on a time-triggered architecture. The corresponding approach also relies on a synchronous hypothesis: the logical execution time (LET) of the different tasks of the system takes zero time.

These hypotheses (global clock and null LET) ease the design of formally defined, analyzed and deterministic ADRES. In return, the reactivity of such application is lowered.

Time and event-triggered synchronous technologies

Releasing the time-triggered hypothesis but conserving the zero LET hypothesis, COMDES [KE 07] and xGiotto [GHO 04], have been proposed to develop ADRESs. As a consequence of releasing the clocks synchronization hypothesis, the computation of correct scheduling is harder and its completion is not guaranteed. As a compensation, the main benefit of these approaches is an increased reactivity.

These solutions do not emphasize the hierarchical management of control loops in higher level of abstraction than in the software specification. In industry, the definition of operational modes is mainly part of the system specification in the early stages of the development process. At these stages, modes can be described in a hierarchical manner following the system/subsystem decomposition. This aspect is weakly considered in these contributions.

Time and event-triggered asynchronous technologies

In order to prioritize rapid adaptation, MyCCM-HI [BOR 09] releases the zero LET hypothesis and gives the software designer the possibility to rely on a time-triggered, event-triggered, or hybrid architecture. In case of an event-triggered adaptation, MyCCM-HI implements one or the other of the synchronous mode switches presented in section 8.3.3. In order to apply the desired protocol, MyCCM-HI relies on the configuration provided by the software designer (using the COAL).

Beyond this flexibility, MyCCM-HI helps to manage the adaptation loops in a hierarchical manner from the very beginning of the system/software specification.

	Giotto	COMDES/xGiotto	MyCCM-HI (time-triggered)	MyCCM-HI (event-triggered)
Reactivity	---	--	-	++
Determinism	+++	++	+	---

Table 8.1. *Comparison of design technologies dedicated to ADRES*

8.4.3. *How to choose of the appropriate technology*

As stated at the beginning of this chapter, the design choices of an ADRES will vary drastically from one system to another one. In section 8.2.2, we presented these choices with respect to the autonomy/predictability trade-off. Here, it can be seen that the choice of one or other framework will be led by a reactivity/determinism trade-off. Table 8.1 lists the characteristics of the considered technologies with regards to this trade-off.

To illustrate the different aspects presented at the start of this chapter, we will apply one of these MDE technologies to a specific example of ADRES. In order to cover both the (i) hierarchical aspects of the adaptation loops management, (ii) the configuration of the mode switch protocols, and (iii) the automatic implementation of reconfiguration actions, we have used MyCCM-HI. More precisely, we rely on MyCCM-HI using event-triggered mode switches in order to illustrate the usage of mode switch protocols as described in 8.3.3. Before entering the details of the design based on this technology, we present in the next chapter the case under consideration.

8.5. An example of adaptative system from the robotic domain

We present in this section a mobile robot designed to explore a disused tunnel that requires renovated for future use. In order to evaluate the state of the tunnel, the robot sends images of the tunnel via a camera installed in it. This video stream is also used by the user to guide the robot inside the tunnel. As the robot travels along in the tunnel, the bandwidth of the wifi-connection between the robot and the operator goes down and the quality of the video stream is not sufficient to drive the robot. It is then necessary to use the robot in an automatic mode: the robot determines the direction to follow to join predefined points, and send pictures of the tunnel to the user upon requests from the latter.

8.5.1. *The pilot system*

The example we focus on is more precisely the piloting system of the robot. This system is compound of two subsystems; a localization and a navigation subsystem. In automatic mode, the localization subsystem is active and sends the current position of the robot to the navigation subsystem. Once it has received several data corresponding to the robot's position, the navigation subsystem computes the robot's current speed and the command sent to the wheels controllers in order to maintain the desired speed.

In order to design each system and subsystem independently, modes are associated to each level of specification: the pilot system may operate in automatic or manual mode, the localization system may be activated or deactivated, and the navigation system may operate in automatic or manual mode.

8.5.2. *Avoid mode inconsistencies*

Suppose the localization subsystem is activated while the navigation subsystem operates in manual mode: the localization subsystem sends position information to the navigation subsystem that will not be used by this system until it goes back to automatic mode. Receiving these data while the navigation subsystem is in manual mode constitutes a perturbation that may lead to faults (CPU overload, buffer overflow, etc.). Consequently, these two modes (activated and manual) are *strongly inconsistent*: they must never be simultaneously active.

Conversely, suppose the localization subsystem is deactivated while the navigation subsystem operates in automatic mode. In this case, the direction is automatically determined without updates of the robot's position. Depending on the maximal speed of the robot, this situation is acceptable during a fixed delay. Consequently, these two modes (automatic and deactivated) are *weakly inconsistent* modes: they can be simultaneously active, but only during a fixed delay.

8.6. Applying MDE techniques to the design of the robotic use-case

In order to illustrate in more details the importance of the design of the ADRESs, we describe in this section the specification of the robot using MyCCM-HI. Thus, we present in this section the main modeling artifacts used by the COAL in order to describe an ADRES. These artifacts are:

– generic software components, to model the functional breakdown of the systems' functionalities;

– real-time dedicated components to represent the real-time characteristics of the software architecture;

– operational modes and associated (re)configurations to specify the pseudo-dynamic reconfiguration.

8.6.1. *Functional breakdown*

A generic software component is basically a functionality that can be assembled with others to provide the full functional coverage of the system. The description of these passive components allows the functional part of the system to be broken down into smaller, independently manageable pieces that can be configured and assembled by an external application. To achieve this, generic components provide the explicit description, by means of ports, of the services that the component provides and requires during its execution. It also defines some configuration attributes in order to tune the component functions of a particular utilization context.

Thus, generic components make the deployment and configuration of the application by another application easier. Following the same principles, component configuration and deployment can also be modified by an external application (thus realising the reconfiguration).

To describe generic software components, COAL relies on the standard Lightweight CCM[1] (standardized by the OMG[2]). This choice was led by the industrial context in which MyCCM-HI was developed. These passive components provide ports that are either facets/receptacles (to implement synchronous operation calls), or event sinks or sources (to implement asynchronous message passing) through which they can be activated.

A facet represents a provided interface (i.e. a provided set of functionalities) while a receptacle represents a required interface. Thus, when a component's receptacle is connected to another component's facet, the functionalities provided by this facet might be called through the connected receptacle. It is inside the code implementing components' functionalities that we can find out when a receptacle is actually used. Lightweight CCM also enables configuration attributes to be defined in order to configure the function of a component regarding its usage context. Similarly, an event connection enables a component to emit messages through a source towards one or several sinks.

Figure 8.8 is an illustration of a Lightweight CCM component defining one facet (get_pos), one receptacle (compute_direction), one event sink (cur_pos) and one configuration attribute (size). The behavior of this component is the following:

– when receiving position data on cur_pos, the component stores it in an array, the size of which is defined by the configuration attribute. Once this array is full, the component calls the component connected to compute_direction;

1. Light-weight CORBA component model revised submission, OMG document realtime/03-05-05 edn
2. Object Management Group

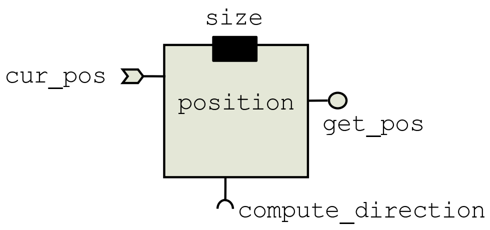


Figure 8.8. *Lightweight CCM example*

– when receiving a get_pos request, the component immediately returns the array of positions.

8.6.2. *Real-time systems specificities*

In the realm of real-time systems, domain specific languages (such as AADL) have proven their utility in automatic configuration and analysis of critical systems [LAS 09]. These languages aim to represent both domain-specific software components (tasks, partitions, shared data, etc.), domain-specific hardware components (processors, memory, buses, etc.), and their specific characteristics (scheduling policy, network bandwidth, processor speed, etc.). Real-time systems specific languages thus enablethe deployment of software components onto hardware components to be modeled, as well as the specific configuration of each of these components. Using such languages makes the analysis and synthesis of the whole system easier.

Inspired by AADL, COAL enriches a Lightweight CCM specification due to the real-time system characteristics such as periodic or sporadic activities. Configuring the facet of a component instance, a periodic activity will activate this component with the period and priority parameters provided by the software designer. Configuring an event sink of a component instance, a sporadic activity will trigger the corresponding component when receiving a message, and two consecutive messages will be treated with a minimal period of time separating the treatments.

Figure 8.9 represents the usage of activities in a very simple COAL specification: in this figure, triangles represent activities that will activate components odometer and position. While associated with a facet, an activity is periodic and calls the activation operation of this facet every time the period of time configuring the activity has elapsed. Thus the facet activation will be activated every 10 ms. While associated to an event sink, an activity is sporadic. Thus, current_pos will be activated when receiving a message on port current_pos, given that at least 10 ms have passed between the reception of two messages.

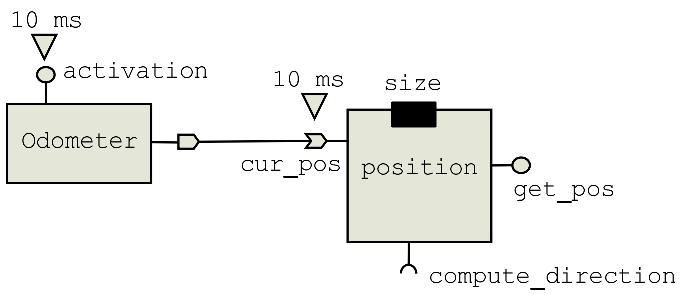


Figure 8.9. *Activities definition in COAL*

8.6.3. *Pseudo-dynamic reconfiguration*

In order to model the pseudo-dynamic reconfiguration of an ADRES, the COAL represents:

– the system and subsystems operational modes;

– the mode transitions logics;

– the software configuration associated with each of these modes.

We describe hereafter the corresponding artifacts available in the COAL.

Operational modes

The COAL models the operational mode of a system in a dedicated component called "mode automaton". A mode automaton defines the operational modes of the system (or subsystem) it is contained in.

Regarding the robot we are designing, two operational modes have been identified: automatic and manual. We have also identified two subsystems, localization_system and navigation_system, which provide two operational modes: automatic/manual for the navigation_system and on/off for the localization_system.

The mode switch logics

Beyond the definition of operational modes, a mode automaton describes the mode transitions of the system. A mode transition is described by:

– one source mode;

– a set of conditions (received messages, timing guards, configuration attributes of the automaton);

– a set of actions (send messages, call software components interfaces);

– a target mode.

If different transitions share the same source mode, their relative priority must be defined (the automaton is thus deterministic)

In the robot, in order to guarantee that strongly inconsistent modes are never simultaneously active, we define a mode switch sequence, from manual to automatic, that first activates the navigation subsystem in the automatic mode and then activates the localization subsystem. To design this behavior, we add intermediate modes to the operational modes of the pilot system. These intermediate modes will ensure the synchronization between the modes of the system and its subsystems in order to avoid inconsistent modes.

Figure 8.10 illustrates the modes and mode transitions of our pilot system and its two subsystems. In this figure, modes are represented by states while transitions are represented by labeled arrows. Italic labels represent conditions on a mode transition and Bold labels represent actions that are executed once the target mode of the transition is reached.

Following the transition from manual to automatic in the pilot_system mode automaton, we have the following three transitions:

1) when receiving the mode switch request from manual to automatic (*Pilot_A?*), the transition to the mode *MtoA1* is activated. Reaching *MtoA1*, a mode change request (**Nav_A!**) is sent to the navigation subsystem. This way, an activation request is sent to the navigation subsystem while the localization is still stopped;

2) when receiving the confirmation that the navigation is activated (*Nav_A?*), the pilot system goes to *MtoA2* and requests the activation of the localization subsystem (**Loc_on!**). Thus we ensure that the navigation system is started before the localization subsystem starts, avoiding strong inconsistency between the modes of the two subsystems;

3) finally, when receiving the acknowledgment that the localization subsystem has started, the automatic mode is reached.

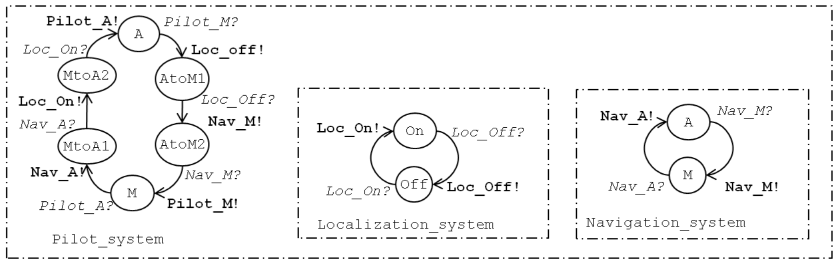


Figure 8.10. *Modes of the pilot system and its subsystems*

Another condition of the transitions between manual and automatic modes in the navigation subsystem is that the robot's speed is null. For commodity reasons in the presentation of the mode automata, we did not represent the corresponding modes and mode transitions.

Software configuration associated to each mode.

Similarly to the AADL, COAL associates some of the configuration artifacts of the software architecture with a set of operational modes. Thus, those configuration artifacts are valid only if the system is in one of these modes. For instance, when defining a connection in the model, the software architect can associate this connection to a set of modes thanks to the keywords **in modes**. As a consequence, the corresponding ports will be connected when entering one of these modes and disconnected when entering a mode that is not part of the defined set.

Regarding the definition of an adaptation process provided at the beginning of this chapter, the above sections illustrate the *analysis* and *planning* specification using COAL: the analysis is specified due to the mode transition logics included in the mode automata definition, while the planning is deduced from the difference between the software configuration associated to the source mode and the software configuration associated to the target mode of a transition.

Figure 8.11 synthesizes the system and software architecture of the piloting system.

System architecture. System boundaries are drawn with dashed boxes: the piloting system is comprised of two subsystems. A localization subsystem, that provides the current position of the system every 10 ms, and a navigation subsystem whose behavior is different according to its current mode: in automatic mode, the navigation subsystem computes the guidance commands when receiving the current position of the system from the localization subsystem; in manual mode, the navigation computes the guidance commands from orders of an end-user of the vehicle.

In figure 8.11, three mode automata are represented (Pilot_supervisor, Nav_supervisor, and Loc_supervisor). Those automata come with event ports that enable them to receive mode switch orders, and to send mode switch status. The mode switch specification of these automata corresponds to the definition illustrated in figure 8.10.

Software architecture. In figure 8.11, we also represented the software architecture of the different applications of this system. The behavior of the software application is:

– in automatic mode, the guidance activity triggers the position component every 100 ms to retrieve the last 10 position of the system, compute the current speed of the system, and store these data in the speed component (by invoking the connected facet).

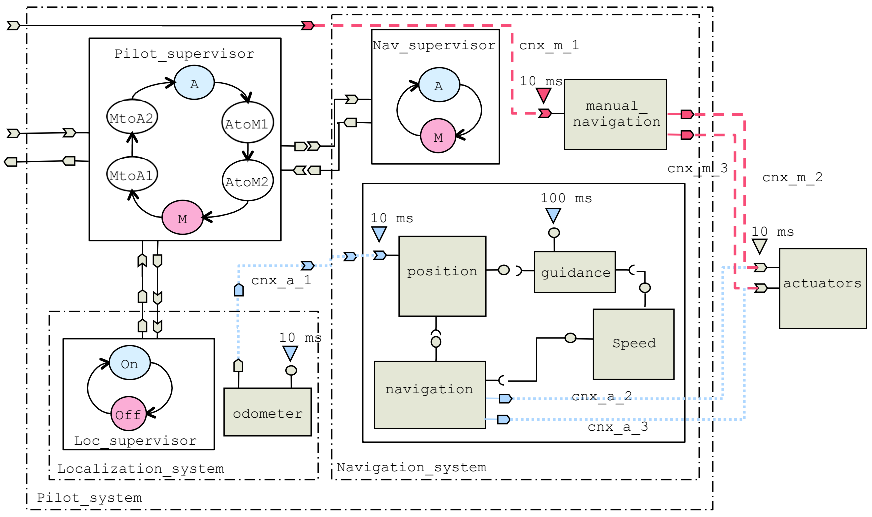


Figure 8.11. *System and software architecture of the pilot system*

When invoking the position component to retrieve the last positions, the connection between the component's position and navigation is not invoked, which is consistent with the implementation of the position component given in section 8.6.1. Conversely, when the navigation component is triggered by the position activity (sporadic 10 ms), it reads the current speed of the system, computes, and sends the commands to the actuators.

– in manual mode, the guidance activity does not trigger any component, and the commands are sent to the robot from the manual_navigation component.

To model these different behaviors in our specification of the pilot system, connections between components are either valid in a set of modes, or in every mode. In figure 8.11, connections cnx_a_1 is only valid when the localization_supervisor is in mode On; cnx_a_2, and cnx_a_3 are only valid when navigation_supervisor is in automatic mode. Similarly, cnx_m_1 is only valid when Pilot_system is in manual mode, and cnx_m_2, and cnx_m_3 are only valid when Navigation_supervisor is in manual mode. Thus, the current position of the system is sent to the navigation subsystem only when the localization subsystem is in on mode, and depending on the mode of the navigation subsystem, the commands are sent to the wheels either by the *manual_navigation* component or by the *navigation* component.

Mode switch protocol configuration

Finally, we describe hereafter how to specify the mode switch protocol associated with a mode transition. Among the mode switch protocols presented in section 8.3.3, a software description based on COAL can lead to the use of one of three protocols:

– idle time protocol, which is instantiated by default if the COAL specification does not require the usage of another protocol;

– single offset protocol with periodicity if the COAL specification states that the port triggering this transition must be activated in “emergency” (this keyword being associated to the considered port);

– sampling preserving mode switch protocol for periodic tasks identified as “synchronized” in a COAL model.

Guidelines regarding how to choose the mode switch protocol will be given in next section, in which the exploitation of the model (in terms of code generation for mode switch mechanisms) is also presented.

8.7. Exploitation of the models

COAL models are interpreted by an open-source component framework called MyCCM-HI. The main feature of MyCCM-HI is to generate code led by adaptation requirements as explained in the previous section. We explain the corresponding code generation patterns in this section, and present experimental results that help to comparing the different mode switch protocols.

Before we continue, we need to introduce an important principle with respect to the remainder of this section: we call the “impacted task” a task for which the control flow depends on the value of the current mode of the system.

8.7.1. *Code generation: the pseudo-dynamic execution step*

The code generation process proposed by MyCCM-HI implements the execution activity of an adaptive system (according to the definition given in section 8.2.1) in a pseudo-dynamic fashion: configurations and reconfigurations are known at the design stage, they are generated in the code of a software application to handle adaptations during runtime. The objective of this approach is to ensure that reconfiguration mechanisms are safe, i.e.:

– associated with a synchronous mode switch protocol; synchronous here refers to the property that no mode switch occur while an impacted task is executing;

– conforming to the safety requirement specified with COAL: schedulable mode switch; fast mode switch; mode switch preserving data sampling consistency;

– respecting frequent development restrictions when it comes to ADRES: dynamic memory management is prohibited in order to improve the possibilities of analysis of the produced code.

Mode switch mechanism

To implement a solution to this problem, we generate and configure code according to the principles presented hereafter: because it is read by impacted tasks and written by tasks implementing the mode switch logic, the current mode of a system is a shared variable that we protect with a read/write lock mechanism. The code of the automaton locks this resource as a writer in order to modify the value of the system's current mode. The following execution pattern is generated for impacted task in order to ensure a synchronous mode switch:

– at the beginning of its execution, an impacted tasks locks the resource as a reader;

– then it performs the computations associated with the current mode of the system;

– finally, the task releases the lock at the end of its execution.

These principles ensure a synchronous mode switch protocol as the mode switch cannot occur while an impacted task is running: running impacted tasks locks the resource and the mode switch is blocked until all these tasks have finished executing.

The main advantage of this implementation is that it is easily configurable to adapt the mode switch protocol used. For instance, when the single offset protocol is used, the task implementing the mode switch is configured with a higher priority than the highest priority of impacted tasks and the lock is configured with the immediate ceiling priority protocol (a well-known solution in real-time systems to reduce the number of preemptions, avoid deadlocks, etc.).

Reconfiguration

In a component-based model, basic reconfiguration actions are instantiation/deletion of components, connection/disconnection of components, creation/deletion of tasks, activation/deactivation of tasks, data initialization, and connection/disconnection of components.

The reconfiguration actions provided by MyCCM-HI are:

– the connection/disconnection of software components;

– the update of values of the software components attributes;

– the activation/deactivation of activities.

One limitation of this set of actions is related to a frequent development restriction when it comes to ADRES: the dynamic memory management is prohibited. This impedes the dynamic instantiation/deletion of activities or components to improve the possibilities for analysis of the produced binary code.

In MyCCM-HI, the execution step of the adaptation control loop is thus very simple, as it consists of (i) updating the value of the current mode, (ii) updating the software component attributes (according to the difference between the source mode configuration and the target mode configuration), and (iii) invoking the components interfaces as specified in the corresponding actions of the mode transition. Avoiding dynamic memory management, a disconnection is implemented as a connection switch that, depending on the current mode value, invokes the component connected to this mode (or raises an exception if the component is really disconnected).

8.7.2. *The choice of the mode switch policy*

The choice of the mode switch will be led by the following considerations:

– Question 1: in the system does a data sampling require a specific mode transition protocol?

– Question 2: is the promptness of the adaptation more important than the periodicity?

Depending on the answers to these questions, different mode switch protocols will be chosen.

First, it is interesting to note that these two questions are loosely coupled: imagine that a system requires a specific mode transition because of data sampling as stated by question 1. This means that the mode switch will begin when a set of synchronized tasks have reached its hyper-period. It is still possible that other tasks of the system are impacted by the mode switch. Thus, the software architect needs to answer to the second question concerning those non-synchronized but impacted tasks.

Then, if it is easy to answer the second question (a choice between rapidity and periodicity), it is more difficult to answer the first. We note here the conditions that require the use of a specific mode transition protocol because of a data sampling:

– a set of data is sampled (produced and read at different paces);

– data produced in the source mode are substantially different from the data produced in the target mode;

– the absence of updated data over one hyper-period must never occur as it can lead an unrecoverable failure of the system.

Answering yes to the first question implies that the sampling-based protocol must be used. In our robot example, the sample data are the set of positions that are used to compute the current speed of the robot. Thus, the answer to the first question depends on the specification of the weakly incompatible modes. If the time bound associated with the simultaneous activation of the stop mode of the localization subsystem and the automatic mode of the navigation subsystem is inferior to the considered hyper-period, then the sample-based protocol should be used. Otherwise, it is obvious that the navigation can work without updated position data during at least one hyper-period.

In order to help to choose the mode switch protocol, Table 8.2 summarizes the characteristics of the protocols with regards to three important properties: periodicity, rapidity, sampling preservation.

Criteria / Protocol	P1	P2	P3	P4
Periodicity	+++	-	+++	+
Rapidity	-	+++	-	+
Sampling preserving	++	++	+++	+++

P1 : Idle time protocol without sampling
P2 : Single offset without sampling
P3 : Idle time protocol with Sampling
P4 : Single offset with sampling

Table 8.2. *Comparison of the mode switch protocols implemented by MyCCM-HI*

8.7.3. *Illustration of properties of the mode switch protocols*

In order to show the impact of the choice of protocol on the rapidity of the adaptation, we have designed and implemented the robot example presented in this chapter. Using MyCCM-HI, we have configured the mode switch protocol with each of the four possibilities presented in previous section.

We have evaluated the reconfiguration time obtained functions of the chosen policy and the complexity of functional tasks. To simulate this complexity, we implemented active waits in the functional code, with increasing time lengths. This time constitutes a simulation of the CPU occupation time.

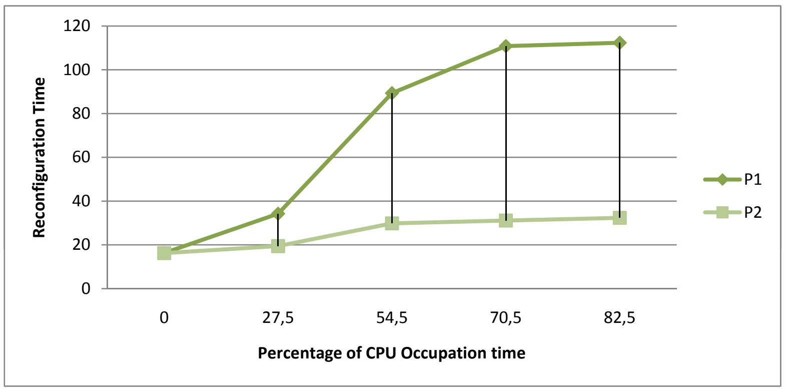


Figure 8.12. *Measured reconfiguration time, functions of the policy (with synchrony) and the simulated CPU load*

Figure 8.12 represents the reconfiguration time obtained using P1 and P2, functions of the simulated CPU occupation time. These results show clearly that the reconfiguration time is smaller using P2 than using P1; this being all the more true when the CPU occupation is high, which means that the functionalities implemented are complex. A similar result has been obtained comparing P3 and P4: P4 gives a shorter reconfiguration time than P3.

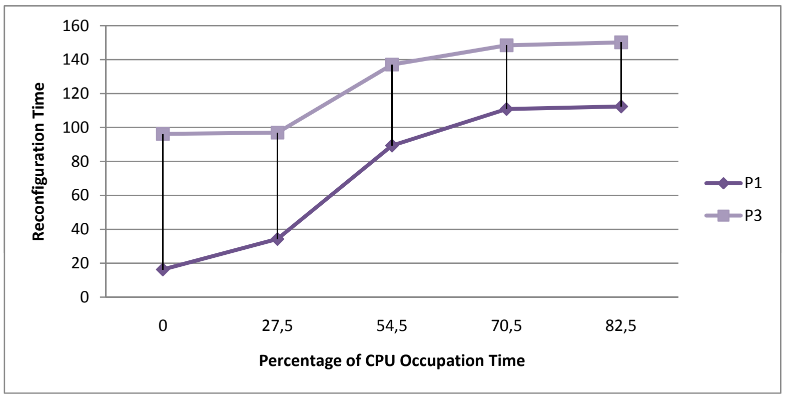


Figure 8.13. *Measured reconfiguration time, functions of the policy (with synchrony) and the simulated CPU load*

Figure 8.13 compares the reconfiguration time obtained using P1 and P3 as functions of the simulated CPU occupation time. In this figure, we detect a gap between the two curves due to the wait for the synchronized tasks to have reached their hyperperiod.

The results presented in Figures 8.12 and 8.13 validate comparison of mode switch rapidity stated in Table 8.2.

8.8. Conclusion

In this chapter, we have presented the principles underlying ADRES design. Beyond the design principles dedicated to such systems, we have presented different technical solutions that help in configuring and analysing such systems. We have provided an evaluation of these solutions, which stated that they mainly pursued different types of applications (depending on the safety level required by the system under design). We then chose a concrete example from the robotics domain, and used one of the previously described technologies to realise its design. As a conclusion, we discussed the reasons that led to the choice of one or the other of the adaptation mechanisms.

The field of ADRESs is broad, and raises many problems, solutions, and possible discussions. This is part of the additional complexity described at the beginning of this chapter. Some interesting aspects have already been addressed in this chapter. One of the most important considerations is the link between adaptation and fault tolerance: adaptation can be a recovery solution to faults, but also requires it to be fault tolerant.

8.9. Bibliography

[ARM 03] ARMSTRONG J., TO H., Making reliable distributed systems in the presence of software errors, PhD thesis, The Royal Institute of Technology (KTH), 2003.

[BOI 00] BOINOT P., MARLET R., NOYE J., MULLER G., CONSEL C., "A declarative approach for designing and developing adaptive components", *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, Washington, DC, USA, IEEE Computer Society, p. 111–, 2000.

[BOR 09] BORDE E., FEILER P. H., HAÏK G., PAUTET L., "Model driven code generation for critical and adaptative embedded systems", *SIGBED Review*, vol. 6, 2009.

[BRI 05] BRINKSCHULTE U., SCHNEIDER E., PICIOROAGA F., "Dynamic real-time reconfiguration in distributed systems: timing issues and solutions", *ISORC*, p. 174-181, 2005.

[BRO 03] BROOKS R., "A robust layered control system for a mobile robot", *Robotics and Automation, IEEE Journal of*, vol. 2, p. 14–23, January 2003.

[CHA 01] CHAPIN N., HALE J. E., KHAM K. M., RAMIL J. F., TAN W.-G., "Types of software evolution and software maintenance", *Journal of Software Maintenance*, vol. 13, p. 3–30, January 2001.

[COM 06] COMPUTING A., "An architectural blueprint for autonomic computing.", *white paper*, vol. 36, num. June, Page 34, IBM Corp., 2006.

[DAI 06] DAI Y., MARSHALL T., GUAN X., "Autonomic and dependable computing: moving towards a model-driven approach", *Journal of Computer Science*, vol. 2, p. 496-504, 2006.

[DAV 09] DAVID P.-C., LEDOUX T., LÉGER M., COUPAYE T., "FPath and FScript: language support for navigation and reliable reconfiguration of Fractal architectures", *Annales des Télécommunications*, vol. 64, p. 45-63, 2009.

[FAR 06] FARCAS E., Scheduling multi-mode real-time distributed components, PhD thesis, University of Salzburg, Austria, 2006.

[FEI 06] FEILER P. H., GLUCH D. P., HUDAK J. J., The architecture analysis & design language (AADL): an introduction, Report , Software Engineering Institute, 2006.

[GAN 03] GANEK A. G., CORBI T. A., "The dawning of the autonomic computing era", *IBM Systems Journal*, vol. 42, p. 5–18, January 2003.

[GHO 04] GHOSAL A., HENZINGER T. A., KIRSCH C. M., SANVIDO M. A. A., "Event-driven programming with logical execution times", *Proc. of HSCC 2004, Lecture Notes in Computer Science*, p. 357–371, 2004.

[GRO 06] GRONDIN G., BOURAQADI N., VERCOUTER L., "MaDcAr: an abstract model for dynamic and automatic (re-)assembling of component-based applications", *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2006)*, vol. 4063 of *LNCS*, Västeras, Sweden, Springer-Verlag, p. 360-367, June 2006.

[HEN 01] HENZINGER T. A., HOROWITZ B., KIRSCH C. M., "Embedded control systems development with Giotto", *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, LCTES '01, New York, NY, ACM, p. 64–72, 2001.

[HOF 93] "Surgeon: a packager for dynamically reconfigurable distributed applications", *Software Engineering Journal*, vol. 8, p. 95-101, 1993.

[IVA 99] IVAN-ROSU D., Dynamic resource allocation for adaptive real-time applications, PhD thesis, Atlanta, GA, USA, 1999, AAI9966960.

[KE 07] KE X., SIERSZECKI K., ANGELOV C., "COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems", *RTCSA*, IEEE Computer Society, p. 199–208, 2007.

[KRA 90] KRAMER J., MAGEE J., "The evolving philosophers problem: dynamic change management.", *IEEE Transactions on Software Engineering*, vol. 16, p. 1293–1306, 1990.

[LAS 09] LASNIER G., ZALILA B., PAUTET L., HUGUES J., "OCARINA: an environment for AADL models analysis and automatic code generation for high integrity applications", *Reliable Software Technologies'09 - Ada Europe*, Brest, France, June 2009.

[MAG 95] MAGEE J., DULAY N., EISENBACH S., KRAMER J., "Specifying distributed software architectures", *Proceedings of the 5th European Software Engineering Conference*, London, UK, Springer-Verlag, p. 137–153, 1995.

[ORE 99] OREIZY P., GORLICK M. M., TAYLOR R. N., HEIMBIGNER D., JOHNSON G., MEDVIDOVIC N., QUILICI A., ROSENBLUM D. S., WOLF A. L., "An architecture-based approach to self-adaptive software", *IEEE Intelligent Systems*, vol. 14, p. 54–62, May 1999.

[PED 98] PEDRO P., BURNS A., "Schedulability analysis for mode changes in flexible real-time systems", *ECRTS*, IEEE Computer Society, p. 172–179, 1998.

[REA 04] REAL J., CRESPO A., "Mode change protocols for real-time systems: a survey and a new proposal", *Real-Time Systems*, vol. 26, num. 2, p. 161–197, 2004.

[ROY 07] ROY P. V., "Self Management and the Future of Software Design", *Electronic Notes in Theoretical Computer Science*, vol. 182, p. 201–217, 2007.

[SHA 89] SHA L., SHA L., RAJKUMAR R., RAJKUMAR R., LEHOCZKY J., LEHOCZKY J., RAMAMRITHAM K., RAMAMRITHAM K., "Mode change protocols for priority-driven preemptive scheduling", *Real-Time Systems*, vol. 1, p. 243–264, 1989.

[TIN 92] TINDELL K. W., BURNS A., WELLINGS A. J., "Mode changes in priority preemptively scheduled systems", *Proceedings of the Real-Time Systems Symposium*, p. 100–109, 1992.

[VAN 06] VANDEWOUDE Y., EBRAERT P., BERBERS Y., D'HONDT T., "An alternative to Quiescence: Tranquility", *ICSM*, IEEE Computer Society, p. 73–82, 2006.

[WOL 05] WOLF T. D., SAMAEY G., HOLVOET T., ROOSE D., "Decentralised autonomic computing: analysing self-organising emergent behaviour using advanced numerical methods", *ICAC*, IEEE Computer Society, p. 52–63, 2005.

Chapter 9

The Design of Aerospace Systems

9.1. Introduction

In the past 20 years, space systems have assumed a tremendous importance in our lives. There are hundreds of satellites above our heads, which provide various services that we now continuously rely on: navigation, weather forecasting, television, telecommunications, etc. Earth is monitored by ultra-precise sensors, while deep space is explored by robots, with the hope of helping us to understand physical phenomena that might be at the origin of human life. Exploring space is somehow similar to the long boat travels that were undertaken by our ancestors to discover the limits of our current world, and this is what is so exciting about it. Today we hear enthusiast scientists talking about traveling back to the Moon or even to Mars, hoping that such objectives, even if they are not easy to justify, will produce unexpected side effects.

The point about exploration is that it is the perfect tool for discovering paths that are not necessarily the ones we expected. For example, in the past space research led us to discover important innovations in many fields that are now of great use on Earth: solar panels, textiles, medical instruments, or recycling techniques. The fact that these outcomes are sometimes not foreseen motivates scientists to propose new ambitious missions that can give them the freedom required to achieve any major creation. They have to go beyond cultural and sometimes financial habits, also beyond limits that dominant thoughts impose in order to find a new revolutionary technology that will make our technical, intellectual, or scientific knowledge go a step further.

Chapter written by Maxime PERROTIN, Julien DELANGE, and Jérôme HUGUES.

Distributed Systems: Design and Algorithms Edited by Serge Haddad, Fabrice Kordon, Laurent Pautet and Laure Petrucci
© 2011 ISTE Ltd. Published 2011 by ISTE Ltd.

9.1.1. *About space systems complexity*

The space system designers today master the combination of domains as complex as mechanics, thermal, energy storage, and management, radio communications, etc. Very precise mathematical and physical models are used in order to define the trajectory of spacecraft, or to predict the effects of the hostile space environment on embedded electronic boxes. However, on-board autonomy is still difficult to master using current technology. Space systems must remain simple because software complexity is just too hard to validate with the current development methods due to constraints enforced, potential hazards, and safety/security issues.

It took us hundreds of years to master the wheel up to a point where they are even embedded as means to correct the attitude of satellites; but we only recently found out the techniques to program and automatically control the behavior of a machine. Most terrestrial machines use software; computers are everywhere even in coffee machines, yet many of them are simply unreliable. Home computers frequently crash, and no technique other than end-user validation has been found to improve the overall reliability of software. Users do not really complain about this situation, which, therefore, has not significantly evolved in the past decade. What is more dangerous, car software tends to crash more frequently, and for some reason users complain more about it.

Handling complexity poses problems, that is obvious, and as there are no real mathematical models or scientific approaches available to develop better software, it is hard to see how we can do better. The limit of this approach is reached when dealing with really expensive and critical systems: nuclear power plants, aircraft, spacecraft, and launchers. For these systems it is not an option to blindly develop software and execute it on target, praying for correct execution at the first attempt. Everything must be foreseen and deterministic; similar to a bridge when the first truck goes over it: we cannot tolerate any dramatic events.

A new trend in space systems is to move from big centralized satellites to smaller, but distributed systems: we call that formation flying, because the idea is to let several satellites achieve complex missions by combining the capabilities of payloads when they are distributed over several physical spacecrafts. For example, the Proba 3 mission will make use of two satellites to study sun coronas. The challenge is big, and will address all the questions that are raised when designing a distributed system: how to take decisions, how to detect faults, how to elect a master when none of the nodes has complete knowledge of the state of the entire system.

To solve these issues, a fail-proof software development process must exist and be supported by tools, so that the system designer can concentrate on what is really new and challenging: the distributed algorithms.

9.1.2. *Aim of this chapter*

This chapter explains how the space sector in Europe has tackled the problem of defining a proper development process, and what tools have been developed to deal with the specificities of embedded, heterogeneous systems. We first present the main characteristics of satellite systems and software, then the issues that are faced when developing them, what can be improved, and the solutions developed in the scope of the TASTE project to ease the development, integration, and validation of on-board software (OBSW).

9.2. Flight software typical architecture

Most satellites have a simple dual structure: they consist of a platform, and one or several payloads. The platform includes software used to control the satellite, while the payloads perform the mission itself, such as taking pictures of the Earth, acquiring values through sensors, etc.

In this section we present the platform software; it is the part that is developed by the space industry, in collaboration with a system team and software experts. What we call "system" combines both the flight and ground components that are needed to fulfill a specific space mission. The system team is a set of people who have an overview of the complete mission, and who know what each component is supposed to do. The platform software can be seen as an essential subsystem that is responsible for the control of the spacecraft when it is launched.

We describe the various applications that are part of this OBSW, and discuss real-time aspects, emphasizing the enforcement of timing constraints (for example, data must be sent before a precise and strict deadline). We write it as a support to understand the mechanisms that we have put in place to improve the development of OBSW ensuring their reliability, robustness, and correctness.

9.2.1. *OBSW applications*

Basically, an OBSW comprises several major applications:

– The **altitude and orbit control subsystem** (AOCS) controls the position of the satellite in space. It implements control laws that are established from mathematical and physical models, following Kepler's laws on the movement of objects in space. One of these laws describes the theoretical trajectory of a satellite orbiting around the Earth.
In practice, the movement of a satellite is affected by its environment so that it is necessary to periodically compute and adapt its trajectory. In that purpose, the AOCS reads information from sensors and computes commands sent to actuators. This part of the AOCS is cyclic, executed according to a fixed period;

– The **guidance and navigation control software** (GNC) performs mission-specific maneuvers. Most of the time, GNC designers skills are more specialized in automatics rather in software. For that reason, these functions are designed using software such as Matlab-Simulink or SCADE to specify and validate their algorithm;

– The **mode manager** controls the satellite operational modes and data, depending on the mission phase. There are usually a limited number of modes at system level, that are further refined in sub-modes for each application. For example, there can be an operational mode, a maintenance mode, a safe mode (in case of on-board failure), a launcher mode corresponding to the satellite while it is still in the rocket, etc. Modes are there to characterize a particular satellite state, and the transition from one mode to another can be either done autonomously, or manually on reception of a ground (earth operator) order;

– The **fault detection, identification and recovery** (FDIR) detects potential failures and possibly recovers them when occurring: This is a central application that implements algorithms that depend on the mission. For example, a geostationary satellite is always visible from the Earth; operation centers can at any time determine the status of the spacecraft equipments, and possibly send data allowing a reconfiguration of failed units; the need for autonomy is limited, and the FDIR function becomes simple, which saves on-board resources and validation effort. In the case of a deep space exploration mission, on the other hand, it might be the case that for several days the satellite is not visible to the Earth, and even if it is, it can take a very long time to communicate with it. In that case a more complex FDIR function can be developed;

– The **power, thermal and communication management** handles power regulation of satellite equipment, thermal control (temperature is within expected limits) and sends or receives network packets to or from the Earth. Packets received from the Earth are called *telecommands* (TC) while those sent to the Earth are called *telemetries* (TM). Today most satellite systems are “centralized”, and communication is limited to the exchange of single messages between Earth and space. However, in newly-designed space systems, this is evolving towards inter-satellite communications to support formation flying systems.

9.2.2. *Applications support environment*

OBSW applications are executed on top of an specific operating system that has to respect several constraints: precise timing, low memory footprint, analyzability of functions, etc. In the context of space-related systems in Europe, RTEMS is commonly used. It fulfills the requirements described above and also supports space-specific execution environment (dedicated processors and transport buses such as LEON2 or SpaceWire).

OBSW applications also require several services to communicate with the satellite environment: store data in mass memory, send/receive packets, operate satellite manually, update software, etc. These services are provided by a specific standardized middleware called the *Packet Utilization Standard* (PUS). Several implementations exist, most of them are available as commercial products.

9.2.3. *Execution patterns of space software*

The real-time aspects of most OBSW are rather simple: most functions are cyclic, executed according to a fixed period. Some are sporadic (such as reception of telecommand from Earth) but can be simplified as a cyclic activity: in fact, sporadic activity occurs according to a minimal inter-arrival time and the maximum waiting time can also be deduced (with pessimist approaches). This way, even sporadic tasks become to be cyclic. According to the current state of the art in scheduling analysis, it is considered as the simplest way to analyze the feasibility of a system.

9.3. Traditional development methods and their limits

The traditional development cycle of a space OBSW is a V-cycle [HOF 97]. It contains phases for the specification, design, implementation, and testing of the product. At the end of each phase, an independent review is conducted by the customer who will authorize the next phase to start or not. Review requirements are specified in standards that depend on the system operating domain. In the context of space systems, the ECSS standards are applicable (more specifically, ECSS-E-40 [ESA 03]series for software aspects). The main purpose is to ensure that requirements are enforced as expected. For that, developers and designers have to document, test and assess what they produce.

This design process and review activities are conducted for both software and system levels but each activity is loosely coupled. Moreover, as software aspects need inputs from the system design, it is generally delayed, waiting inputs from system designers and successful reviews. It results in a complex and hard to schedule design process, which rarely goes to market at the expected date.

9.3.1. *Traditional specification methods*

In past projects, specifications were traditionally written as a list of textual requirements, while design was actually the code itself; there exists no shared view on how a software design shall be documented, and lack of guidelines and tools has always made this phase very difficult for suppliers to interpret.

As a result, verification of the software was achieved by manual reviews of the code according to its textual specification. As this activity was not done automatically and inputs were not formalized, it led to an error-prone process so that some requirements and important aspects of the systems were not correctly reviewed.

On testing aspects, tests have always been long and expensive partly because of a difficult mastering the overall design, which is poorly documented. As a result, the success of many space projects was mainly due to the experience and talent of a few people, more than the application of scientific, rigorous, and reproducible development methods.

9.3.2. *Need for an engineering process*

Recently, satellite complexity has increased together with new, more powerful hardware (computing capacity, memory dimensioning, etc.) without considering the potential issues it could introduce. This resulted in mission failures, some rockets exploded due to incorrect software. In fact, the traditional approach that relies on good programmers reached its limits and it was impossible for a single person (or even a few people) to design a complete software system, even with high skills: this indicated that a more rigorous development process was needed.

These observations led into the space community in Europe to put some efforts into trying to precisely define the requirements and develop the means to improve the way we develop software. One of the first significant projects that triggered this process was called ASSERT [1] and it addressed the following subjects:

– Reference architectures: identification of common characteristics of several systems: needs in terms of reliability, availability, safety, etc. The idea came from the fact that today, the avionics architecture (redundant computers, choice of communication means, but also software real-time architecture) is redefined for each new system, manually from past experiences. The objective is therefore to define a systematic method to begin with the requirements and to deduce the most adapted architecture that fulfills these needs;

– logical software architecture description by capturing software functions with its interfaces (sent/received data). Designers can describe their software functions/blocks and validate them early in the development process;

– hardware architecture description that describes the environment run-time of the system: processor boards, memory, network buses, etc.;

– definition of an execution run-time that supports system functions execution on the hardware architecture. It consists of a middleware executed on top of the operating system which provides generic services to software functions, such as sending/receiving data, tasks management, etc.
This software layer has to meet several constraints (bounded computation time, low memory footprint, etc.) that have to be enforced on all space-related and high-integrity software;

1. ASSERT was a FP6 project, involving 33 partners across Europe between 2004 and 2007. See http://www.assert-project.net/ for more details

– automate code production to avoid error-prone manual code that relies on human interpretation of non-formal (textual) specifications. With most commercial tools, it was impossible to generate the code of a complete system if all components were not developed within the same environment; so that engineers still needed to perform manual integration of system components, which potentially introduced errors. In fact, automatic code generation was hardly used in practice due to this limitation. In addition, most code generators were not meant for embedded platforms, and included constructs that are generally not accepted by space coding standards (e.g. dynamic memory allocation).

Following the ASSERT project, which documented most of the issues that needed to be addressed, the European Space Agency decided to create a series of follow-up studies with the sole objective of realizing all of the "dreams" of the ASSERT project. This is available today in the form of a set of tools called TASTE, and which is presented in the following sections (see [CON 10]).

The objective is not to detail all of the outcomes of the ASSERT project. A theme such as the constantly evolving reference architectures would require details that would go much beyond the scope of this chapter; instead we focus more on software-specific aspects (the four other points listed above).

9.4. Modeling a software system using TASTE: philosophy

When we started this project, we first asked ourselves if there was anything that was so typical to our systems that could explain or justify why barely any tool was used to support the development phase: no modeling tools, no code generators, no formal requirement capture.

We have standards that are very demanding in terms of software documentation and quality but operational projects will hardly comply to requirements regarding the need for early verification of the system based on models. Then by looking closer at the existing tools, and there are many on the market, we realized that nothing really addressed the problem sufficiently. Most of the tools in fact are not applicable for building complete systems, because they are not meant for engineers – actually, their target users are rather unclear; they will produce nice drawings but at the same time will give the user a severe headache.

To be fair, that is not entirely true: if we take a few tools independently from each other, we can create useful things: control laws, powerful state machines – but not a single tool that addresses our systems as a whole. To explain this last statement, look at these observations about our systems: first the nature of embedded systems is profoundly and fundamentally heterogeneous. This heterogenity goes over at least three axes:

– the nature of required software capabilities;
– hardware buses and processors;
– industrial consortia making the systems.

We shall look at these three points.

First of all, inside a satellite, software realizes various functionalities that are very different from one another: control laws to move the spacecraft in orbit and perform altitude maneuvers, system mode management, mission planning, thermal control, communication protocols, etc. In order to achieve these functions, project teams gather different specialized skills from scientists and software engineers.

Then regarding hardware it is similarly heterogeneous: the microprocessors we send into space are most often derived from the SPARC architecture, in which the internal data representation is different from those of the standard PC used to control spacecrafts from the ground. For this reason, it is for example not an option to send raw data to the ground without conversion to make sure it will be interpreted correctly. On-board buses can themselves have addressing schemes that require pre-processing involving swapping bytes. This kind of manipulation, related to endianness, is of course well known and frequent, but remains hard to implement because programming languages do not provide easy mechanisms for bit and byte manipulation.

Third point, in the space sector it is quite common that the development of one OBSW involves teams from several companies, from several countries, each having their own way of working, their own development environment, and so on.

Therefore, it is clear that our systems deal with heterogenity, and it is obviously also the case for most embedded systems.

The philosophy driving the TASTE approach focuses on finding out how to deal with heterogenity in an *automated* way. All technical issues seem easy to solve when taken individually; but is tedious and error prone when everything has to be put together. This is even more true when systems become distributed, since defining proper interfaces can quickly become a serious challenge.

Another observation we made is that when we build a new system – whatever system it is – the most important issue is the aim of the system. We want to make sure that the biggest effort is put on what makes the system novel and different from what has already been done in the past. For instance, the orbit and trajectory of a satellite are central issues that require an important engineering effort. During this phase, do we want to be distracted by implementation details and to introduce so early in the design process bulky software artifacts? If the answer seems obvious, look at existing, real software system specifications and count how often you find references to semaphores, binary frame definitions, thread identifications at the very first outline of a project.

The temptation is strong because it is important to occupy software engineers early, when they have no direct knowledge of the system, hence no real added value regarding its definition. What we want to point out here is that whatever solution we come up with, it has to be related to the system requirements, and not to software issues.

A third observation ensues from this last point: the skills of software engineers in this context are often misused: we ask software experts, whose aspiration it is to solve technical challenges (such as optimal resource usage) to develop applicative code. This code, which is often quite simple, consists of performing algorithms that other people have conceived, as they are experts in that field. At best, this generates frustration, and at worst the best software developers prefer to move to pure software companies where they think they can be more efficient. The space sector is partially protected from this extreme situation, because of its particular appeal but of course this is not always the case. Then replacing experienced and valuable people can become a real challenge now that many schools and universities have given up with low-level languages and concepts to concentrate on web-based developers.

Modeling a software system using TASTE involves spending hours working only on the system's capabilities, and forgetting about software technical details. The philosophy is to create tools that take care of all the bothersome tasks and let creative designers quickly prototype their ideas on the platform of their choice.

9.5. Common solutions

In practice, very few solutions exist that address the problems that we have just described. In fact, for about 15 years, it is not an exaggeration to say that an important part of the software community simply ignored them, favoring the solution that seemed to be accepted by everybody: the UML language. Flouting all efforts to formalize both syntax and semantics that used to be considered essential not only to programming languages but also to most other existing modeling languages (SDL, Lustre), UML gave up with the idea that the development of systems needed to be supported by a process, and rather proposed a huge palette of sometimes abstruse graphical editors.

Actually, despite the claims, very few people found a real added value in their development when using the UML standard. As a consequence, many research groups were created and started enhancing the language by adding new concepts, and new profiles in an inconsistent manner. The unfortunate result is that UML is mostly used only to create drawings for documentation, and that most tool vendors have disappeared from the market or were swallowed by bigger companies that buried their tools.

In addition to this, companies feel that they have invested a lot in new technologies and that nothing has come out if it, which of course had the disavantage of undermining the credibility of people proposing alternate technologies, since in general the arguments used to support one or the other are close. In theory, modeling software is a major evolution of the discipline – combined with powerful tools, it can be a breakthrough in how we think and develop systems, we have no doubt about that. However, decision makers now hardly believe it, and we cannot completely blame them for that. The idea that a unique language can address all the facets of a software development is absurd and unrealistic, as it amounts to denying the heterogeneous nature of systems. In practice, who has seriously thought of replacing Matlab or SCADE with UML to design a control law?

9.6. What TASTE specifically proposes

The TASTE process and tools propose a different solution to address the problems of capturing system functions and implementing them using modeling techniques. As systems are heterogeneous by nature in terms of software (different languages) and hardware (CPU, buses, etc.), one issue is to integrate all software components and enable communication between system nodes.

What TASTE does is to automate this integration phase by replacing manual (risky) activity that is, by definition, error-prone. Using modeling languages early in the development process enables the properties of the system to be checked using specialized tools. By doing so, system developers focus on their own part of the system and do not have to take into account integration constraints of their functions.

Only the non-functional, critical properties are captured at system level; then sophisticated and evolutional machinery is used that produces a consistent software set, guaranteeing that the constraints imposed by the system designers are respected at run-time. As a result, TASTE generates complete real-time, possibly distributed applications, running either on top of a real-time operating system combined with a middleware, or simply on a native, non-real time environment, such as a Linux box. In addition to this, many functionalities enable an efficient and iterative development to be created, which facilitates testing and analysis of results at run-time.

To sum up, the TASTE process is divided into four steps:

1) a system modeling phase that abstracts software requirements and constraints;

2) a transformation phase that translates models into a real-time architecture with all the resources to be used (tasks, semaphores, etc.);

3) a validation phase that verifies (prior to implementation) the feasibility of the system according to the selected physical architecture;

4) a code generation phase that assembles functions and configures the middleware/operating system for their execution. It results in one or several (in case of distributed systems) binaries ready for execution.

The remainder of this chapter is organized as follows:

– section 9.7 describes the TASTE process;

– section 9.8 details the technology and its underlying tools;

– section 9.9 focuses on the transformation from high-level models into a real-time execution platform;

– the final section describes the initial feedback from external users who have worked with TASTE, and the future of the toolset.

9.7. Modeling process and tools

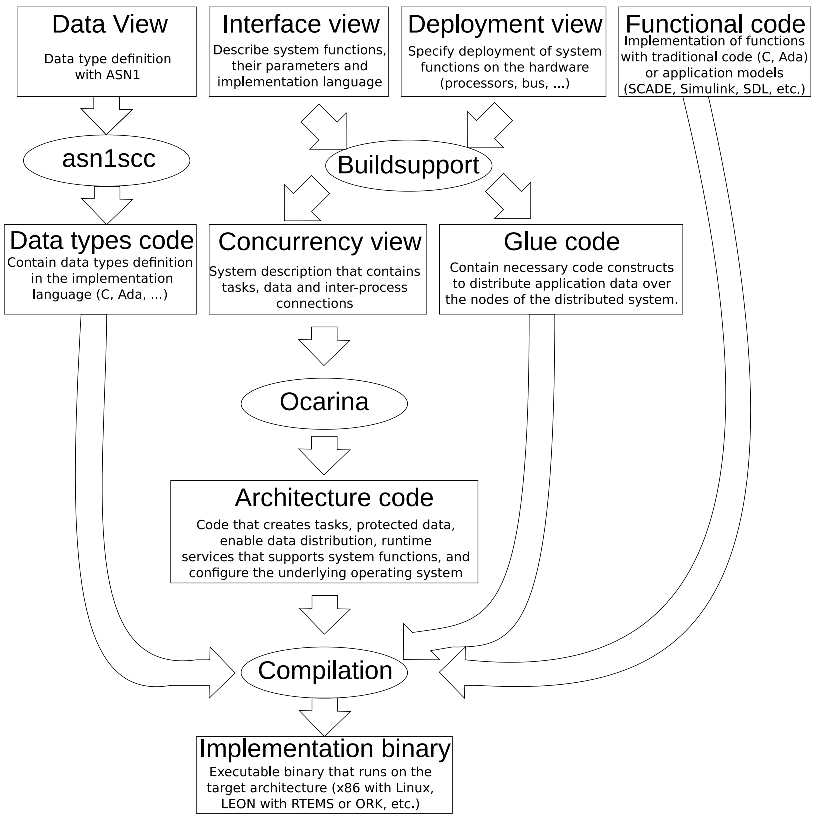

Data View
Data type definition with ASN1
Interface view
Describe system functions, their parameters and implementation language
Deployment view
Specify deployment of system functions on the hardware (processors, bus, ...)
Functional code
Implementation of functions with traditional code (C, Ada) or application models (SCADE, Simulink, SDL, etc.)
asn1scc
Buildsupport
Data types code
Contain data types definition in the implementation language (C, Ada, ...)
Concurrency view
System description that contains tasks, data and inter-process connections
Glue code
Contain necessary code constructs to distribute application data over the nodes of the distributed system.
Ocarina
Architecture code
Code that creates tasks, protected data, enable data distribution, runtime services that supports system functions, and configure the underlying operating system
Compilation
Implementation binary
Executable binary that runs on the target architecture (x86 with Linux, LEON with RTEMS or ORK, etc.)

Figure 9.1. *TASTE development workflow*

Design, analysis and implementation of application involves several languages (ASN.1 for data modelling, AADL for functional and architecture modelling concerns, etc.) and tools (`asn1scc` to translate data definition into implementation code, `buildsupport` for the generation of application skeletons, `ocarina` to automate deployment and configuration of applications on embedded and real-time platforms, etc.), as depicted in Figure 9.1.

Prerequisites consist of the availability of a preliminary software system logical architecture, which we assume results from a joint project between system engineers (who know what they want the system to do) and a software architect. This phase is today out of our scope. What we need is to know the main capabilities of the system, preferably already translated into a set of functional blocks. Our tools then allow this knowledge to be captured and make use of this important information.

The main idea is that we do not want to impose a particular language or tool to implement the functional blocks: users can choose those that they consider to be the most appropriate for each block, and let TASTE take care of the integration. In practice, TASTE currently supports: Matlab/Simulink, SDL (ObjectGEODE and Real-Time Developer Studio from Pragmadev), C, Ada and VHDL for hardware blocks.

To ensure data consistency between functions and also create appropriate encoder/decoders, data types have to be described using a standardized formalism. For that purpose, the ASN.1 language [DUB 01] was chosen: it is already widely used in the telecommunications domain and several tools already support it.

To describe the execution run-time with its requirements, the TASTE approach relies on the Architecture Analysis and Design Language (AADL) [SAE 09]. It abstracts software and hardware concerns with their constraints in a way that we can process them to validate several requirements and also automatically generate code.

In the following, we present how the process we defined to support this process that is the capture of high-level interfaces, the transformation to programming code, as well as the intermediate steps to model each facet (such as deployment, behavior).

9.7.1. *Interface capture*

The interface view editor is a graphical tool that aims to describe the logical interactions between the various functions of the system. In order to support large-scale architectures, functions can be grouped into hierarchical containers. Each function (represented by a box in Figure 9.2) is described by its provided and required interfaces. Provided interfaces (triangles) are themselves characterized by a set of non-functional properties and represent activation entry points of the function.

A provided interface can possess one of the following attributes:

– a cyclic interface has its own execution context. It does not have any parameters and is activated according to a fixed period;

– a sporadic interface has one input parameter, its own execution context/task and is activated at each arrival of incoming data. System designers can specify a minimal inter-arrival time between two occurrences of an incoming data;

– a protected interface is executed in the same context as its caller. However, when a protected interface is invoked, the execution run-time ensures that no other execution entity is accessing the function data. This prevents race conditions that could lead to errors and failures;

– an unprotected interface is, similar to protected ones, executed in the caller context but does not enforce mutual exclusion between concurrent entities.

Restrict interfaces types ease the validation process: by doing so, the computation model is strict and clearly defined. Thus, it eases the mapping of the model into a real-time architecture with concurrent entities (tasks, shared resources, etc.). Finally, it can be easily imported into scheduling validation tools so that our models can be processed by appropriate tools to check that real-time constraints are met before implementation efforts.

In addition, all interfaces should also specify real-time constraints, such as their deadline (the time until the function *has* to be performed) and their WCET (worst-case execution time).

Finally, required interfaces are connected to the provided interface so that the model depicts exactly interface dependencies between each function.

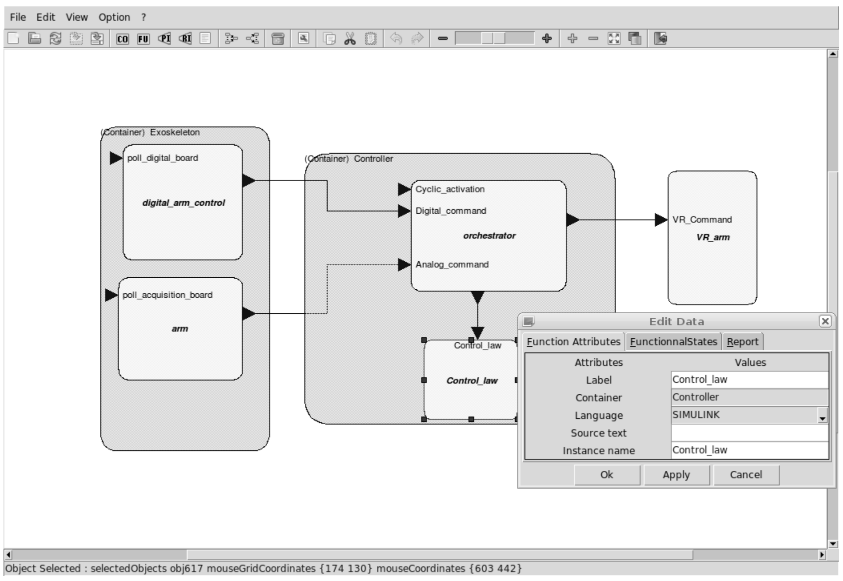
9.7.2. *Deployment and hardware configuration*

The deployment view editor (see Figure 9.3) is another graphical tool that is used to describe the hardware architecture of the system and allocate the functions identified in the interface view onto partitions located on a processor.

Inter-processor communications can be specified through buses and bus drivers. Each of these modeling entities can be characterized by a set of properties that are necessary for further code generation. Similar to the interface view, this modeling work is stored in an equivalent AADL textual representation.

9.7.3. *Behavior modeling*

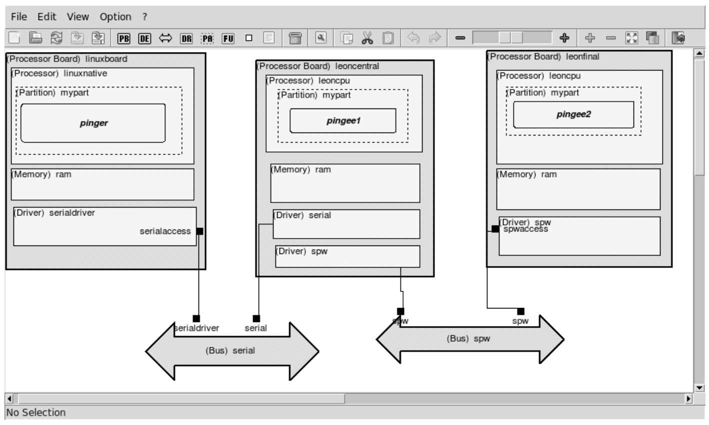
What we call "behavior" is the core of a function, the only thing end users should really care about. It corresponds to the description of what the function will do when



Attributes	Values
Label	Control_law
Container	Controller
Language	SIMULINK
Source text	
Instance name	Control_law

Object Selected : selectedObjects obj617 mouseGridCoordinates {174 130} mouseCoordinates {603 442}

Figure 9.2. *TASTE interface view editor*



No Selection

Figure 9.3. *TASTE deployment view editor*

it is executed according to its type (cyclic/sporadic/protected or unprotected) and constrains (period, etc.).

Using TASTE, it is possible to express the behavior of functions using virtually any language. The current supported subset is listed below:

– programming languages: C, Ada;

– modeling languages: the Specification and Description Language (SDL), using either ObjectGEODE or Pragmadev RTDS; the SCADE tool from Esterel Technologies; Matlab-Simulink;

– hardware description languages: VHDL, System-C.

What we have developed is a technology that is capable of adapting to any tool generating code with minimal effort, provided that the code generators are compliant with some constraints imposed by embedded coding standards. For example, our runtimes do not tolerate dynamic memory allocation outside of initialization functions, and most system calls have to be avoided.

VHDL and System-C are supported, which means that we can also communicate transparently with FPGA functions.

9.7.4. *Vertical transformation*

The edited interface and deployment views edition can then be submitted to a “vertical transformation” tool (see Figure 9.4). The aim of this fully automated activity is to produce a complete combined software and hardware architecture encompassing all the real-time and distribution properties of the system (in particular a set of processes, threads, shared resources, etc.). The output of the transformation is another AADL specification that is called the concurrency view.

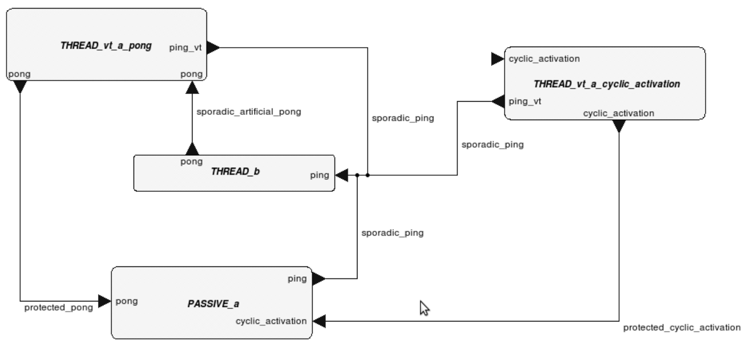


Figure 9.4. *Concurrency view*

9.7.5. *Concurrency view editor*

Although the concurrency view can be seen only as an intermediate internal step within the tool chain, it brings a unique opportunity to perform performance analysis on a model of the system.

As the concurrency view is described as a complete and legal AADL architecture, all the existing AADL analysis tools can be used at that stage. As detailed in section 9.7.1, the computational model of TASTE models has been simplified so that it can be easily exported to scheduling validation tools. Currently, two tools are integrated in the TASTE tool chain to perform this validation: Cheddar and Marzhin.

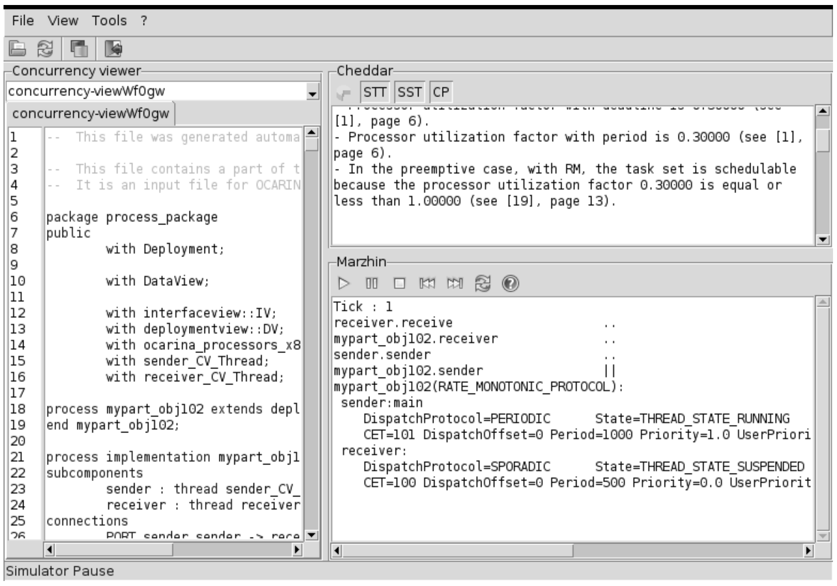
Cheddar is a schedulability analysis tool that aims to perform scheduling feasibility tests. Based on state-of-the-art scheduling theory algorithms (such as RMS, EDF), it is able to process AADL models and assess whatever computing resources (tasks, shared data) can meet their deadlines. If timing requirements cannot be met, it reports which entity (task, protected data, etc.) is generating a timing error in order to assist the designer in the refinement of the architecture.

Marzhin is a tool that simulates timing behavior from its architectural description. It processes AADL models and shows the state of each execution entity (running, sleeping, waiting for activation, etc.) while system is running. Marzhin details the state of each task according to its execution constraints (period, deadline, execution time, etc). For shared data, it shows their use by tasks and the blocking period time (when they are locked). Using this information, system designers are able to trace system execution and detect potential issues (such as a deadlock when using protected data without an appropriate locking mechanism).

The following figure illustrates the use of TASTE-CV with a basic producer/consumer example (a periodic task that produces data and sends it to a sporadic task). In Figure 9.5, the left column of the tool contains the AADL definition of the system (due to lack of space, we cannot include the entire model). The upper part of the right side shows the result of the schedulability analysis performed by Cheddar using various scheduling analysis techniques. In the following, the task set is schedulable using the preemptive rate monotonic algorithm. The lower part of the right pane illustrates the simulation of the system using Marzhin, showing the state of each task (sender and receiver). The figure depicts the beginning of the system execution, when the sender task is active (production of a new data) and the receiver is suspended, waiting for fresh data to arrive on its ports.

9.7.5.1. *Other schedulability analysis: MAST*

Our toolset also has the ability to export AADL descriptions into MAST models to analyze system schedulability using MAST. As for Cheddar, MAST provides several functionalities to analyze system schedulability using well-known scheduling techniques (EDF, Holistic, Offset-based, etc.). One particular interesting feature of MAST consists of its ability to consider distribution aspects and take into account the time required to send or receive data from one node to another (sending through a network device, data dependencies across distributed systems, etc.).



```
--  This file was generated automa

--  This file contains a part of t
--  It is an input file for OCARIN

package process_package
public
        with Deployment;

        with DataView;

        with interfaceview::IV;
        with deploymentview::DV;
        with ocarina_processors_x8
        with sender_CV_Thread;
        with receiver_CV_Thread;

process mypart_obj102 extends depl
end mypart_obj102;

process implementation mypart_obj1
subcomponents
        sender : thread sender_CV_
        receiver : thread receiver
connections
```

```
[1], page 6).
- Processor utilization factor with period is 0.30000 (see [1],
page 6).
- In the preemptive case, with RM, the task set is schedulable
because the processor utilization factor 0.30000 is equal or
less than 1.00000 (see [19], page 13).
```

```
Tick : 1
receiver.receive                    ..
mypart_obj102.receiver              ..
sender.sender                       ..
mypart_obj102.sender                ||
mypart_obj102(RATE_MONOTONIC_PROTOCOL):
 sender:main
    DispatchProtocol=PERIODIC      State=THREAD_STATE_RUNNING
    CET=101 DispatchOffset=0 Period=1000 Priority=1.0 UserPriori
 receiver:
    DispatchProtocol=SPORADIC      State=THREAD_STATE_SUSPENDED
    CET=100 DispatchOffset=0 Period=500 Priority=0.0 UserPriorit
```

Figure 9.5. *TASTE concurrency view editor*

MAST also provides the capability to automatically compute tasks timing properties (such as the period). For a given task, the tool deduces its priority according to its constraints (deadline, execution time, period, etc.). By using this feature, system designer are assisted in the definition of their architecture, ensuring deadline enforcement.

The use of different tools that provide various schedulability analysis and verification techniques strengthen our approach and makes system architecture design more reliable and robust: designers can verify their requirements using different tools and consider all aspects of their application, ensuring that requirements will be met before implementation efforts.

9.7.6. *Automatic code generation*

The last step of the TASTE modeling process consists of building the executable application from the functional blocks, the glue code generated by TASTE to handle transparent communication, and the AADL architecture defining the hardware and software interactions of the system. The Ocarina tool controls this task and generates the complete compilable set of source files while taking into account the run time execution characteristics of the Ravenscar computation model that has been selected for TASTE. Then compilation and link are performed automatically by the tool-chain orchestrator. Several possible operating systems can be used: bare systems using the Ada run-time (i.e. any operating system having a an implementation of the GNAT compiler), but also the RTEMS real-time operating system (not depending on the Ada run-time), which is a standard operating system in space and military applications. Note, however, that using a C run-time such as RTEMS prevents the benefits of Ada compiler checks (that can ensure the user does not use forbidden constructs in his code). We will now give some specific examples of what you can find in TASTE models and what important features it proposes.

9.8. Technology

With the work described in the previous section, TASTE uses a high-level architectural view of the system, which formally depicts the partitioning of the overall system in distinct subsystems and their interfaces. This information is expressed in the AADL.

The following is an excerpt from an actual design:

```
SYSTEM cyclic_function
    FEATURES
        cyclic_activation   : PROVIDES SUBPROGRAM ACCESS
            interfaceview :: FV:: cyclic_activation . others
          { Taste :: RCMoperationKind => cyclic;
            Taste :: RCMperiod => 500 ms;
            Taste :: Deadline => 500 ms;};
      compute_data : REQUIRES SUBPROGRAM ACCESS
            interfaceview :: FV::compute_data.others
          { Taste :: RCMoperationKind => unprotected;};
    PROPERTIES
      Source_Language => C;
  END cyclic_function;
  -- ...
  SUBPROGRAM compute_data
    FEATURES
      my_in : IN PARAMETER DataView::T_POS
          { Taste :: encoding => UPER; };
       result  : OUT PARAMETER DataView::T_POS
          { Taste :: encoding => NATIVE; };
  END compute_data_obj108;
```

As seen in the example, the interface descriptions include information about the:

– execution profile of the interface – e.g. timing information such as period or WCET, call type (cyclic, sporadic, etc.);

– implementation language/tool of the interface (e.g. “Simulink”);

– naming and direction of the parameters (e.g. “result”, “in”) of the interface;

– type of the interface parameters through ASN.1 grammar specifications (in the example above, “RequestGNC” is a type of the ASN.1 module “DataView”). ASN.1 encoding specifications (e.g. “UPER” stands for Unaligned Packed Encoding Instructions, one of the many available ASN.1 encodings).

The types of the interface parameters are described in ASN.1 specifications. ASN.1 is an ISO/IEC and ITU-T standard which enables specification of data structures, both from the semantic as well as the encoding point of view. It is widely used in telecommunication protocols, and has been selected for use in TASTE.

```
T−POS ::= SEQUENCE {
        longitude  REAL (−180.0 .. 180.0),
         latitude   REAL (0.0 ..  90.0),
        height  REAL (0.0 ..  100.0),
        subTypeArray Subtypearray
}
```

It includes all the semantic information about the data carried across the interface’s invocation, as well as the limitations (ASN.1 constraints) on the values that are allowed to pass through. For example, the first field (“longitude”) is a real that must be limited in the [-180.0 .. 180.0] range.

The formal descriptions of interfaces (in AADL and ASN.1) allow TASTE to automatically handle a number of issues by using the provided information.

9.9. Model transformations

The TASTE process is made of several model transfomations step. We now detail them.

9.9.1. *Model to model*

This transformation is supported for a variety of modeling tools (Simulink/RTW, ObjectGEODE, Pragmadev RTDS, etc.) and implementation languages (Ada, C, SystemC/VHDL). As it is based on the AADL/ASN.1 model, it is always guaranteed to generate the same semantic content for the interface parameters, regardless of the implementation tool/language – i.e. the “translated” definitions of the ASN.1 types are semantically equivalent in all the supported target tools/languages. Figure 9.6 shows the generation of Matlab/Simulink function from its definition within the Interface View: the process creates input (`my_in`) and output (`result`) parameters to interact with the remaining functions of the system.

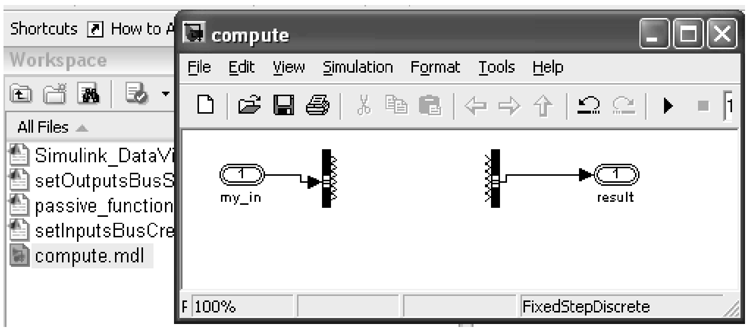


Figure 9.6. *Skeleton file generated for Simulink*

9.9.2. *Model to code*

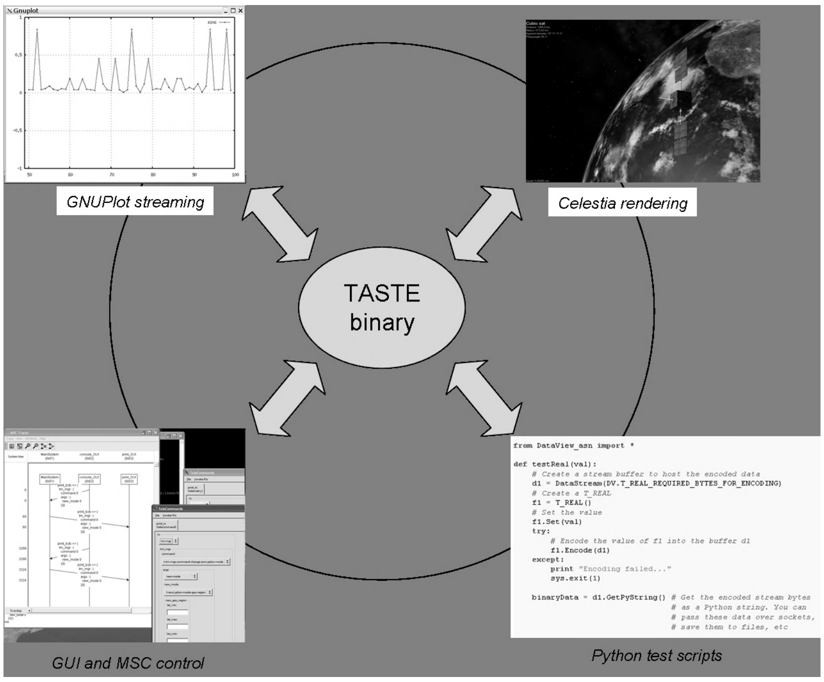
When functional modeling is completed, the code generators of the modeling tools are invoked, and C code is generated. Modeling tools generate code in different ways, however – and even though (thanks to the previous step) the data structures of the generated code across different modeling tools carry semantically equivalent information, the actual code generated cannot interoperate as is:

```
/* Declaration from ObjectGEODE */

typdef struct {
        GU_RG_51_10 fd_height;
        GU_RG_50_9 fd_latitude;
        GU_RG_49_8 fd_longitude;
        GU_SEQOF_52_11 fd_subtypearray;
} GU_T_POS;

/* Declaration from Simulink */

typedef struct {
        real_T  longitude ;
        real_T  latitude ;
        real_T  height ;
        Subtypearray_type subTypeArray;
} T_POS;
```

Therefore, integrating the code generated by different modeling tools requires “data bridges” to be built that translate (at run-time) the data structures from one modeling tool to those of the other and vice versa. Manually creating these data bridges would be a very error-prone process, and would have to be repeated if the messages were changed. In TASTE, they are automatically built by our custom-made code generators.

9.9.3. *Automated GUIs and regression checking using Python scripts*

In the overall AADL system design, the designer can specify the subsystems for which a graphical user interface should be created (see Figure 9.7). The TASTE tool chain reads the interface information of these subsystems and automatically generates code for interactive graphical user interfaces that operate on these interfaces. These GUIs provide real-time access to running systems, allowing information exchange, e.g. invoking telecommands or receiving real-time telemetry. The same information is also used in order to build Python run-time bridges that allow real-time interaction with a running system. Complex regression checking suites can be written easily with the combined clarity and brevity (and developing speed) of a ubiquitous scripting language.



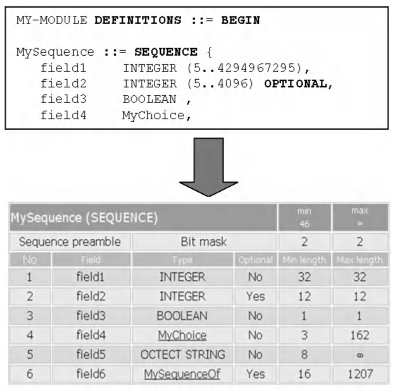
```
from DataView_asn import *

def testReal(val):
    # Create a stream buffer to host the encoded data
    d1 = DataStream(DV.T_REAL_REQUIRED_BYTES_FOR_ENCODING)
    # Create a T_REAL
    f1 = T_REAL()
    # Set the value
    f1.Set(val)
    try:
        # Encode the value of f1 into the buffer d1
        f1.Encode(d1)
    except:
        print "Encoding failed..."
        sys.exit(1)

    binaryData = d1.GetPyString() # Get the encoded stream bytes
                                  # as a Python string. You can
                                  # pass these data over sockets,
                                  # save them to files, etc
```

Figure 9.7. *GUI and tests functions provided by TASTE*

Telemetry can then be piped to plotting and monitoring applications, for easy real-time monitoring and control of running systems.

9.9.4. *Generation of interface control documents (ICDs)*

In order to support legacy development, one of the TASTE tools (the ICD generator) automatically creates an interface control document that describes all interface parameters as they are encoded at the bit-level from ASN.1 encoding (see Figure 9.8). This allows interoperability with other development teams that choose – for whatever reason – to not use the TASTE tools. Following the same philosophy as the rest of the TASTE tools, the ICD generator allows the designers to get free and immediate updates of their ICD, without the cost (and potential errors) involved in a manually maintained ICD.



```
MY-MODULE DEFINITIONS ::= BEGIN

MySequence ::= SEQUENCE {
   field1     INTEGER (5..4294967295),
   field2     INTEGER (5..4096) OPTIONAL,
   field3     BOOLEAN ,
   field4     MyChoice,
```

MySequence (SEQUENCE)				min 46	max ∞
Sequence preamble		Bit mask		2	2
No	Field	Type	Optional	Min length	Max length
1	field1	INTEGER	No	32	32
2	field2	INTEGER	Yes	12	12
3	field3	BOOLEAN	No	1	1
4	field4	MyChoice	No	3	162
5	field5	OCTECT STRING	No	8	∞
6	field6	MySequenceOf	Yes	16	1207

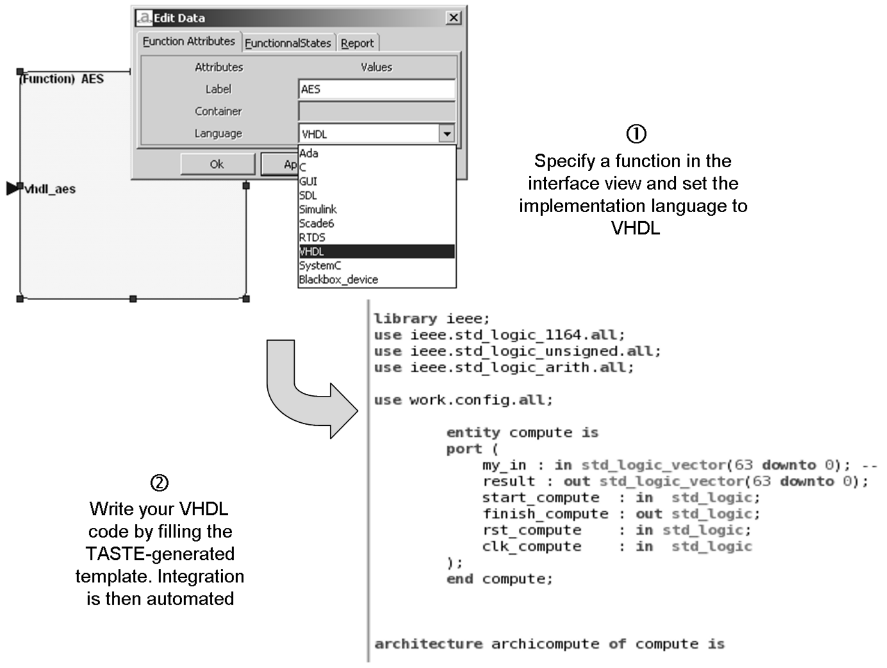
Figure 9.8. *Generation of ICD*

9.9.5. *ASN1SCC and ACN (ASN.1 Encoding Control Notation)*

As the primary target of the TASTE process and tools is the space domain, we created a custom ASN.1 compiler (ASN1SCC) that generates code specifically designed to be executed in limited-resource environments. It involves no dynamic memory, it uses no system calls, and is portable to all the target architectures, including Leon (i.e. the generated code includes no outside references to "black-box" libraries). To support legacy encodings and be able to communicate with existing protocols and implementations, the ASN.1 compiler was enhanced with the ASN.1 encoding control notation (ACN) which enables direct control of the encoding – i.e. the binary format of the generated streams.

9.9.6. *Support for hardware development*

The TASTE method and tools have been recently upgraded to also support development (and automatic integration) of hardware components. If a subsystem is marked with “VHDL” or “SystemC” in the high-level AADL specification of the interfaces, it automatically acquires VHDL and SystemC skeletons (in the “model-to-model” phase described before) as well as the appropriate device drivers (in the “model-to-code” phase) that communicate with the chip at run-time (see Figure 9.9).



Specify a function in the interface view and set the implementation language to VHDL

② Write your VHDL code by filling the TASTE-generated template. Integration is then automated

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

use work.config.all;

        entity compute is
        port (
            my_in : in std_logic_vector(63 downto 0); --
            result : out std_logic_vector(63 downto 0);
            start_compute  : in  std_logic;
            finish_compute : out std_logic;
            rst_compute    : in std_logic;
            clk_compute    : in  std_logic
        );
        end compute;



architecture archicompute of compute is
```

Figure 9.9. *VHDL integration within TASTE*

9.10. The TASTE run-time

Modeling time and effort is a valuable asset that is to be used and reserved for the construction of the final system. To do so, the TASTE tool-chain integrates a set of code generation tools to map all models down to the source code targeting a dedicated run-time environment.

Let us describe it from a top-down perspective. From the full set of models (ASN.1 and AADL), we have a complete description of the system: types manipulated, interfaces of processes and threads, connection topology, and flow of information and interaction. These models are used as direct inputs to build an appilcation-specific run-time using the Ocarina code generator, and the PolyORB-HI set of run-times.

9.10.1. *The Ocarina code generator*

We rely on Ocarina code generation facilities (see [HUG 08]) to generate optimized code for all entities which can be optimized through a careful examination of the architecture: communication buffers, structure of requests, request marshaling/unmarshaling, optimized task body, so as to avoid dead code. We have extended the Ocarina AADL-to-code tool-chain to also integrate device drivers as model artifacts. Such modeling allows seamless integration of both functional code (as application blocks), and device drivers.

In the context of TASTE, functional code is the output of the previous code generation steps: code generated for marshaling ASN.1 data types definition, or generated from other modeling framework supported by TASTE: SDL, Simulink, etc. Device drivers are integrated as functional models with a specific interface for (1) initializing the driver using dedicated API provided by the underlying RTOS, (2) sending or receiving data. Point (2) is modeled as any functional block using the same modeling artifacts as the functional code for concurrency (e.g. how to process data in parallel, etc.), and the call to the driver API to perform the actual send/receive.

Such an approach greatly eases the integration of protocols or drivers: they are seen at the same level as the functional block, and take advantage of the whole TASTE tool-chain to combine functional blocks, drivers and the generated code.

9.10.2. *The PolyORB-HI middleware and the operating system*

The generated code targets the high-integrity run-time infrastructure PolyORB-HI. This infrastructure acts as a portability layer for the integration of multiple languages (C or Ada), RTOS APIs (Ada Ravenscar, RT-POSIX, RTEMS), but also for the integration of device drivers (serial, Ethernet, SpaceWire). PolyORB-HI acts as an AADL run-time: it provides support for each model pattern defined at the upper-level. Two variants of PolyORB-HI have been implemented: an Ada variant, which relies on the RCM. It defines a set of patterns for deterministic concurrency. It makes provision for analyzability through the RMA and RTA frameworks. In addition, great care has been taken to ensure that the code meets more stringent requirements for high-integrity: the compiler to ease code review, and strengthen quality enforces restrictions that explicitly forbid dynamic memory, object-orientation or pointers. This variant runs either on native systems, RTEMS, or on the bare-board ORK+ [VAR 05] or GNAT Pro for high-integrity run-times.

A C variant, that uses the same concepts from the RCM, on top of the RTEMS operating system, or the RT-POSIX. Although C provides less support to check code quality, great care has been taken to ensure a level of quality similar to the Ada variant. The choice of one variant is mainly dictated by the availability of specific device drivers (e.g. CAN, MIL-1553, GPS receiver, etc.), or non-functional properties like memory overhead of the RTOS, run-time performance (such as WCET or jitter), and analyzability features. Current case studies did not fully evaluate schedulability of systems; this is currently on-going work. We have evaluated the impact of each variant in terms of memory consumption. Ada on top of RTEMS is obviously more demanding in terms of memory, than RTEMS/C and ORB+, which is a restricted kernel. Let us note that ORK+ also provides better safety capabilities thanks to the use of Ada, yet it lacks the driver support of RTEMS/C.

In general, we consider that TASTE does not generate any particular overhead, compared with an equivalent code that would be hand written. As an example, a simple binary containing a couple of threads will not occupy more than 15-20 kbytes, including everything, and running on Linux.

Both Ada and C variants provide the same level of support to the application: the same patterns can be applied. In addition, we are currently integrating more drivers to ORK+ to ensure both variants are equal from the designer perspective.

9.11. Illustrating our process by designing heterogeneous systems

We shall illustrate the development process using a specific example, which is a simple system with several producer/consumer, deployed on heterogeneous hardware. It shows the overall deployment process, describes how we specify each view (*data view*, *interface view* and *deployment view*) and presents some metrics about memory consumption, emphasizing the low overhead introduced by the code generation.

9.11.1. *Case study overview*

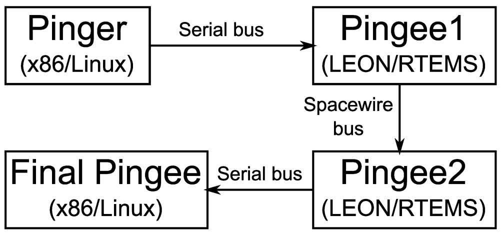


Figure 9.10. *Case study: overview*

The overall architecture of the case study is depicted in Figure 9.10. It is composed of four distributed nodes that communicate over several networks. The first node, “pinger” periodically produces data and sends them to the “pingee1” node through a serial bus. The “pingee1” node forwards the data to the “pingee2” node through the serial bus. Then, the data are finally transmitted to a last node, “finalpingee”, through a serial bus. This last node on the chain prints the data initially produced by “pinger”.

Nodes “pinger” and “finalpingee” are running a traditional x86 processor with a regular Linux operating system (such as Debian or Ubuntu) while “pingee1” and “pingee2” use a LEON processor (SPARC architecture) with the RTEMS operating system.

This case study is then the opportunity to demonstrate the following points:

– smooth integration of application languages into distributed architectures;

– automatic handling of architecture-dependent concerns, such as data encoding across the distributed nodes;

– schedulability validation prior to implementation efforts;

– integration flexibility: each node communicates with another using different buses (serial or spacewire).

Each following subsection illustrates these points more precisely by describing the development process. We first define the data view which describes types to be used within the distributed system. Then, we present the interface view which contains functions executed by each node and its associated deployment view which specifies the execution platform with its constraints (processor architecture, buses, etc.) and describe the distribution strategy of each function over the nodes.

9.11.2. *Data view*

As we have reduced the data of the system to the minimum, we only exchange integers across system functions. For that reason, the data view of our system is quite basic and defines an basic integer using ASN.1. The definition is shown below:

```
DataView DEFINITIONS AUTOMATIC TAGS ::= BEGIN
My-Integer ::= INTEGER (0 .. 65535)
END
```

From this ASN.1 specification, appropriate tools export data types definitions into AADL models so that they can be integrated by the other tools. This export functionality provides an exchange format for the data types and eases the integration of ASN.1 types in other models.

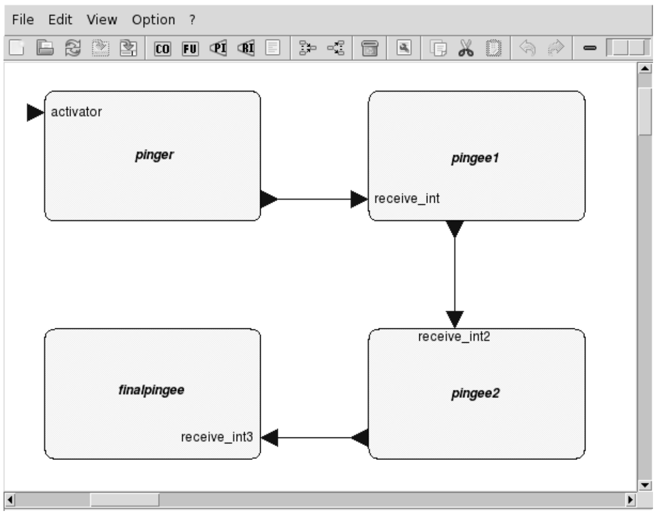


Figure 9.11. *Case study: interface view*

9.11.3. *Interface view*

Once the data types have been defined, we specify system functions in an interface view, illustrated in Figure 9.11. Four functions are defined (one function per node):

1) “pinger”, which produces the initial integer and transmits it to “pingee1”. As this function is executed on a periodic basis, it has a cyclic interface that triggers system activation according to a fixed period (1 second);

2) “pingee1”, which provides a sporadic interface to receive data from the “pinger” function. Application code is then triggered when incoming data are received and forwards them to the “pingee2” function;

3) “pingee2” is similar to the “pingee1” function (and so, uses a sporadic interface) and transmits received data to the “finalpingee” function;

4) “finalpingee” function provides a sporadic interface that prints the data received from “pingee2”.

9.11.4. *Generating code skeletons and write application code*

Once the interface view has been defined, TASTE tools generate:

– application skeletons, which contain function interface prototypes to be completed by system developers;

– glue code, which automates data communications through system functions. For example, in our system, the glue code provides a function to send data across the nodes so that developers do not have to worry about the underlying bus on the remote target that receives the data. These functions automatically request the appropriate encoding functions so that all architecture-dependent concerns (such as endianess) are dealt with.

The following listing describes the application skeleton generated for the "pinger" function.

```
extern void pinger_RI_receive_int(const asn1SccMy_Integer *);

void pinger_startup()
{
}

void pinger_PI_activator()
{
        /* Write your code here! */
}
```

The `pinger_startup()` function can be filled by the developer to perform initialization procedures (data to be created, etc.). Then, the `pinger_PI_activator()` routine is called periodically, when the interface is triggered. As the activator interface is periodic, it is triggered cyclically, each second. Finally, the generated code also contains the `pinger_RI_receive_int()`. This function corresponds to the glue code, which automatically encodes its argument and sends it on the appropriate required interface: in that case, it transmits data to the "pingee1" function.

Developers then completes the code, the following listing shows a code example for the "pinger" function:

```
void pinger_startup()
{
        printf ("PINGER STARTS\n");
}

int foo = 0;

void pinger_PI_activator()
{
        printf ("PINGER SENDS %d\n", foo);
        asn1SccMy_Integer tmp = foo++;
        pinger_RI_receive_int(&foo);

}
```

According to its code, the function prints a message when the function is initialized. Then, at each second (when the cyclic interface is triggered), it increments a global variable and sends its contents to the "pingee1" function using the routine `pinger_RI_receive_int()` provided by the glue code.

9.11.5. *Deployment view*

Then, we define the execution environment (processors, boards, buses, and devices) and bind each function to a platform by defining the deployment view, illustrated in Figure 9.12.

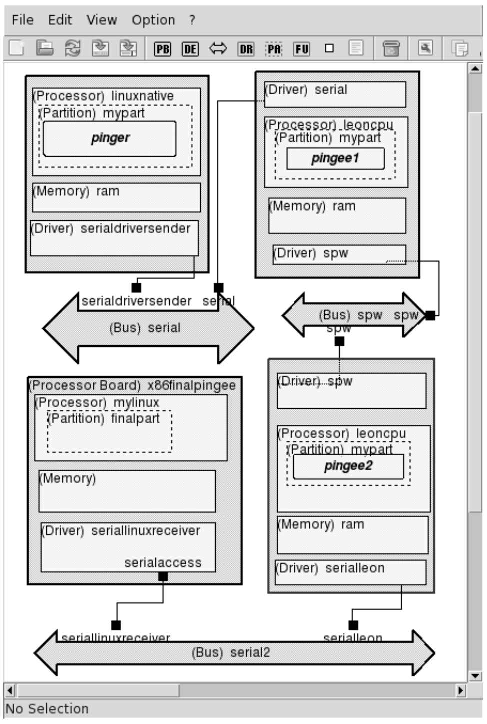


Figure 9.12. *Case study: deployment view*

As the system is executed on four separated nodes that communicate with each other, we define four boards, one for each function. Then, on each platform, we add devices that aims at enabling communication links across the nodes:

– nodes for the "pinger" and "finalpingee" functions host a serial line device to communicate with the LEON over the serial protocol;

– nodes for the "pingee1" and "pingee2" functions contain two devices: a serial line device to exchange data with "pinger" and "finalpingee" functions and a SpaceWire device to communication between the two LEON boards.

Finally, devices share communication buses, modeling the specific cables that are connected through the nodes. Aggregation of these components aims to depict the execution environment, providing all necessary elements to generate the complete run-time system.

9.11.6. *Concurrency view and schedulability analysis*

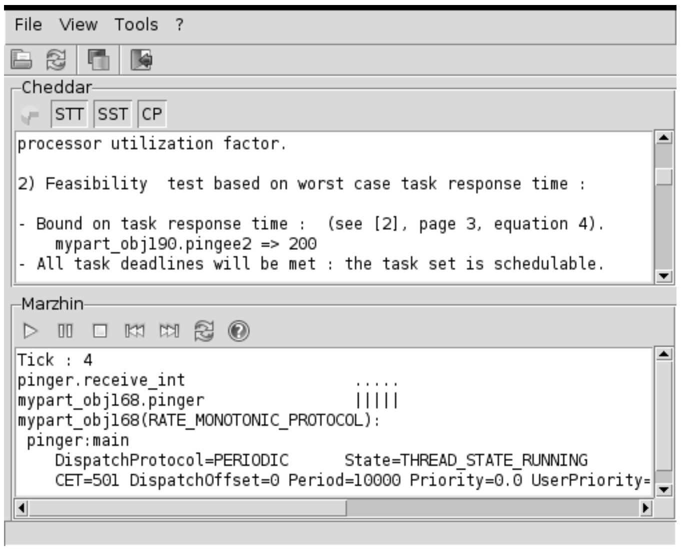
Once data, interface and deployment views have been defined, our tool translate these models into a concurrency view that instantiate system functions and adds all the required resources to implement the system. The result is an AADL model that contains execution components, such as processes, tasks, or shared resources (variables, mutexes, semaphores, etc.) with all their properties (period, deadline, priority, etc.).

As this description contains all the non-functional requirements regarding the execution, dedicated tools can process and analyze it. In the context of high-integrity system, one major concern is the real-time requirements and the system designer may want to check their enforcement as early as possible in the process. Our tool chain interfaces the so-called concurrency view with two scheduling analysis tools: *TASTE-CV* and *MAST*. The following paragraphs illustrate their use in the case study.

TASTE-CV analyzes the generated concurrency view for the simulation or the validation of scheduling aspect of the system. For scheduling validation, it relies on the Cheddar scheduling validation tool: TASTE-CV transforms the concurrency view into a suitable representation for Cheddar. Then, the Cheddar [SIN 08] tool analyzes the system and its executable entities, ensuring that real-time constraints can be met (deadline, execution time, etc.). Depending on system entities and requirements, Cheddar outputs the result, detailing whether scheduling constraints are met or not. The upper part of Figure 9.13 shows the scheduling analysis performed in the case study.

TASTE-CV also embeds Marzhin, a system simulator. This analyzes the system and simulates it, showing the execution of system tasks, shared resources, and mutexes. To do so, it takes into account specific system requirements, such as the scheduling policy, tasks and shared resources properties (priority, period, deadline, locking policy, etc.). The bottom part of Figure 9.13 shows the simulation in the case study.

MAST [HAR 01] also provides scheduling validation functionalities to ensure that timing requirements of the system are met. Even if the approach is similar to TASTE-CV and its associated Cheddar tool, it also provides other functionalities, such as the analysis of distributed aspects. Indeed, MAST is able to analyze timing aspects with respect to communication concerns (such as buses latency). As a result, depending on system deployment and environment, scheduling validation is more accurate with either MAST or Cheddar.



```
processor utilization factor.

2) Feasibility  test based on worst case task response time :

- Bound on task response time :  (see [2], page 3, equation 4).
    mypart_obj190.pingee2 => 200
- All task deadlines will be met : the task set is schedulable.
```

```
Tick : 4
pinger.receive_int                        .....
mypart_obj168.pinger                      |||||
mypart_obj168(RATE_MONOTONIC_PROTOCOL):
 pinger:main
    DispatchProtocol=PERIODIC      State=THREAD_STATE_RUNNING
    CET=501 DispatchOffset=0 Period=10000 Priority=0.0 UserPriority=
```

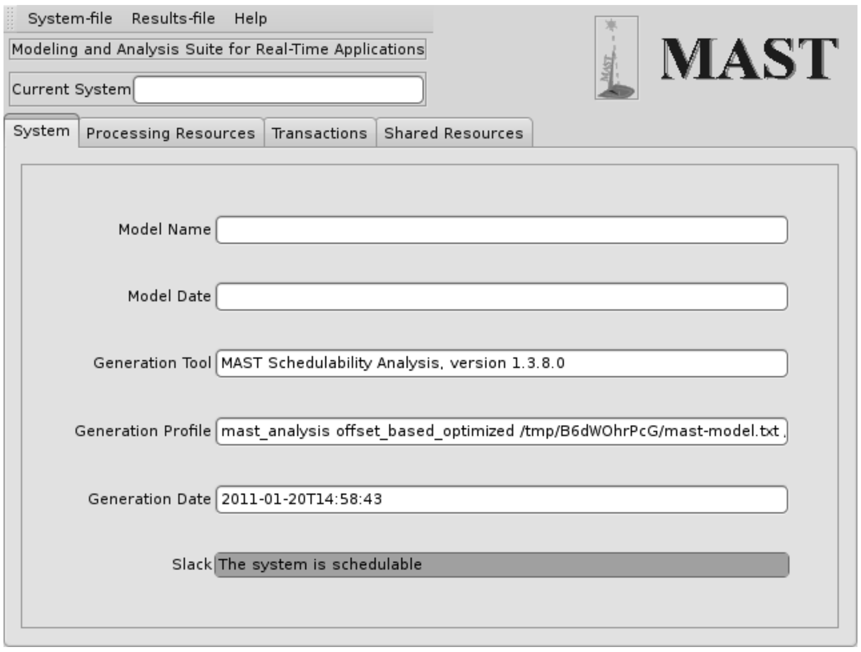
Figure 9.13. *Case study: concurrency view edition with scheduling validation functions*

To interface the generated concurrency view with MAST, our tool chain converts it into a new representation for MAST (a XML file). This translator transforms execution entities (tasks, shared resources, processors, buses) from AADL to a suitable representation. Then, MAST analyzes the system according to a specific scheduling protocol and indicates whether scheduling constraints can be met or not. Figure 9.14 shows the result of the scheduling validation in the case study.

These validation functionalities are quite useful for system designers: they ensure enforcement of real-time constraints early in the development process. However, it is possible to do that for two main reasons:

1) the language used to describe system architecture and its associated execution entities (AADL) does not introduce ambiguous notation and clearly defines each entity property or constraint. Then, these models can be processed later and can be translated into another representation, such as that used by Cheddar or MAST;

2) the concurrent execution model introduces important assumptions and is clearly defined from a semantic point of view so that it can be easily processed by scheduling validation tools. Indeed, our approach relies on four different types of execution model to execute system code (*cyclic*, *sporadic*, *protected*, and *non-protected*). In this way, task constraints are clearly defined and their execution does not introduce construction that can break scheduling analysis.



Modeling and Analysis Suite for Real-Time Applications

Current System

Model Name

Model Date

Generation Tool MAST Schedulability Analysis, version 1.3.8.0

Generation Profile mast_analysis offset_based_optimized /tmp/B6dWOhrPcG/mast-model.txt

Generation Date 2011-01-20T14:58:43

Slack The system is schedulable

Figure 9.14. *Case study: scheduling feasibility analysis with MAST*

9.11.7. *Memory overhead metrics*

Generated applications were analyzed to assess their compliance regarding high-integrity requirements. The previous section presented the validation of real-time constraints, the following paragraphs shows how our approach deals with embedded requirements.

In embedded systems, one major concern is the memory footprint. Indeed, high-integrity systems use specific memory footprints (protected against various hazards that occur in hostile environment) that are very expensive. In addition, application binaries must contain only required functions and avoid so-called *dead-code* (code that should not be executed but could be invoked in case of security or safety issues, such as buffer overflow).

Memory analysis of generated applications was performed, results are illustrated in the table below. In Table 9.1, we report memory metrics for each layer of the system:

– application size (code written by the user);

– glue size (code that encodes/decodes data and interfaces the middleware with the application code);

	Pinger	**Pingee1**	**Pingee2**	**Final Pingee**
Memory (stripped)	23 kB	275 kB	274 kB	24 kB
Function size	3 kB (12%)	3 kB (2%)	3 kB (2%)	4 kB (18%)
Glue size	12 kB (46%)	12 kB (7%)	12 kB (7%)	8 kB (35%)
Middleware size	11 kB (42%)	33 kB (18%)	32 kB (18%)	11 kB (47%)
Executive/OS size	N/A	133 kB (73%)	133 kB (73%)	N/A kB

Table 9.1. *Metrics for each system layer*

– middleware size (functions that manage all executable entities of the system, handle device drivers and adapt the generated code to the underlying operating system);

– execution run-time/OS size.

Function/application size is quite small. This result is expected as we only write some C code that outputs data on system terminal. The size of the glue represents only a few kilobytes. It means that the implementation approach avoids many common issues of high-integrity system design by introducing a small memory overhead. Middleware size is also quite small: a few kilobytes to integrate the generated code and system functions with the underlying hardware, as other middleware (such as CORBA) traditionally introduces dedicated programs that consume several megabytes of memory just for framework implementation. Finally, the size of the execution platform can be measured only for the LEON target as it is contained in a single object file. For Linux nodes, it is difficult to evaluate the size of the run-time as it relies on a monolithic kernel and a specific C-library. One solution would be to consider the size of the kernel and its associated C-library, but the result would not be accurate as the kernel was built to run a complete system that collocate many other applications.

Our design approach introduced only a few kilobytes of memory overhead. Compared to other approach, it reduces the amount of code used to interface application-level concerns with the execution environment. In other words, the cost of automatic application integration can be considered as quite small compared to the benefits it brings to system developers: no integration problems of heterogeneous systems, automatic integration with device drivers, etc. Then, the decision to use such a method to design high-integrity systems would reside in the assessment of its accuracy: is it better to continue to use traditional approaches and deal with all these well-known and old problems? Or is it possible to afford a small memory overhead and avoid all these implementation traps and pitfalls from the beginning of the development? The choice is up to developers and designers; solutions are now available to them.

9.12. First user feedback and TASTE future

The complete development of TASTE required a significant amount of work and time to reach the level of a full working prototype with an appropriate level of maturity. Most of the work was initially concentrated on the development process that TASTE is supporting as we consider it more important than the technologies used. Then, we focused on the modeling languages and integration issues to deliver something that requires a small effort at the beginning but affords many benefits in the end, by automating most of the development phases and ensuring system consistency.

When the initial prototype turned out to be an efficient product, we decided to give the system and software designers a chance to experiment it. As for any new product or technology, potential users are initially confused by the richness and complexity of the proposed solution. Developers and users have to work together with an open mind to succeed. The very first steps on user side were carefully accompanied by strong support provided by the tool's developers. Questions were asked and answered, comments were processed and disagreements about the way TASTE was dealing with the process were expressed and discussed.

Strong cooperation between teams was essential to overcome the initial barriers and address the real issues. Although we claim that TASTE was compatible with different kinds of existing programming or modeling languages that were familiar to software designers, using the toolset requires the use of two additional languages: AADL and ASN.1, but TASTE designers were clever enough to simplify its use by providing a graphical user interface that hides the AADL description. This clearly speeds up the learning curve while keeping the advantage of using a system design language backstage for possible future property verification and connection to additional tools. Using such an approach maintains the benefits with a limited disavantage at the system design level. ASN.1 was somewhat newer to most of the users but at the same time quite close to very well known programming languages. The idea of having a data model that was fully integrated to system design and used consistently to produce full software by ensuring the right integration of software components, was really new to many users. In existing projects, data models are not formally defined and nothing exist to ensure automatic consistency from top to bottom, except the Interface Control Documents but they are just papers. Most of the users found ASN.1 very valuable up to the point where they could envisage the use of this language outside the TASTE environment.

Although the benefits were clearly identified (a user even claimed he successfully generated a complete software implementation that exhibits higher performances than the manually coded version), some limitations were found. A category of users claimed they did not need such technology as they usually do not have heterogeneous systems. In a sense they were right when they see software development as a pure programming activity and not as a combination of modeling and programming. Other users regretted the absence of key support functions, such as traceability management tools, configuration management facilities, or document generation features. At least such remarks prove that the core facilities offered by the tool were found efficient up to the point where people may envisage the full deployment of the tool.

9.12.1. *The future of TASTE*

Following the long and hard development phases of TASTE, and having analyzed the first user's feedback, we are now at a point where the future of this technology will be carefully defined. Part of this future is made of technical perspectives; the rest is dealing with the toolset itself as a potential commercial product. From the technical side, we see many open opportunities related to the use of standard languages or coming from user feedback. The use of AADL is clearly a strong advantage as it ensures that TASTE can be easily integrated into a system development process using the language to capture and verify system designs.

The flexible architecture also guarantees the future inclusion of additional languages in a way similar to that achieved for the currently supported languages. User suggestions provided us with many interesting ideas to improve the usability in a real industrial environment (connection to process support tools, extension of testing features, etc.). TASTE in its current state is close to a commercial product that would be usable in an industrial context. Each underlying technology (AADL, ASN.1, etc.) can be used independently having already a positive impact on a standard development process, but TASTE by itself is more than the sum of its components and brings additional benefits when used in its entirety: automatic design and code generation, consistency insurance with the data model, flexibility with respect to the various development platforms. This led us to open discussions with the development team and potential users to clearly identify commercial interest and build a commercialization strategy for TASTE, possibly outside the space domain. Regarding licensing schemes, at the moment most of the TASTE tools follow a GPL license for non-commercial use.

9.13. Conclusion

The flexibility brought to digital systems by software components is so high that it seems that there is no limit to the functions those systems can handle. However, increasing system complexity is now pushing software engineering to the limits of currently used technologies and this convinced the initiators of ASSERT to propose a new approach. The main drivers of this new process are first to capture a minimal set of inputs from the system designer, to automate most of the software implementation tasks and to restrain programmers with the use of rigorous rules. As a positive result, a good consistency is preserved during system design, multiple implementations can be generated from one unique model, and the time from design to code is drastically reduced.

This new approach is the result of initial efforts partially funded by the European Commission under the FP6 ASSERT Project and further completed by ESA funding. TASTE is now a fully operational toolset that captures the system architecture with AADL, defines the data model with ASN.1, and finally, combines heterogeneous components into an homogenous software application to be uploaded on different targets up to the flight model. Different extensions are today undergoing research or planned by the community with the financial and technical support of ESA (links to system modeling tasks, introduction of hardware components, and connection to development process support tools such as configuration management tools).

The choice of standard languages, such as AADL and ASN.1, together with the open architecture of the tool implementation leaves many doors open for extensions to better cover all the design steps from system requirement capture down to software deployment. Initial user feedback clearly indicates that TASTE does not have everything a system designer may wish to have but provides strong support to ensure consistency down to software deployment and reduces the risk of having integration difficulties, which generally impact on the development schedule. The community that created ASSERT is now contemplating the different options to disseminate and possibly commercialize TASTE, while maintaining a steady effort to extend its capacity: the main goal is to push forward the current technological barriers and release the system designer to develop new ambitious missions in the space sector within acceptable budget, and quality envelopes.

A last word: many thanks to the development team for providing input to this chapter, and for the excellent work they undertake on a daily basis for TASTE. In particular we are very grateful to Thanassis Tsiodras and George Mamais from Semantix, Eric Conquet from ESA, and Pierre Dissaux from Ellidiss.

9.14. Bibliography

[CON 10] CONQUET E., PERROTIN M., DISSAUX P., TSIODRAS T., HUGUES J., "The TASTE toolset: turning human designed heterogeneous systems into computer built homogeneous software", *Proceedings of Embedded Real Time Software and Systems 2010*, Toulouse, France, May 2010.

[DUB 01] DUBUISSON O., FOUQUART P., *ASN.1: Communication Between Heterogeneous Systems*, Morgan Kaufmann Publishers Inc., San Francisco, CA, 2001.

[ESA 03] ESA, "Part 1: Principles and requirements", *ECSS-E-40 Parts 1B : Space Engineering - Software*, November 2003.

[HAR 01] HARBOUR M. G., GARCÌA J. G., GUTIÉRREZ J. P., MOYANO J. D., "MAST: modeling and analysis suite for real time applications", *13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands,, p. 125–134, June 2001.

[HOF 97] HOFFMAN M., BEAUMONT T., *Application Development: Managing the Project's Life Cycle*, MC Press, LLC, 1997.

[HUG 08] HUGUES J., ZALILA B., PAUTET L., KORDON F., "From the prototype to the final embedded system using the ocarina AADL tool suite", *ACM Transactions in Embedded Computing Systems (TECS)*, vol. 7, p. 1–25, July 2008.

[SAE 09] SAE, *Architecture Analysis & Design Language v2 (AS5506A)*, SAE, January 2009, available at http://www.sae.org.

[SIN 08] SINGHOFF F., PLANTEC A., DISSAUX P., "Can we increase the usability of real time scheduling theory ? the cheddar project", *Reliable Software Technologies'08 - Ada Europe*, Venice, Italy, 2008.

[VAR 05] VARDANEGA T., ZAMORANO J., DE LA PUENTE J. A., "On the dynamic semantics and the timing behavior of Ravenscar kernels", *Real-Time Systems*, vol. 29, p. 59-89, 2005.

THIRD PART

Security in Distributed Systems

Chapter 10

Introduction to Security Issues in Distributed Systems

10.1. Problem

Distributed systems intrinsically exhibit communication between processes that can be situated at different locations, on different machines. Their functioning should satisfy two categories of properties:

– *safety* properties, which guarantee the correct operation of the system itself. In a standard operating regime, no undesired event or situation should occur [BER 01];

– *security* properties, which prevent an intruder from misbehaving.

These two categories of properties are generally necessary for the correct behavior of any distributed system. The techniques employed to guarantee these two kinds of properties differ.

10.1.1. *Safety*

As mentioned in the previous parts of this book, numerous applications present *critical* aspects, in particular *embedded systems*. A failure of such a critical system may have severe consequences, be it economic (loss of costly hardware material), or human (such as the malfunction of healthcare equipment or a nuclear power plant).

Chapter written by Laure PETRUCCI.

Distributed Systems: Design and Algorithms Edited by Serge Haddad, Fabrice Kordon, Laurent Pautet and Laure Petrucci
© 2011 ISTE Ltd. Published 2011 by ISTE Ltd.

A communication protocol constitutes a typical example of a system exhibiting safety characteristics: messages must transit from an emitting process to one or several reception processes. Even though the communication *medium* loses messages, the protocol should ensure that any message sent will eventually be received (safety property).

When designing such critical systems, an *a priori analysis* ensures that desired properties will effectively be satisfied. For that purpose, several steps are followed, as detailed in Volume 2, Part 1:

– a *formal model* of the system is designed. The target language depends on the kind of system modeled, its expected qualitative and/or quantitative properties;

– the *expected properties* are expressed in some logics, compatible with the chosen modeling language;

– *model-checking techniques* then help to formally, i.e. mathematically, check that the properties hold for the model [BER 01, CLA 00].

However, when designing large critical systems, these *model-checking* techniques may fail. Indeed they perform an exhaustive exploration of all states the system can reach. Then, the "state space explosion problem" often occurs. Some techniques, such as those presented in Volume 2, Part 2, cope with this problem by reducing the state space, but this may not be enough to guarantee the properties remain.

Middlewares (e.g. AADL [SAE 10], see also Volume 2, Chapter 5), as in the air-traffic control project of Chapter 9, provide a software environment for which part of the system properties are correct by construction, as the tool suite handles the development from the modeling phase until code generation.

10.1.2. *Security*

The security problem is concerned with the possible intrusion of a misbehaving party within the distributed system. An external actor may spy upon the communication between two processes and can even change the content of messages exchanged. The actions of this external actor can thus modify the normal behavior of the system, and may cause dramatic consequences, such as the modification of a bank transaction, a critical calculus result, etc. Hence, several problems arise:

– *confidentiality* of the exchanged information: an external party should not be able to understand the information he accesses. If he is aware of some information, he does not have the means to interpret it;

– *integrity*: a third-party must not be able to alter data;

– *authenticity* of a message guarantees that the receiver knows the actual sender;

– finally, a message *cannot be repudiated* by its receiver, which cannot then pretend it was not received.

The following sections introduce the security issues that will be further described in Chapters 11 and 12 of this book.

10.2. Secure data exchange

In order to represent data exchange between processes or users, the paradigm of two persons, *Alice* and *Bob*, is often used, where Alice wants to send a message to Bob. This message is a secret and should not be read by anybody else. Hence, an intruder, usually named *Charlie*, must not read the message. Moreover, Charlie should not be able to alter Alice's message, nor replace it with his own, or make Bob believe he is Alice.

EXAMPLE.– Let us consider this example with a surface mail message (included in a packet). The packet is such that padlocks can be used to secure the contents. The postman has some grudge against Bob and Alice, and thus is motivated by bad intentions. Initially, each person has his/her own padlock and the corresponding key. A possible communication among these different actors is depicted in Figure 10.1.

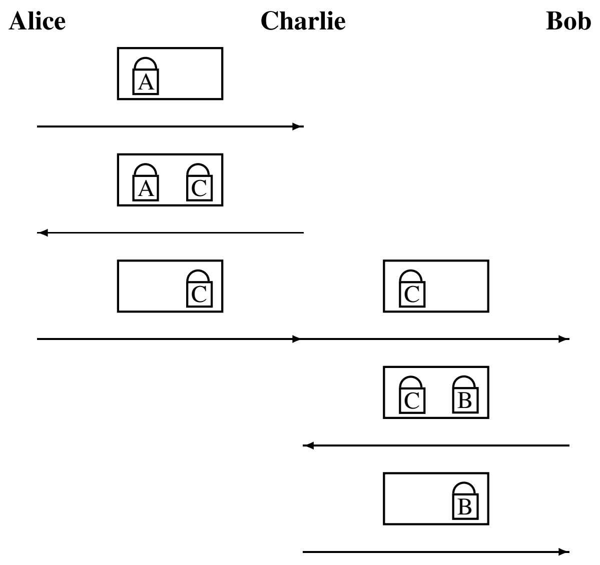


Figure 10.1. *Communication between Alice and Bob, spied on by Charlie*

In this scenario, Alice sends a packet with her padlock. The postman, Charlie, intercepts the packet, adds his padlock, and sends it back to Alice. She then believes the new padlock belongs to Bob, and thus removes her own, before sending the packet again. Charlie can then intercept it, remove his own padlock, read the message, and then transmit the packet to Bob, operating as Alice would have.

This example exhibits several problems:

– Charlie obtained knowledge of the secret message;

– when Charlie reads the message, he can also modify it;

– Alice and Bob believe they communicate with one another (the addresses of both the sender and the receiver are correct) and they do not suspect the postman of any misbehavior.

The techniques used in this simple example give the feeling that some security is guaranteed; however, it is not the case. The famous Needham-Schröder protocol [NEE 78] has widely been used before a weakness was discovered [LOW 95]. Hence, efficient techniques are rather elaborate and complex.

10.3. Security in specific distributed systems

Section 10.2 has given evidence of some general security issues, which may happen during communications within a distributed system. These problems also occur in more specific frameworks, such as grid computing. The solutions used in such cases are adapted to the specifics of the system configuration, its hardware and software characteristics. The mechanisms used to guarantee the system security derive e.g. from the system operating on an open network (which can then possibly be accessed by external intruders), or not (as is the case when using an institutional grid).

10.4. Outline of Part III

This part aims to present the security problems inherent to distributed systems, as well as the techniques used to counter these problems. This part is divided into two chapters, the first showing the techniques specific to grid computing, while the other has a more general nature, addressing ciphering issues.

Chapter 11 tackles the security problem in grid computing. In such a context, security and safety issues are intertwined. The main types of grids are presented. They are dedicated to support different sorts of applications. Criteria and practical aspects are presented. The main grid computing systems use specific methods to address security problems, depending on the intended use of the system and on the particular attacks that the grid is vulnerable to.

Chapter 12 introduces cryptography and ciphering. First, the basic concepts are described, along with the expected properties of the ciphering systems. As intrusions operate with numerous mechanisms, the safety of a cryptographic system can be measured by criteria for resisting particular attacks. Two main classes of cryptographic algorithms are distinguished: symmetric algorithms and public-key algorithms. Chapter 12 describes their characteristics, how they operate, and details the classic algorithms.

10.5. Bibliography

[BER 01] BERARD B., BIDOIT M., FINKEL A., LAROUSSINIE F., PETIT A., PETRUCCI L., SCHNOEBELEN PH., *Systems and Software Verification. Model-checking Techniques and Tools*, Springer Verlag, 2001.

[CLA 00] CLARKE E. M., GRUMBERG O., PELED D. A., *Model Checking*, MIT Press, 2000.

[LOW 95] LOWE G., “An attack on the Needham-Schröder public key authentication protocol”, *Information Processing Letters*, vol. 56, p. 131–136, 1995.

[NEE 78] NEEDHAM R., SCHRÖDER M., “Using encryption for authentication in large networks of computers”, *Communications of the ACM*, vol. 21, p. 993–999, 1978.

[SAE 10] SAE, Architecture analysis & design language (AS5506), available at http://www.aadl.info, 2010.

Chapter 11

Practical Security in Distributed Systems

11.1. Introduction

Among the large number of available distributed systems, such as client-server, peer-to-peer (P2P) systems to name a few, this chapter focuses on one type of distributed system: namely, the grid paradigm. The concept, and some of its variants, will be introduced later in the chapter. For this architecture, we mainly discuss security and safety issues in this chapter. We cover the following topics after a general presentation of the notions:

– confidentiality;

– authentication;

– availability;

– ensuring resource integrity;

– result checking.

Chapter written by Benoît BERTHOLON, Christophe CÉRIN, Camille COTI, Jean-Christophe DUBACQ and Sébastien VARRETTE.

Distributed Systems: Design and Algorithms Edited by Serge Haddad, Fabrice Kordon, Laurent Pautet and Laure Petrucci
© 2011 ISTE Ltd. Published 2011 by ISTE Ltd.

11.1.1. *Grid typology*

Grid computing [FOS 01] is a term referring to the combination of computer resources taken from multiple administrative domains to solve a certain problem requiring a lot of computing power. We can say that grid computing is mainly about coordinating computation across domains. This is a key property. Grid computing is distinguished from conventional high-performance computing systems (HPC), such as cluster computing, in the sense that grids tend to be more loosely coupled, heterogenous, and geographically dispersed. Grids are often constructed on top of general-purpose grid software libraries known as *middleware*. In grid computing, many middlewares have been developed in recent years, such as Globus and Unicore. Many developments are still running under the supervision of the Open Grid Forum [1] (OGF). Large grids with dependable computing resources under the control of large supervising councils (the union of the members, each providing a significant amount of resources) are called *production grids*. Some of these grids have open access (for free, for an annual fee, for a per-job fee), others are not (they act as private clusters; they are sometimes called *enterprise grids*). *National grids* are another specific case, where the supervising entity is a public organization affiliated to the Ministry of Science (or equivalent).

A grid is characterized by its hardware infrastructure as well as the middleware it uses. The chosen middleware is therefore a key aspect of a grid definition. For instance, the Simple API for Grid Applications (SAGA) is an open standard defined and maintained by the Open Grid Forum (OGF) that describes a high-level interface for programming grid applications. In fact, SAGA is ultimately a set of OGF documents:

– SAGA use cases: an experimental OGF document describing the target use cases for SAGA;

– SAGA requirement analysis: another experimental OGF document that extracts specific requirements from the use case document;

– The SAGA core API specification: the basis of the standard, defining the look and feel of the SAGA API;

– SAGA API extensions: additional functional API extensions that use the look and feel of the SAGA API;

– SAGA API language binding: mapping of the language-neutral SAGA API to various programming languages.

OGF is a group of people promoting grid infrastructures. OGF also promotes standards and, more specifically, the subgroup working on security issues promotes the following standards:

– Certificate Authority Operations Working Group (caops-wg);

1. http://www.ogf.org.

– Firewall Issues Research Group (fi-rg);

– Levels of Authentication Assurance Research Group (loa-rg);

– OGSA Authorization Working Group (ogsa-authz-wg).

Contrary to production grids, *desktop grid systems* represent an alternative to supercomputers and parallel machines, offering computing power at low cost. The nodes of such distributed systems are made of personal computers (PCs), located at home and the underlying network is the Internet. The key idea is also based on volunteer computing: if a computer becomes available, then it downloads a code and data from a server, computes a result, then returns the result to the server. This concept has been popularized by the `Seti@Home` project from the University of Berkeley, over 10 years ago.

Desktop grid [KON 04, AND 02, CHI 03, LO 04] systems are attractive when running distributed applications with significant computational requirements. The `Seti@Home` [AND 02] project is one of the many success stories. While the increasing number of users of such systems does demonstrate the potential of desktop grid, current implementations, for instance BOINC [BOI10, AND 04], United Devices [2], Distributed.Net [3], and Xtrem-Web [FED 01, CAP 05b] still follow the client-server or master/slave paradigm. The computing power that can be obtained from these systems is constrained by the performance of the master node [4]. This is particularly the case for data-intensive applications. Then, depending on the performance of the master node, with thousands of workers (or slaves) and user applications, the central scheduler could become a bottleneck. Such a problem does not occur with a decentralized resources' management. Furthermore, this kind of platform requires full supervision by administrative staff who guarantee that the master remains operational. Although the master crashes infrequently and replication techniques can resolve this problem when it occurs, we still believe in the need for decentralized approaches, as with *PastryGrid*, in which case the control of faults becomes a challenging problem.

Indeed, desktop grids have important features that explain the large number of international projects aimed at better exploiting this computational potential. Many desktop grid systems have been developed using a centralized model. These infrastructures run in a dynamic environment and the number of resources may increase dynamically. Hence, the need for decentralization is becoming increasingly important. BonjourGrid [ABB 09] is a new decentralized approach of desktop grid systems. Its main objective is to provide a decentralized infrastructure of multi-coordinators, using the services offered by a publish/subscribe system. Unlike classical desktop grid

2. http://www.ud.com.
3. http://www.distributed.net.
4. Or set of master nodes, in some middlewares that allow replication.

systems, BonjourGrid can create a dynamic and decentralized execution environment for each user on-demand, based on existing computing systems, such as XtremWeb [CAP 05b], or Condor [THA 05], to run any kind of application without the intervention of a system administrator.

11.1.2. *General introduction to security*

The world of *grid computing* is made of a large number of computer systems that interact with one another to form a distributed system able to mobilize resources for some computation work, typically involving heavy calculation and data processing.

Even if grid computing usually involves the aggregation of resources that are geographically and administratively diverse, the main characteristic of grid computing is the *heterogenity of the resources*, combined with the *low bandwidth* of the weakest data links. Some parts of the grid may have high bandwidths (some high-end clusters are part of some grids, especially nationwide grids); the main fact is that connecting these local computing facilities at the software level is the goal of grid computing.

Grids often cross usual boundaries of trust in high-performance computing. Even if large companies are capable of building homemade grids by assembling their local facilities (what we shall call a *cluster of clusters*), grid computing often involves using public networks, when not using many[5] private desktop computers or state-owned or companies' office computers during "inactive" hours (what we shall call a *desktop grid*). From this description, it should be clear that private trust is difficult to assert.

More complex projects also involve cooperation between countries. For example, the following projects show that the success of these multinational initiatives must bring *security* to these users that come from very different locations:

– EuMedGrid (development of a transnational grid spanning both sides of the Mediterranean Sea);

– EGI (European Grid Initiative);

– BRIDGE (Cooperation between Europe and China).

What is desired by the users of grids is many-fold. The term *security*[6] covers security against [ISO05]:

– thieves and eavesdroppers (information theft, *confidentiality*);

– error-laden work (user-made errors, *integrity*);

– transient and chronic failures (machine-made errors, *availability*);

5. "Many" as in two millions users for the BOINC project.

6. Also *safety*, see Chapter 10. The distinction is blurry in grid computing.

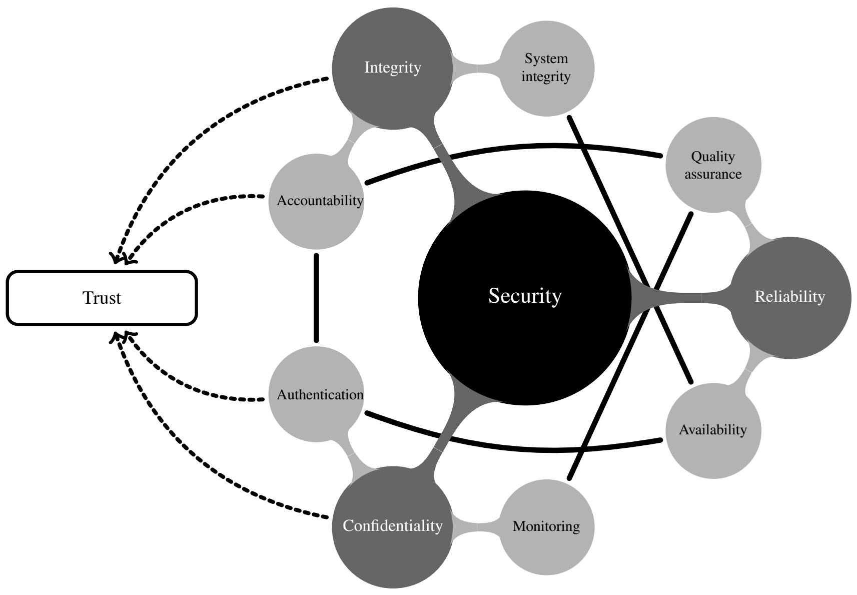


Figure 11.1. *Concepts linked to security*

– other factors (that will be detailed in section 11.1.2).

These problems are different facets of the same concept: *trust*. Establishing *trust* between the various human, hardware, and software components of the grid is the challenge of *Grid Computing*. Security is articulated around several ideas (see Figure 11.1).

The goal of security is to create trust between all legitimate parties and prevent other parties from accessing non-disclosed information. As stated above, parties need to create trust because grids extend over administrative domains. This trust is not given with the grid infrastructure; it has to be built both in the hardware and software assemblage that constitutes the grid network, but also in the P2P relations between the users and the managers of the grid resources.

Security in computer engineering is essentially threefold: reliability, integrity, and confidentiality.

Confidentiality is a major component of security. The data submitted by the users are often of great importance for the user, and the specificity of the grid [7] is the fact that resources are shared with other users. The contractual obligations of confidentiality is not sufficient to build trust between parties on its own merits. The technical interface must also prevent the leakage of any information that is not officially abandoned. The information can spread very quickly and some tools are usually not controlled or easily accessible to participants, such as the monitoring ofinformation (sometimes for good technical reasons, but this means that information about the activity of some user can be tracked, giving potential information about current research of the user). The volunteer computing aspect guarantees that some grids need exposure to a large public (since the volunteer base should be as large as possible).

Another problem in grid computing is creating high *reliability*. As the users of the grid need high computing power [8], the grid must be trustworthy, thus, reliable. Security lies in the usefulness of the available resources, if they are available for a cost. Reliability means controlling the availability of resources so that the user can gain access to more power with a grid than the power he can access without (especially massive computing power; massive data usage can also be undertaken, but moving around massive data requires high bandwiths, which a grid does not have, unless it is made of clusters of high-performance clusters). General efficiency is a security point in the sense that if user-submitted jobs cannot complete because of the lack of resources (either at start of the job or in the course of the computation), it renders trust in the system pointless. However, a policy of quality assurance (which includes, amongst other things, a quest for more power) and technical solutions for high availability of each resource makes a sound system. This also ensures that usable resources are given only to the participants of the project, which make authentication a crucial point of the systems. Reliability is ensured at the system level through the techniques of *monitoring*.

The question of *integrity* is more acute. The integrity must be devised by thinking from the resource owner's point of view (external computations must not damage the resources, and in some cases, these resources must be able to be removed from the grid infrastructure quickly); and also from the job submitter's point of view, which requires that the results are not tampered with.

Cryptography is a key point in ensuring integrity of data (using signature functions) and confidentiality (using encryption, either asymmetric or symmetric). The details of cryptography will not be described here, as they constitute the core of Chapter 12.

7. Especially in the desktop grid model, but more generally in all the shared grids models. Most of the grids are shared, except the rented clouds.

8. TeraGrid, for example, includes more than a petaflop of raw computing power.

11.1.3. *Case studies*

11.1.3.1. *OASIS*

As stated on their web site, “OASIS [9] (Organization for the Advancement of Structured Information Standards), is a non-profit consortium that drives the development, convergence, and adoption of open standards for the global information society”. The consortium produces web services standards including standards for security. The WS-Security 1.1 standard is the last release of the standard and dates from 2006.

Current grid middleware, such as Globus or gLite [10], provide functionalities that are based on WSS but we note that such specifications do not intensively investigate the problems of anonymity for instance. In contrast, we can find the following specifications promoted by the OASIS Security Services (SAML) Technical Committee:

– OASIS SAML (*Security Assertion Markup Language*) is an initiative that defines a standard mean for communicating in a secure way, information about authentication, and delegation of rights for the web services. SAML is designed to make “single sign on” (SSO), manually or automatically between systems. It allows participants to connect to a domain other than the original one, and it defines permissions and manages the exchange between them;

– OASIS XACML (*XML Access Control Markup Language*) is a complementary technology of SAML, which allows policies for the access control to be expressed.

The WSS Technical Committee also promotes the SOAP (Simple Object Access Protocol) communication protocol. The SOAP Message Security module describes enhancements to SOAP messaging to provide message integrity and confidentiality.

11.1.3.2. *Globus 4*

The Globus Toolkit is a well-know middleware that provides a set of packages for using, administrating, and managing computational grids. Since the third release of Globus, the toolkit has integrated the concept of web services (introduced in the previous subsection). The major challenge is to find the best coherency between web services and the specificities of grids. For instance, a typical user session in a computing grid typically follows the following sequence of actions:

1) authentication of participants;

2) submission of a job;

3) transfer of code and data to accomplish the job;

4) starting the application;

9. http://www.oasis-open.org/who/.
10. http://glite.web.cern.ch/glite/.

5) pick-up and transfers of results.

The *global security infrastructure* (GSI)[11] module of Globus 4 [FOS 97] is responsible for the security issues of the Globus middleware. It belongs to the "core service" layer, and is devoted to confidentiality, authentication, and non-repudiation.

A user session involves different resources. The underlying security problem is to ensure that they are all used by the same user. A unique authentication of the user must be maintained along the session. This mechanism is called SSO. This mechanism requires a delegation mechanism that allows some entity to use resources in the name of a user. We distinguish two types of delegation that must be efficient, restricted in time, and revocable:

1) *delegation of rights*, which allows data to be accessed with the same rights as a user;

2) *delegation of responsibility*, which allows jobs to be started.

For all these operations, GSI relies on public key cryptography and more precisely on a Public Key Infrastructure (PKI). Each entity (user, resource, task) is associated with a digital certificate asserted (i.e. signed) by a trusted third party. This ensures a homogenous global vision of the grid components. From a technical point of view, the GSI certificates are encoded into the X.509 format, which is a standard of the *Internet Engineering Task Force* (IETF). From a conceptual point of view, if two entities have certificates and if these entities trust the Certification Authority (CA), then both entities can mutually prove to each other that they are what they pretend to be. This will be detailed in section 11.3.5. The mechanisms for delegation are obtained, with GSI, according to the delivery of delegated certificates and temporary certificates (they typically expire after 12 or 24 hours). They are generated and signed on demand by another certificate such that they can act as proxies of their signer. This mechanism is standardized and is the subject of a specific RFC (Request For Comments) [TUE 04]. In a distributed context, they ensure SSO for the users at each step of their grid usage. Therefore we can find *user proxies* (UPs, that act under the rights of a given user) and *resource proxies* (which act for resources or services). A trust path is exhibited between all these entities as shown in Figure 11.2.

To interact with the local security policy of each involved institution, a mechanism to establish in a unique and reliable way a correspondence between a global identity and a local one is required. This enables the local policy to be applied so as to authorize (or not) the access to the resources managed by the local institution. In practice, two approaches are available:

– using a correspondence table called *grid-mapfile*;

11. http://www.globus.org/toolkit/docs/4.0/security/.

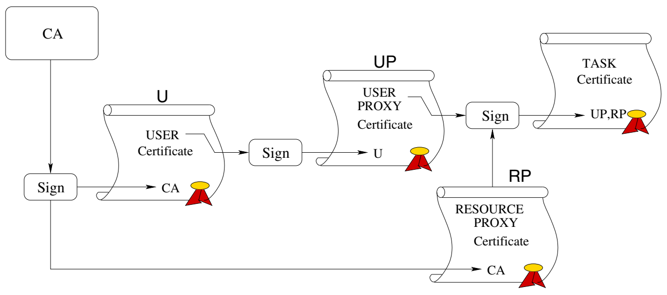


Figure 11.2. *Trust path in GSI between the grid entities.*

– using the SAML standard so as to request the Community Authorization Service (CAS) responsible for allocation policies.

Furthermore, the GSI module is built according to the model depicted in Figure 11.3, which illustrates the different steps of secure task execution on a grid composed by two virtual organizations. It assumes that the correspondence mechanism between global and local entities has been configured. Then, a UP is created such that it can act in place of the user, which minimizes the circulation of the real user credentials on the network. All allocation and scheduling strategies of a local site are handled by a resource proxy (RP). This interacts directly with the UPs. After a mutual authentication between them (detailed in section 11.3.5), the RP can check the local access policy to authorize or deny the UP. It follows the creation of the tasks to be executed on the grid, each of them identified by a certificate signed by both the UP and RP.

Finally, Globus 4 uses the SOAP service to implement secure communication over TLS (a variant of SSL). Moreover, secured data transfers are performed using GridFTP. GridFTP can use GSI to ensure authentication and secure the communications. It is an extension of the popular *File Transfer Protocol* (FTP) with communication strategies that are particularly adapted to the communication media that are available on grids (e.g.data stripping to aggregate bandwidth). The *Reliable File Transfer* service (RTF) uses GridFTP to enable file transfer as a service. For example, the user can provide a list of files to be transferred, and disconnect from the grid. The data transfers will be performed on the grid in batch mode. One interesting feature of RFT is its fault-tolerance capabilities. Replicas can be used to avoid having the data on a single server on the grid. If the server fails during a file transfer, the client knows from which point it must be resumed. If the client crashes, RFT provides a way to recover and resume the transfer from a previous point. It can also handle network failures with retries with exponential backoff.

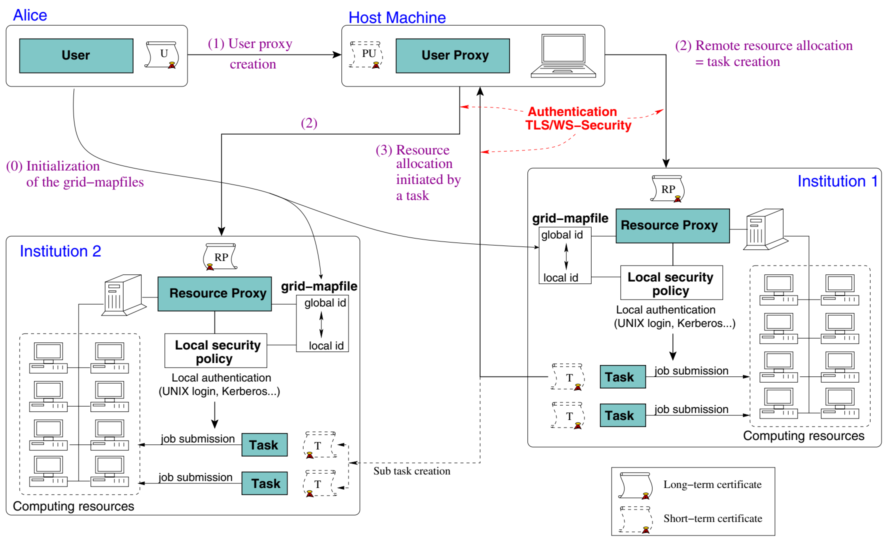


Figure 11.3. *Overview of the Globus GSI module*

11.1.3.3. *ALADDIN-G5K*

ALADDIN-G5K is a testbed for experiments in large-scale grid computing attached to the French academic community [CAP 05a]. Grid'5000 (the first name of the project, ALADDIN being the name of the continuation of the project) is a multisite (10 sites located in metropolitan France) federation of clusters (ranging from 236 to 684 nodes; 736 to 1804 cores). The clusters are heterogeneous (different processors and network types). ALADDIN-G5K is different from the other case studies presented here in the sense that it is not a grid framework (or middleware), but a real grid (with hardware attached to it). Some features detailed here pertain to the physical infrastructure adopted for the project.

The target community of this grid is more specifically the researchers in grid computing that need a great variety of highly reconfigurable nodes, sometimes with root access. The advantages of Grid'5000 are the following:

– a reservation system capable of managing the complexity of the heterogenity of the grids: OARGRID (a grid-enabled version of OAR, a reservation system for a cluster);

– a system to quickly reconfigure thousands of nodes with complete reinstallation from a disk image with root access: `kadeploy` (which uses `taktuk` for deployment);

– a global network infrastructure designed to protect experiments from outside interference and the Internet from the power of what could amount to a small bottleneck.

Grid'5000 is not a production grid: the reservations or usage are not paid for (either in money or reciprocity) and the hardware support is done at the national or regional level through public grants and funding. The nodes may be reserved for specific large-scale experiments by mutual agreement of the executive board. This grid is quite similar in some aspects to the production grids of many national academic grids, for example the National Grid Service (NGS) in the UK. The underlying communication network relies on standard Unix protocols to communicate between clients (ssh access to individual nodes) on a ready-made disk image. The bandwidth is provided by Renater [12], which provides a very solid service quality but comes with a network usage charter with many constraints (which are not problematic in this professional example).

Each cluster is designed to have several nodes behind a single front-end that has access to user accounts by NFS (individual nodes can also access data through NFS, but this prevents scalability and work on the data is done on local disks). The front-ends are the only entry points to the clusters (including from a network point of view), and serve as outgoing proxies. This helps to monitor the full Grid'5000 charter (which states that no external services should be provided through the grid, and this means not being able to create distributed denial of services; the external connectivity of the cluster is maximized by the proxy). The front-end node also acts as a router to the other sites of Grid'5000 (with a large bandwidth [13] and specific physical links entirely dedicated to the connectivity of the cluster of clusters). The physical links are *dark fiber* to ensure physical separation of inter-cluster communications and general Internet traffic.

The versatility of this grid is further enhanced by the availability of several and even customizable disk images for booting (large disk images can be stored in the user's account). The software that can reinstall and remotely reboot machines so that they get is called `kadeploy` [KAD05]. At reservation time (or later by shell access, which is the most used method by large), a disk image that is available on the front-end (by NFS) can be specified and a machine can be rebooted on this disk. Physical cards inside (most) nodes enable monitoring of the remote boot process as if watching from the terminal. This tool uses `taktuk` [CLA 09] to deploy itself very quickly (`taktuk` uses work-stealing techniques to share the sending data the cluster).

OARGRID offers the possibility to use several clusters installed with the OAR batch scheduler. It coordinates cross-cluster reservations, manages the resources at grid level by coordinating the various cluster-level resource managers and federates

12. Réseau national de télécommunications pour la technologie, l'enseignement et la recherche / National Telecom Network for Technology, Teaching and Research.

13. Most inter-cluster links are at 10 Gb/s, some are only at 1 Gb/s.

all the resources in a single job. OAR enhances security and confidentiality between users by integrating SSH with OAR, through the `oarsh` communication utility. Among other things, `oarsh` verifies that the source and destination machines involved in a communication are part of the same job. Hence, people cannot connect to machines that have been reserved by other people. When a reservation has been issued by `oargrid`, all the cluster-level reservations are considered to be part of a single job, and `oarsh` allows communication with one another across the grid.

11.1.3.4. *BOINC*

BOINC [BOI10, AND 04] is one of the reference middlewares that exploits desktop grids and, therefore, perfectly illustrates the management of a volunteer computing (VC) system. Reusing the concepts of the historical SETI@Home project, BOINC enables the idle cycles of our CPUs (or GPUs) to be shared for many volunteer computing *projects* (and not only one as was the case for SETI@Home). These projects are typically based in universities and research labs, and it is up to the user to decide to participate in any number of these projects. The general BOINC architecture is illustrated in Figure 11.4.

From a client's point of view, contributing to BOINC consists of downloading and installing the BOINC software. This software is divided into several elements:

– the BOINC client (or core client), which takes care of communications with the BOINC servers. It also downloads application code for the projects and the input data files, ensures the binaries are up-to-date, and schedules CPU (or GPU) resources between the different projects (assuming several ones have been installed). Once a job is computed, the BOINC client is also responsible for uploading the output files to the project server and reporting the results;

– the BOINC manager, which provides a GUI interface to control BOINC clients on the same machine or, if necessary, on others (which makes the BOINC integration in cluster environments easier);

– a screensaver that notifies the core client about idle periods.

On the project server's side, the two main software components are the scheduler (responsible for scheduling application jobs among different clients, checking job results, etc.) and the data servers that take care of hosting the application code and input data files. BOINC uses digital signatures to allow the core client to authenticate and verify these files. Additionally, the data server collects the output files. Finally, the project's server keeps track of how much work (or *credit*) has been done by each client.

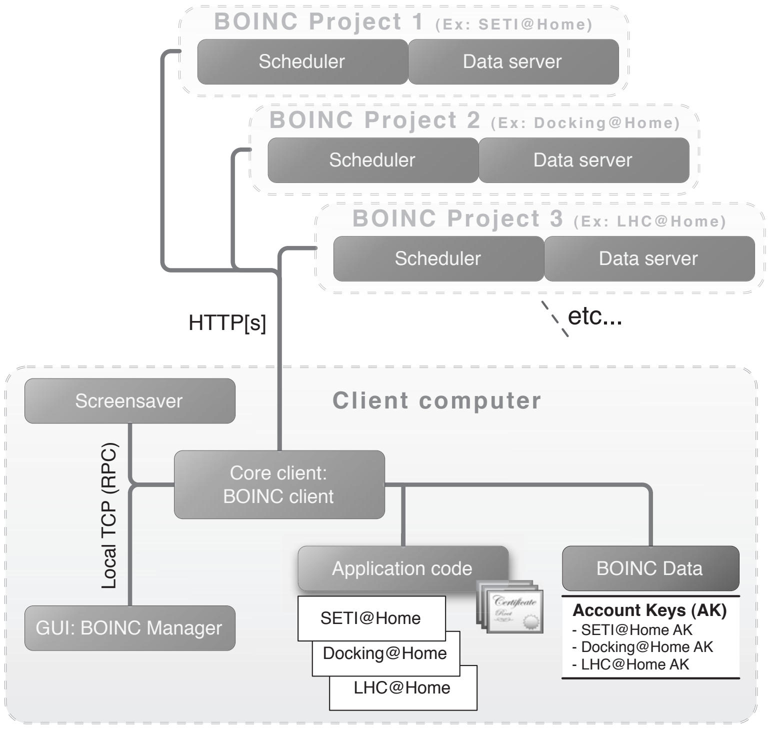


Figure 11.4. *General software architecture in BOINC*

11.2. Confidentiality

Large-scale computing platforms are subject to various threats and security issues. Among them, confidentiality is probably one of the most difficult properties to address, but also the least studied domain in the grid context. The reason for this last statement is mainly due to the historical *open* characteristic of the distributed architectures. It obviously applies to VC systems, but also to most clusters and computing grids where the confidentiality of the data put on the grid or the privacy of the computation performed were not considered as a critical feature. What makes confidentiality so hard in the distributed context comes from the following facts:

– it is impossible to fully trust a remote software that runs on a resource where we have no complete control;

– encryption schemes, especially to protect code execution, are still under heavy investigation;

– last, but not least, formal *proof* of confidentiality remains very hard to obtain, even in the general case.

The purpose of this section is to provide a brief overview of the state-of-the-art techniques applied in this domain.

11.2.1. *Data confidentiality*

Encrypting the data transferred to avoid eavesdropping from external attacker spying on the communication channel can be easily done using e.g. various protocols over a TCP/IP network (TLS [RES 01, DIE 99b], IPSec [NOR 03] or SSH [BAR 01b]). It becomes trickier to protect data manipulated before, during, and after the execution of a program from an attacker having full access to the computing node (and therefore to the registers, the RAM, the hard-drive, the CPU controllers etc.). Actually this is one of the reasons companies are so reluctant to use public, large-scale computing platforms.

One approach is called encrypted computations. Cryptography is described more thoroughly in Chapter 12. The idea is to transform an algorithm into an isomorphic algorithm that acts on ciphered input and produces ciphered output instead. Some encryption systems are already homomorphic for one operator, for example unpadded RSA [KAL 98] is homomorphic for the multiplication, which means that anybody multiply two encrypted messages c_1 and c_2 without knowing the messages m_1 and m_2. This property is intrinsic to RSA encryption/decryption scheme. Considering the two encrypted messages using the public key (n, e):

$$\begin{aligned} c_1 &= m_1^e \texttt{ mod } n \\ c_2 &= m_2^e \texttt{ mod } n \end{aligned}$$

The product of the two previous messages is the following:

$$c_1.c_2 = m_1^e.m_2^e = (m_1.m_2)^e \texttt{ mod } n$$

This product corresponds exactly to the encryption of the message $m_{1.2} = m_1.m_2$:

$$\begin{aligned} c_{1.2} &= (m_{1.2})^e \texttt{ mod } n \\ &= (m_1.m_2)^e \texttt{ mod } n \end{aligned}$$

The owner of the private key then has access to the multiplication of the two messages without the need to decrypt messages m_1 and m_2. Similarly the Paillier encryption system [PAI 99] is homomorphic for the addition. But until 2009, no cryptographic scheme was homomorphic for both multiplication and addition. Craig Gentry developed in his thesis [GEN 09] a fully homomorphic encryption scheme using lattice-based public key encryption.

It is also possible to add some authentication mechanisms to certify that the computation is done. Efficient isomorphic algorithms exist for some simple algorithms (i.e. relying on very small set of instructions) but, unfortunately, this seducing approach has an intractable complexity for real-life programs [LOU 02]. Since basic computations (such as, for example, floating point operations that are highly optimized in modern processors), are transformed in elementary operations executed to emulate some enciphered circuits, the expected efficiency on grids is no longer possible. Further effort is required to find novel ways of dealing with this issue and it is one of the current academic challenges in this domain.

11.2.2. *Code confidentiality*

Furthermore, if industrial companies were so sceptical about the use of grid computing environment, it is not only because of the data privacy issues but also of executing confidentiality problems. Indeed the main fear resides in reverse engineering of proprietary code, motivating major investments in buying expensive private computing clusters where companies keep full control. These aspects become of prior importance over the cloud computing paradigm as user's programs are executed successively (and eventually concurrently) on computing nodes where the user has no complete control (and potentially trust). Researches in the domain of execution privacy are mainly limited to studying embedded environments, such as java cards [COL 02] which are limited by huge constraints in terms of the atomic operations elligible in the program.

In addition to encrypted computation mentioned above, other approaches, such as code obfuscation (where the code is transformed at the source or binary level to render its readability more complex and also change the data coding). However, Barak *et al.* [BAR 01a] have proven that a "virtual black box" obfuscation is impossible. Nevertheless, time-limited blackbox security could be used [HOH 98]. However, evaluating the security of these schemes assumes a metric able to quantify the level of obfuscation achieved i.e. the difficulty of the reverse engineering task required to recover the initial code; currently, such a metric does not exist.

11.2.3. *Other aspects of confidentiality*

As stated in section 11.1.2, other aspects can be leaked that may matter to the user. The monitoring facilities of large grids often require either no authentication or only basic authentication that can be easy to unravel (by submitting resources in a volunteer project). Some projects are even constitutionally bound to publish information.

The monitoring information may contain three kinds of valuable information: the user list, information regarding their jobs, and the nature of the resources (probably the least important of all). User lists may be readily available through the authentication scheme. LDAP access is often available. Also, job information most often cites the project or the individual that made a reservation, and therefore provides a list of the active users. Job information in itself may give some indication of the work being achieved. In heterogenous grids especially, the kind of job that has been submitted can be guessed by cross-referencing the choice of nodes (RAM, power, and quantity), the installed system, the activity of the processor, and of the network. This can be hidden by steganographic use of the resources, but in grids, a kind of monetary-like accounting is often done. The information about the resources is often given, and is a good element for advertising the grid. The openness of the monitoring systems may, however, signal a problem with availability, which can damage the trust of the users. Also, as written above, the resource statistics can help to guess the nature of the work.

11.3. Authentication

Authentication is one of the key components of distributed systems as its permits the allocation and the access to the resources while ensuring monitoring of their usage by the users. The security of this component is of course essential (for instance to prevent usage abuse) yet the security level strongly depends on the type of distributed system considered (see section 11.1.1):

– *desktop grids* (or volunteer systems) rely on nearly anonymous resources, so authentication at this level only ensures usage monitoring, for instance to fill the well-known *hall of fame* of these systems that ranks the users depending on their contribution;

– *clusters* and *institutional grids*, on the contrary, assume a strong and secure authentication system able to deny unauthorized users access to resources. This applies also in the context of the *cloud* paradigm that typically involves money-oriented incentives.

After providing a quick overview of classical authentication schemes (in section 11.3.1), we detail the authentication systems used in the above classes of distributed systems.

11.3.1. *Overview of classical authentication schemes*

Formally speaking, we focus in this section on *entity authentication* generally defined as the process whereby one party is assured (through acquisition of corroborative evidence) of the identity of a second party involved in a protocol, and that the second has actually participated (i.e. is active at, or immediately prior to, the time the evidence is acquired) [MEN 96].

In our case, the considered entities are the user and the resource of the distributed system.

Entity authentication techniques may be divided into three main categories, depending on which of the following the security is based:

1) *something known*, such as a password, Personal Identification Numbers (PINs), or a private key whose knowledge is demonstrated in challenge-response protocols;

2) *something possessed*, such as a physical accessory (a smart card for instance) or the private part of a digital certificate;

3) *something inherent (to a human individual)*, which mainly applies to biometric-based authentication, which is not considered here.

Based on these categories, we now describe the most common entity authentication schemes. For more details, the reader should refer to [MEN 96].

Password-based authentication

Conventional password schemes involve time-invariant passwords, which provide so-called *weak* authentication. The user submits a pair (userID, password), and the system checks that the password matches corresponding data it is holding for that userID, and that the stated identity is authorized to access the resource. Demonstration of knowledge of this secret (by revealing the password itself) is accepted by the system as corroboration of the entity's identity.

On the system, the passwords are generally salted, i.e. each password, upon initial entry, is augmented with a t-bit random string (the *salt*) before applying a one-way function. This is the case, for instance, with traditional UNIX passwords. The salted passwords are then stored (eventually encrypted) in files (`/etc/passwd` or `/etc/shadow` typically on UNIX systems) or databases. Time-invariant password-based authentication have been qualified as *weak* schemes: a major security concern is eavesdropping and subsequent replay of the password. Even without considering eavesdropping, such schemes are subject to password-guessing and dictionary attacks, which are more than common against interconnected systems as considered in this chapter.

A partial solution to these issues are *one-time passwords* (OTPs): each password is used only once. Several approaches are possible at this level: shared lists of OTPs, sequentially updated OTPs (e.g. using hash chains linked with S/Key [HAL 95]) or OTP sequences based on a one-way function such as Lamport's scheme [LAM 81]. It is worth mentioning that many OTP technologies are patented, even if some standardization efforts exists (see for example RFC 2289 [HAL 98], RFC 4226 (HOTP) [M'R 05] or the RSA Labs OTP standardization proposal (OTPS) [OTP05].

Challenge-response authentication

The idea behind cryptographic challenge-response protocols is that one entity (the claimant) "proves" its identity to another entity (the verifier) by demonstrating knowledge of a secret known to be associated with that entity, without revealing the secret itself to the verifier during the protocol. This is done by providing a response to a time-variant challenge, where the response depends on both the entity's secret and the challenge. The challenge is typically a number chosen by one entity (randomly and secretly) at the outset of the protocol. Several variations of this scheme exists, some based on a symmetric-key technique, other on public-key approaches (typically via certificates) or on zero-knowledge concepts. The details of these variants are outside the scope of this chapter, even if some protocols based on challenge-response schemes will be presented in the sequel. Some are described in Chapter 12. We encourage the interested reader to refer to ISO/IEC 9798 [ISO10] parts 2 through 5 which specify entity authentication protocols respectively based on symmetric encryption [ISO08], digital signatures [ISO98], keyed one-way functions [ISO99], and zero-knowledge techniques [ISO09b].

Among the most used challenge-response protocols (involving random numbers and digital certificates), the standards FIPS 196 [U.S97] and TLS [DIE 99a] can be cited. Variation used in the authentication protocols are presented in the following sections.

11.3.2. *Authentication in the main operating systems*

All systems at the basis of distributed computing systems run an operating system (OS) – UNIX-like or Windows – that integrate authentication mechanisms. This section briefly discusses the availability and the extensibility of these mechanisms.

Authentication in UNIX *machines*

UNIX is a family of extremely mature operating systems, recognized for the quality of their architectures and their stability. It includes many commercial variations, such as Solaris from Sun, HP-UX from Hewlett Packard, Irix from Silicon Graphics (now SGI), AIX from IBM, Mac OS X from Apple, as well as open source versions like Linux, FreeBSD, NetBSD or OpenBSD.

The management of the access rights of the file system (files and repertories) is associated with the concept of privilege, but also with the concept of a root administrator having all the privileges on the system. Another strong point of the UNIX file system security is the possibility to isolate a user or a process in a defined portion of the file system tree.

Traditional UNIX authentication is performed using a salted password scheme as described in section 11.3.1. Some years ago, the authentication mechanism was hard-coded inside programs that needed to identify and authenticate the user, so any modification to the authentication scheme was very tiresome as all these programs had then to be modified (if they were open-source) or replaced.

To allow the easier replacement of the authentication scheme, some proposals have been implemented in some systems, such as *BSD auth* or NSS [GNU 01], but these schemes still needed the modification of a library to add a new mechanism. The real advance was the introduction of *PAM (Pluggable Authentication Modules)* by Sun in Solaris. With this system, adding a new authentication scheme only requires the addition of a file (a library implementing some basic primitives for this new scheme), and the modification of some configuration files to use it! Neither more recompilation nor modifications of the existing program is needed. This scheme was then integrated by the X/Open Group in the XSSO proposal [GRO 97] and informally standardized. PAM is currently used by most major UNIX systems, such as Linux, Solaris, Mac OS X, or FreeBSD.

Nowadays most authentication schemes are available as PAM modules, in particular for advanced network authentication protocols (Kerberos) and naming service based-authentication (NIS, NIS+, LDAP, etc.) that are used in clusters as it will be seen in section 11.3.4.

Authentication in Windows machines

Recently, Microsoft Windows imposed itself on the desktop computer market and as a possible alternative to UNIX for server systems. Initially, no credible authentication mechanisms existed on Windows desktop versions. Current versions based on Windows NT address this problem.

As for UNIX systems, it is possible to open an interactive session, authenticated by the local user repository called a SAM database (Security Accounts Manager) or with network authentication, by using a user accounts repository stored on a network server (mechanism implemented for the Windows domains).

In practice, a user stored in the users repository of the domain can connect to any workstation of the domain, having the choice between local authentication, which enables working in standalone mode without benefiting from networks services provided by the company, or an authentication on the domain, which affords access to the information system.

The network authentication can be managed by the old NTLM (NT LAN Manager) protocol or by Kerberos (see section 11.3.4), which is the default authentication mechanism for such sessions in Windows 2000, XP, and later versions. The domain controller, which acts as a Kerberos server, can use a users database stored in the active directory.

11.3.3. *Authentication in volunteer grids*

As mentioned above, desktop grids rely on nearly anonymous resources. Authentication at this level only ensures usage monitoring, for instance to rank the users depending on their credits, i.e. their contribution. It follows that the authentication schemes are generally weak on these systems, as in BOINC.

An overview of BOINC middleware has been given in section 11.1.1. Authentication in BOINC works on a per project basis: joining a project comes with the creation of an associated *account key*, i.e. a 32-byte random and unique identifier string that allows the participant to identify itself on the project server. The account key is issued upon request and delivered to the person that made the request through the email address provided when joining the project. All the account keys are stored locally in the BOINC data folder (as illustrated in Figure 11.4 page 249). They are used to establish the identity of the participant to the project servers during each communication session. Of course, the usage of any sniffing tool can permit an attacker to easily recover the account key so as to impersonate the user on the project server. It explains why this scheme has been qualified as "weak".

It should be noted that the current policy for most BOINC projects is that the loss of an account key will require the establishment of a brand new account and all the work effort in the other account stays there.

11.3.4. *Authentication in clusters*

As most clusters run a UNIX-like operating system (95.6% of the systems represented in the Top500 project[14], we focus here on authentication systems for this environment. Without pretending to be exhaustive, we list the most used systems.

Kerberos

Kerberos [NEU 94, NEU 04] is a network authentication protocol (today IETF standard). It has been designed to provide strong authentication for client/server applications by using secret-key cryptography. Many implementations of Kerberos exist, the most well known remains the open source one developed at MIT [KER]. Under Kerberos, a client (generally either a user or a service) sends a request for a *ticket* to the key distribution center (KDC). The KDC creates a ticket-granting ticket (TGT) for the client, encrypts it using the client's password as the key, and sends back the encrypted TGT. The client then attempts to decrypt the TGT, using its password. If the decryption is successful (i.e. if the client provided the correct password), a private session key can be recovered that will be used later as a proof of the client's identity. The TGT and its integrated session key are time-stamped: they expire at a specified time. They allow the client to obtain additional tickets, which give permission to access specific services. The request and granting of these additional tickets is user-transparent, i.e. the SSO feature is ensured.

14. http://www.top500.org.

As the Kerberos protocol is supported by most operating systems including Windows, it is often used in confined environments such as clusters. Various extensions and improvements of the Kerberos infrastructure have been proposed in the literature. Among them, it is worth mentioning KryptoKnight [MOL 92, BIR 95], a network authentication protocol designed by IBM and similar to Kerberos. It uses message authentication codes (MAC) to identify and encrypt the tickets. This permits the size of the protocol messages to be reduced and adds interesting integrity features.

Whether for Kerberos or for KryptoKnight, scaling to computing grids is not yet operational for these systems. Yet within a local institution those authentication system relying on Kerberos can still be integrated into a Globus infrastructure (see section 11.3.5) as there is an option in the GSI module of Globus to use Kerberos credentials locally.

Network authentication based on naming services

NIS

Introduced by Sun in 1985, the Network Information Service (NIS) is used to centralize the administration of systems information. The information is stored in maps under indexed databases (db, dbm) reachable by RPC[15]. Based on a master/slave model, NIS does not allow the treatment of important volumes of data as each modification involves the transfer of the totality of the base. Furthermore, it is particularly hard to organize the data in a hierarchical way and the access security remains weak. In spite of all these drawbacks, NIS remains a well used system at the level of clusters and local networks, mainly because it is simple to install.

NIS+

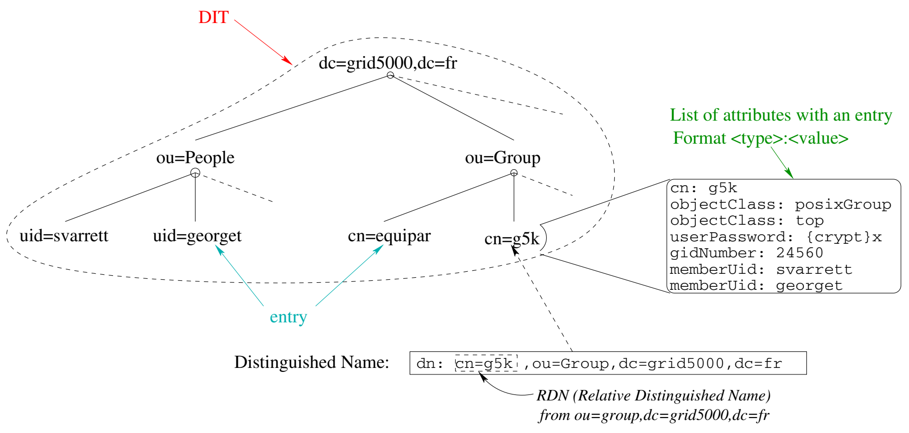
NIS+ was SUN's answer to the drawbacks of NIS. NIS+ introduces the distribution of the data between master and slave servers in an incremental way, in particular by adding the notion of hierarchical tree for the data. Many of the security issues of NIS were addressed by the introduction of certificates. Yet, the lack of flexibility in the hierarchical structure together with a complicated installation procedure have slowed down the transition from NIS to NIS+.

Directory-based authentication using LDAP

A directory is like a database, but tends to contain more descriptive information. Directories are tuned to give quick-response to high-volume lookup or search operations. They may have the ability to replicate information widely in order to increase availability and reliability, while reducing response time.

15. Remote Procedure Call

Based on the X.500 protocol (ISO standard for the management of electronic directories), LDAP [WAH 97] (*Lightweight Directory Access Protocol*) is a lightweight version of the specialization of this standard adapted to TCP/IP networks. Created in 1993, LDAP is an access protocol to electronic directories allowing researches and modifications to be performed. LDAP is based on the model of DNS: data are naturally organized in a tree structure and each branch of the tree can easily be distributed among different servers. LDAP technology has been adopted by many large companies. The generalization of LDAP has also been implemented in the applicative bases (some operating systems, like Mac OS X or Solaris 9, integrate a LDAP directory). In terms of security, LDAP provides various guarantees thanks to the integration of cypher and authentication standard mechanisms (SSL/TLS and more importantly SASL [16], allowing easy integration of new authentication schemes such as Kerberos) coupled with Access Control Lists. These mechanisms enable efficient protection of transactions and access to the data incorporated in the LDAP directory. Thereafter, practical experiments on LDAP were carried out through an open-source and reliable implementation: OpenLDAP [17]. LDAP data are organized in a tree structure called Directory Information Tree (DIT). An example of DIT is provided in Figure 11.5. Each node of the tree corresponds to an entry of the directory and is referred to in a unique way by its *distinguished name (DN)*.



```
cn: g5k
objectClass: posixGroup
objectClass: top
userPassword: {crypt}x
gidNumber: 24560
memberUid: svarrett
memberUid: georget
```

```
dn: cn=g5k ,ou=Group,dc=grid5000,dc=fr
```

Figure 11.5. *Example of DIT: users management*

The directory service provided by LDAP is based on a client/server model. Yet, the servers can be organized in various configurations:

16. Simple Authentication and Security Layer.
17. http://www.openldap.org.

1) *local directory service*: only one server is able to deal with all the clients' requests;

2) *local directory service with referrals*: the server is configured to provide directory services for a local domain and to return referrals (i.e. a pointer) to a superior service capable of handling requests outside the local domain;

3) *replicated directory service*: partial replication can be operated between master and slave servers;

4) *distributed local directory service*: the database is divided into subparts (eventually replicated) that are accessible through a set of referrals between the servers.

The last two modes are particularly interesting in the context of authentication in clusters grids, mainly because in the context of a distributed environment, the authentication system has to address the following constraints:

– availability: the system should work even in case of punctual disconnections;

– security: privacy and integrity of the data should be guaranteed;

– delegation: each administrator of a site should be able to manage its own users.

Therefore the idea is to use LDAP as a naming service for the broadcast of system information which can then be used for authentication purposes via the appropriate PAM module. The tree structure used to store the data in the LDAP server follows the organization of the sites in the grid: each site is responsible for a sub-tree containing data relative to the users and the resources of the site. Figure 11.6 illustrates this architecture, which has been applied in the framework of the ALADDIN-G5K project presented in section 11.1.1. In particular, as demonstrated in [VAR 05], a partial replicated approach is able to solve the problem of availability. Security is ensured by the protocol LDAP itself whereas delegation is due to the tables' distribution in the proposed architecture. This approach is therefore a particularly good candidate for a robust authentication system in a distributed cluster environment.

11.3.5. *Authentication in computational grids*

At this level, public key cryptography (also known as asymetric cryptography) is used as as the basis for every functionality. This is the case in Globus (via its GSI module) or gLite. The primary motivations for this are:

– the need for secure communication (authenticated and perhaps confidential) between elements of a computational grid;

– the need to support security across organizational boundaries, thus prohibiting a centrally-managed security system;

– the need to support "single sign-on" for users of the grid, including delegation of credentials for computations that involve multiple resources and/or sites.

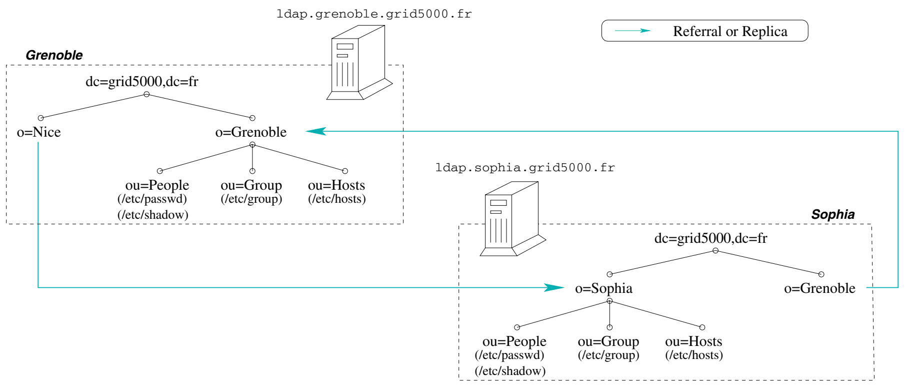


Figure 11.6. *Flat distribution for an authentication system based on LDAP in a grid composed of two clusters, one located in Grenoble (France), the other in Sophia (France). This configuration is used in the Grid5000 project for instance.*

As mentioned in section 11.1.3.2, each entity in Globus owns a certificate. The GSI modules use the secure sockets layer (SSL) for its mutual authentication protocol, which is described below. Before mutual authentication can occur, the parties involved must first trust the CAs that signed each other's certificates. In practice, this means that they must have copies of the CAs' certificates – which contain the CAs' public keys – and that they must trust that these certificates actually belong to the CAs. To mutually authenticate, the first person (A) establishes a connection to the second person (B). To start the authentication process, A gives B his certificate. The certificate tells B who A is claiming to be (the identity), what A's public key is, and what CA is being used to certify the certificate. B will first make sure that the certificate is valid by checking the CA's digital signature to make sure that the CA actually signed the certificate and that the certificate has not been tampered with. (This is where B must trust the CA that signed A's certificate.) Once B has checked out A's certificate, B must make sure that A really is the person identified in the certificate. B generates a random message and sends it to A, asking A to encrypt it. A encrypts the message using his private key, and sends it back to B. B decrypts the message using A's public key. If this results in the original random message, then B knows that A is who he says he is. Now that B trusts A's identity, the same operation must happen the other way around. B sends A her certificate, A validates the certificate and sends a challenge message to be encrypted. B encrypts the message and sends it back to A, and A decrypts it and compares it with the original. If it matches, then A knows that B is who she says she is. At this point, A and B have established a connection to each other and are certain that they know each other's identity. In practice, authentication and confidentiality of communication transfers are combined in the GSI-OpenSSH [18] service.

18. http://dev.globus.org/wiki/GSI-OpenSSH.

11.4. Availability and fault tolerance

11.4.1. *Monitoring*

Especially with large-scale systems, monitoring is a key element to ensuring that confidentiality is unbroken. However, monitoring also encompasses such things as availability accounting, and provides essential feedback to the users of these systems. Monitoring can be seen from essentially two points of view: information gathering on remote systems and intrusion detection.

System monitoring

Monitoring can proceed in two modes: active polling, passive information gathering (of course, some hybrid mechanisms were devised).

Using syslog. System monitoring can be very simple; the UNIX `syslog` mechanism, for example, is used almost everywhere. This service can be configured to forward information through the network. As this is a clear-text protocol, the nature of information sharing on the grid will possibly require the use of SSL or SSH-enabled tunnels. Windows uses a similar mechanism called Windows Event Log. However, this information is fragile; administrator rights are sufficient to tamper with these. The usual safe way is to remotely send every log line to a central server with different authentication credentials; this prevents altering past records.

Passive monitoring by check-summing. The difficulty of detecting tampering at the system level is that once a system has been broken, the perpetrator usually has the means to delete any tracks left by the intrusion. Efficient monitoring relies on fast distribution of any information and frequent check-summing of the monitoring to detect alteration of the log files. As all the chain of monitoring, logging, and computation must be kept intact, a file analysis system (using either signature or simple check-summing) is a standard part of a monitoring chain.

Passive monitoring by hypervisor. Nowadays, virtualization techniques are sufficiently evolved for the computation to be achieved in virtual machines. The host system is capable of monitoring the guest system and uses any other technique to transmit the information.

Using active monitoring. The polling techniques are more intrusive, but they allow for better failure detection. However, they exploit more resources as they are not integrated in the normal flow of operation of the systems. Moreover, as they interact on distinct systems, they may be able to detect tampering not detected by passive methods. They can also gather other kinds of data, especially numeric values (average load, usual network statistics, etc.). The most well-known program for this is *Nagios* (and derivatives). Most of these programs are completed with an interface to present the data in an efficient way. The efficient representation of monitoring information coming from thousands of machines is a difficult task. This is, for example, the task of the *Ganglia* software (developed at UC Berkeley). This will be detailed in section 11.5.1.

Monitoring confidentiality. As one can see, the monitoring data itself may convey some information about the processes being run, some authentication information, and many other trivial or important data (some custom systems logged the attempted passwords, which were then transmitted in clear-text to the monitoring software *through the Internet*).

Network monitoring

The goal of network monitoring is the same as system monitoring: providing feedback about the usage (bandwidth usage may also be a limitation or a bill as well as the CPU usage in large-scale distributed systems); and preventing or signaling intrusions in the networks. Due to the high usage of network in a typical grid application, it is not possible to log every network packet. Network delays also typically imply that passive monitoring is much more intrusive than in the system case (where input/output delays allow the treatment of logging inside the delays). Pure data gathering can take place using the same methods as system monitoring, as long as they are accumulated data (or averaged data, which is the same). Network monitoring thus has to revolve around statistical analysis (in real time or *a posteriori*), or based on patterns analysis (trying to detect unusual behavior in the packets going through). These can take place at each node, or (more generally) in the network infrastructure (firewall, routers; see e.g. [SCH 07] for a good survey of these methods).

Firewalls. Firewalls are a normal part of a network intrusion detection system; however, firewalls prevent access, whereas IDS analyze data *a posteriori*.

Signatures. Most systems detect intrusions based on heuristics based on known attacks. These attacks are summed up to patterns of events (that have to be monitored, either at the system or at the network level). Each pattern is recorded and compared to normal behavior. When the pattern does not trigger any false positive, it is recorded. At run-time, events are scanned (either in real-time or after operation) and an alarm is raised in the IDS when a pattern is recognized. This system has the advantage of labeling the attack (as it is mapped to known attacks).

Statistical anomalies. The other way to detect intrusions is to use statistical laws to detect unusual behavior. This approach is more complicated and a bit tougher to integrate in the reporting systems (as they are not systematically mapped to known attacks), but this is the only process capable of detecting unknown attacks.

Monitoring reporting

Reporting for large scale systems is complicated. Most systems use three forms of reports:

– individual (long) reports on each node or link (usually accessible through a selection, not directly);

– graphs usually produced by the RRD software [19] or some equivalent; this software manages several rates of sampling according to the age of the data, and produces different hourly, daily, monthly, and yearly graphs for data without keeping all intermediate data;

– simple state (usually color-coded) display for nodes or groups of nodes (clusters). The green/yellow/red range is conventional for such systems. Expanded reports are where the detailed failure information is stored.

Almost all systems allow the alarms (usually in the simple state only, possibly also in the expanded reports) to be silenced (temporarily or definitely).

11.4.2. *Failure detection*

One key issue for ensuring resource availability is detecting resource failures. The rest of the system (or part of it) needs to be notified that a failure has occurred in order to take the appropriate measures. For example, a recovery protocol may be triggered the moment the failure is detected. Surviving nodes may simply stop sending messages to the dead node.

Consistent knowledge of the members of a set of processes (called a *group*) is called *Group Membership Service* (GMS). When a failure occurs and disconnects a process from the distributed system, this process *leaves* the set of processes forming the distributed system. When a process joins the computation (e.g.after a recovery protocol is used), it *joins* the distributed system. Hence, failure detection under the fail-stop model can use a group membership service. The GMS has been proved impossible in an asynchronous system with failures in [CHA 96]. The asynchronous hypothesis makes it impossible to determine whether a process is slow or if it is really dead. Moreover, the requirement for a consistent view between the surviving processes of the system reaches the same impossibility result under these hypothesis as the consensus problem [FIS 85].

Heartbeat [AGU 97] is a failure detector that is *unreliable* and *eventually perfect*. For the aforementioned reasons, it cannot be reliable at a given moment. However, if a process dies, it is eventually detected and, conversely, if a process is alive there is a time after which it is not suspected to have crashed. The *heartbeat* mechanism refers to the fact that each process periodically sends a message to the other processes. Each process maintains a vector of counters corresponding to the heartbeats it has received from the other processes. Each process maintains a local list of processes

19. Round-Robin Database Tool, see http://oss.oetiker.ch/rrdtool/.

that are suspected to have crashed. Some processes may be wrongfully put in this list, e.g. if their heartbeat has been received late. This is why the failure detector is unreliable. If a process notices later that a process it has put in this list is actually alive, it removes it from the list. It must be noticed that this failure detector does not use any time-out, nor does it make any assumption on relative process speeds or time-limited communications.

Finally, the *Simple Network Management Protocol* (SNMP) [CAS 90] can be used to detect hardware and software failures on nodes and network disconnections. SNMP-FD [WIE 06] uses SNMP messages between network equipments and hosts to detect failures (using heartbeats and time-out, hardware link down traps and OS notifications) and notify other equipments when a failure occurs.

11.4.3. *Fault tolerance in distributed systems*

System robustness

Defining robustness is not an easy task and many contributions come with their own interpretation of what robustness is. Actually, there exists a systematic framework that permits the characteristics of a robust system to be unambiguously defined. In fact, this should be probably applied to any system or approach claiming to propose fault-tolerance mechanism. This framework, formalized in [ALI 04], answers the following three questions:

1) What behavior of the system makes it robust?

2) What uncertainties is the system robust against?

3) Quantitatively, exactly how robust is the system?

The first question is generally linked to the technique or the algorithm applied. The idea is to explain the general approach used. The second question explicitly lists the type of faults or disturbing elements targeted by the system. Unfortunately, too many studies in the literature miss this part so that there is always a case that breaks the proposed approach. Consequently, answering this question is critical to delimit the application range of the designed system and to avoid counter examples selected in a context not addressed by the robust mechanism. The third and last question is probably the most difficult to answer, and at the same time the most vital to characterize the limits of the system. Indeed, there is nearly always a threshold on the error/fault rate above which the proposed infrastructure fails to remain robust and breaks (in some sense). Answering this last question enables the following to be formally detailed:

– the *robustness metric* used to quantify the resilience of the system in a given context at a given time;

– a set of *bounds* related to the robustness metric above which the system fails i.e. stops to be considered robust.

Fault classification

Faults in a system can be classified according to several criteria:

– whether the fault is *temporary* or *permanent*;

– whether the fault is *intentional* or not;

– the *impact* of the fault on the system.

[AVI 04] makes a distinction between failures, errors and faults:

– a *failure* is when the service provided by the system is in deviation from the expected service;

– an *error* is when the behavior of a system is in deviation from the expected behavior;

– a *fault* is the cause of an error.

Permanent failures cause the system to stop working normally, and the system will not start providing a normal service again without any intervention. A system hit by a *temporary failure* will eventually resume its normal service. A typical example of transient faults is a temporary, external phenomenon that prevents some messages from being delivered on a network (e.g.radio interference on a wireless network). This fault causes a temporary failure, in a sense that the system (i.e. the network) does not deliver the expected service (i.e. does not transmit the messages) during a finite period of time. When the cause of the failure is over (i.e. the radio waves are no longer disturbed), the system can deliver its normal service again.

Failures and errors can be classified into the following categories based on their semantics:

– *crash-stop failures*, also called fail-stop: the system stops working and does not execute any operation, nor does it send any signals;

– *omission* is a communication failure: typically, a message is not transmitted by a communication channel, or not sent by the sending process (send-omission), or not received by the receiving process (receive-omission);

– *duplication* is the opposite from omission: a message is sent or received twice;

– *timing* if the system's behavior deviation concerns only a time criterion (reaction time to a given event for instance);

– *byzantine errors* are arbitrary errors: the system arbitrarily does not have the expected behavior or has an erroneous one;

Detection of fail-stop failures is described in section 11.4.2. Omission and duplication are often detected by the underlying communication protocol (e.g. packet numbering). Byzantine errors are the hardest to detect, as the behavior of the system is often similar to the expected one. A typical example of byzantine failure comes from volunteer computing where it is sometimes referred to as *cheating faults*. Indeed, the motivation for the users to contribute to desktop grids are manifold, including the altruistic desire to help or, more importantly, the assignment of credit points proportionally to a user's contribution. These points reward hard-working clients in different ways [KWO 07]. Unfortunately, incentives also attract cheaters who seek to obtain these rewards with little or no contribution to the system. Such selfish behavior can be achieved by modifying the client software as experienced in Seti@Home [20] [MOL 00]. Whether the consequences of malicious acts or not, this example illustrates a scenario where the results of a computation conducted by a remote client are corrupted by cheaters (hence the reference to cheating faults). What makes this kind of fault byzantine and difficult to catch is that the data sent to the server, although erroneous, generally respect the expected format and do not raise alarm regarding the integrity of the data itself.

The presence of cheaters in grid computing systems is well-known and many countermeasures have been proposed in literature. They will be presented in section 11.6.

Failure risks in distributed systems

Generally, distributed systems are subjected to many different threats. The most common are listed below:

– *scans* corresponding to search phases in order to list available services with a maximum of details (running daemon version, machine bandwidth etc.). Unix tools such as `nmap` [NMA] can be used in this context;

– *denial of services* (*DoS*), eventually distributed;

– system intrusion/extrusion, typically exploiting *software vulnerabilities* and/or *malwares* so as to execute malicious code.

All of these threats naturally increase failure risks. Yet we would like to highlight here that even by considering *only* the aging of the hardware, the probability of failure in a distributed system quickly tends to 1 with an increasing number of processors [TRI 01]. More precisely, let us consider a distributed system composed by n processors. By definition, the *reliability* $R(t)$ of this system is the probability that no failure occurs i.e. that the system remains functional during the interval $[0, t]$. In [DEV 98, SIE 91], it was shown that $R(t) = e^{-\lambda t n}$, where λ is a constant representing the failure rate of a processing element. In practice, λ corresponds to the inverse of the mean time between failures (MTBF) of the system: $\lambda = \frac{1}{MTBF}$. It

20. Some volunteers modified the client executed on local machines to "better" compute the fast Fourier transform (FFT) in order to send results faster. Yet the modification made to the software had so many glitches that incorrect results were sent back from client using the altered version of the software. This leads to the loss of several months of computations.

follows that the probability that the system fail during the interval $[0, t]$ is [SIE 91]:

$$F(t) = 1 - R(t) = 1 - e^{-\lambda t n} \tag{11.1}$$

Figure 11.7 exhibits the evolution of the failing probability in a distributed system composed by n processors. It illustrates the tendency to quickly converge to 1 as the number of processors (and the execution time) increases. It should be seen as a minimal failing probability as, again, a distributed system is subjected to many more threats than the regular hardware aging.

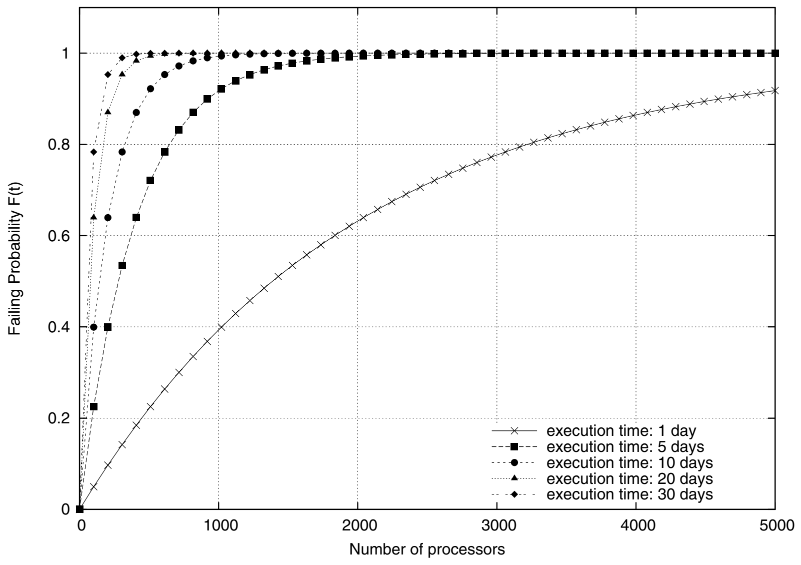


Figure 11.7. *Evolution of the failing probability in a distributed system composed by n processors. Only the aging criterion is considered with processors MTBF estimated to $MTBF = 2000$ days $\sim$ 6 years leading to $\lambda = 0.0005$*

11.4.4. *Rollback recovery*

The state of a process can be saved during its execution and restored later. For example, the process can be migrated on another machine and resume its execution. *Checkpointing* consists of saving the state of a process by writing a snapshot of its current state on a disk, potentially to restart this process from this state. It can be used for fault-tolerance purposes: processes are checkpointed on a regular basis, and rollback to a previously stored state after a failure has occurred. Rollback recovery can be used to restore the state of a single process. Distributed systems, on the other hand, are not only made of a set of processes, but also include communication channels.

Hence, interactions between processes need to be taken into account when saving and recovering the state of a process.

When a failure occurs and some processes of the system are killed, the recovery protocol must make it reach a state that *could have been reached* if no failure had occurred. This state is called a *coherent state*.

Rollback recovery protocols are classified in two categories: coordinated and non-coordinated checkpointing protocols. In coordinated protocols, processes are check-pointed in a coordinated fashion. In non-coordinated protocols, they are checkpointed independently from each other.

Coordinated checkpointing

Most coordinated checkpointing protocols are based on the Chandy-Lamport algorithm [CHA 85]. Coordination relies on a wave algorithm: a marker is sent between processes to trigger local checkpoints. When a failure occurs in the system, all the processes are restarted from a checkpoint that is part of the latest complete checkpoint wave. The set of checkpoints taken by all the processes of the application during a checkpoint wave forms a *cut*. Moreover, this cut is such that the set of process states saved by the checkpoints forms a coherent state. Hence, this cut is called a *coherent cut*. If all the processes rollback to the latest set of checkpoints, the application is restored in a coherent state. Each process is checkpointed upon reception of this marker. It forwards the marker to the other processes using the same communication channels as used for regular communications. Assuming that communication channels follow the FIFO property, messages from a given process that are received after the marker coming from this process have been sent after the marker. Messages sent and received before the checkpoint wave are part of the portion of execution that precedes it. Messages sent and received after the checkpoint wave are part of the portion of execution that comes after the checkpoint wave. If the application is restarted from the latest checkpoint wave, these messages will be sent and received again.

Messages that cross the checkpoint wave (i.e. that are sent during the checkpoint wave) need some specific treatment. They can be handled in two ways. The first consists of blocking communications during the checkpoint wave. Each process stops communicating when it enters the checkpoint wave and performs a local checkpoint after it has received markers from all the other processes of the system. It communicates again after the checkpoint wave. As a consequence, no message can cross the checkpoint wave.

The other solution consists of storing the contents of the messages (i.e. logging them) sent during the checkpoint wave. This involves some participation from the communication library and complexifies its critical path, but the checkpoint wave does not have any preempting effect on the execution of the processes, such as with the blocking implementation.

A comparison of a blocking and a non-blocking implementation of the Chandy-Lamport coordinated rollback recovery protocol in a parallel communication library has been studied and published in [BUN 08]. The architecture of this fault-tolerant library will be described further in section 11.4.6.

The comparison between these two approaches shows that for applications made of a limited number of nodes and using a high-speed network (e.g. a cluster), the cost of logging messages during the checkpoint wave has more impact on the performance than the blocking synchronization. This synchronization has a relatively low cost because of the limited size of this kind of system and the low latency of the network. Logging messages, on the other hand, adds a significant overhead to inter-process communications, regarding the low latency of the network. On large-scale systems with a higher latency between processes (e.g. a grid), the global synchronization involved by the blocking implementation extends the duration of checkpoint waves and, as a consequence, the periods of time during which processes are blocked. Henceforth, the non-blocking implementation gets better performance on grids.

Non-coordinated checkpointing

The two main drawbacks of coordinated rollback recovery are the cost of the synchronization required to coordinate the checkpoints, and the fact that *all* the processes must rollback upon failures, even processes that have not been hit by the failure.

Non-coordinated protocols rely on the notion of determinism in the execution of a process. A process is said to be *deterministic* if for a given initial state, it always reaches the same final state. In a distributed application, processes interact with each other and have causal dependencies with one another (i.e. a message sent by a process can have an influence on the state of the receiver). As a consequence, they cannot be considered as deterministic. However, their execution can be split into phases of deterministic execution separated by non-deterministic events. These events are typically interactions with the rest of the world (I/O, inter-process communications). This is called the *piecewise deterministic assumption* (PWD) [STR 85]. A direct consequence of the PWD is that for a given initial state, if all the non-deterministic events of the first execution are replayed in re-executions, a process will always reach the same final state.

Non-coordinated checkpointing and rollback recovery rely on this assumption. Processes are checkpointed independently from each other. Upon a failure, the process that has been hit by the failure rolls back to a previous checkpoint while the others continue their execution. Non-deterministic events that occurred during the first execution are replayed in order to reach a coherent state.

Non-deterministic events are inputs and, more specifically, interactions with other processes and with the rest of the world. For parallel applications that follow the message-passing communication model, these events are messages that are received by the process. When processes communicate with one another, the message must be saved. If the receiver of the message is hit by a failure and has to rollback, this message and all the other messages that have been received by this process between the moment when it has saved its state and the failure will be replayed in the same order. Sending a message is not a non-deterministic event, but messages must be sent exactly once; messages that have been sent during the first execution are not sent again during the re-execution of a process that has been restarted. As a consequence, the system will eventually reach a coherent state with no pending communication.

In practice, messages are stored locally by the sender (*sender-based message-logging* [JOH 87]) or the receiver (*receiver-based message-logging* [BOR 83]) in their volatile memory. The causality information between them is the only piece of information that needs to be stored on a stable storage support (called *event logger*). This information is used to replay the messages in the same order as during the initial execution, and consists of a vector of logic clocks [LAM 78]. The messages that have been stored locally will be included in further checkpoints of each process.

The way this causality information is saved makes the difference between the various message-logging protocols.

Pessimistic message-logging

Pessimistic message-logging waits for the causality information to be acknowledged by the event logger before proceeding with sending the message [JOH 87]. If a failure occurs on the receiver before the causality information has been saved, then the message has not yet been sent so the sender cannot have had any influence on the receiver.

Pessimistic message-logging ensures the fact that the system will always be restored in a coherent state, regardless of when the failure occurs. However, this protocol adds some latency to message communications.

Optimistic message-logging

Optimistic message-logging makes the assumption that no failure will occur between the moment when the causality information is sent to the event logger and the moment when a message is delivered to its receiver [JOH 90]. The sender of a message sends this information to the event logger and, as soon as it has finished sending it, proceeds with sending the message.

If a failure happens after the message is delivered to the receiver and while the causality information is still in the communication channel between the sender and the event logger, this information is lost and the message cannot be replayed. For this reason, optimistic message-logging assumes that failures are rare. It does not increase the latency of the communications as much as pessimistic message-logging does.

Causal message-logging

Causal message-logging does not require waiting for the acknowledgment from the event logger, and cannot lose any information. The sender of a message sends the causality information to the event logger and proceeds to send the message immediately, adding the causality information with message. This information is kept with the message as long as the event logger has not acknowledged safe storage of the causality information [ALV 95]. As a consequence, the causality information is kept in the system until it is stored on a reliable storage support. If a failure occurs before the information has been stored, it can be retrieved from the piggyback of the messages that have been sent since the message has been sent.

Several protocols define how the causality information can be maintained in the piggyback of the messages. For example, Manetho maintains the causality information in an antecedence graph, which is propagated with messages [ELN 92].

Comparison between protocols

Fault-tolerance protocols induce overheads on two parts of the execution: during fault-free execution (i.e. cost of the protocol used to save the state of the system), and upon failures (i.e. cost of rollback recovery protocol).

Coordinated checkpointing protocols add few overheads on failure-free executions, as they do not do anything between two checkpoint waves. Non-coordinated protocols, on the other hand, require to log messages sent between processes, which induces a significant overhead on the communication performance. This overhead varies between message-logging protocols: besides of the cost of saving a local copy of the message, pessimistic message-logging adds the point-to-point latency (by requiring a communication with the event logger and an acknowledgement) and causal message-logging reduces the bandwith available to messages. Optimistic message-logging does not involve any overhead aside from the cost of the local copy, but it does not guarantee that the system stays in a coherent state.

Upon failure, coordinated protocols require *all* the processes of the system to rollback on the latest checkpoint wave. Computation made by processes since their latest checkpoint is lost. Moreover, if a failure occurs before the next checkpoint wave, the execution cannot progress beyond failures. As a consequence, coordinated checkpointing protocols are not suited to highly volatile environments. Non-coordinated checkpointing protocols, on the other hand, do not require all the processes to roll back upon failures: only processes that have been hit by the failure roll back to their latest checkpoint. However, if the parallel application requires some synchronization between a process that has rolled back and a process that has not, the latter has to wait for the former.

In a given environment, the choice for a fault-tolerance protocol is a trade-off between overheads on fault-free executions and the cost of rollback recovery when failures occur.

11.4.5. *Application-driven fault tolerance*

In section 11.4.4, we described *transparent* rollback recovery mechanisms. Another way to achieve fault tolerance in parallel applications consists of letting the application handle failures itself. The application knows which processes have failed and adapts itself to the new set of available resources. The responsibility to recover the data that have been lost by the failure and adapt the algorithm (e.g. re-calculate the load balancing) is left to the application. This approach is called *algorithm-based fault-tolerance* (ABFT) [CHE 05].

For that purpose, the application can have two kinds of behavior. The first consists of continuing the computation without the failed processes. For example, if the application was executed initially by n processes, and m processes have been hit by failures, the rest of the computation is made by $n - m$ processes. Another solution consists of spawning new processes that replace the failed processes, and proceed with the execution on the same number of processes.

ABFT require that the application must be supported by an appropriate middleware that can keep supporting the application in spite of failures (and therefore, have some self-healing properties to maintain a correct computational environment) and provides some features to implement fault-tolerance in the application. The MPI middleware FT-MPI [FAG 04] provides these options to notify the application that one or several processes have failed and features an interface to implement the actions that must be taken upon failures by the application.

FT-MPI provides four policies to organize processes beyond failures:

– *shrink*: the failed processes are not replaced and the naming of the surviving processes is modified so that the names form a continuous set of numbers;

– *blank*: the failed processes are not replaced and the naming of the surviving processes is not modified. As a consequence, the numbering of the processes is not continuous. Messages sent to the failed process are not sent and the communication routines return as if it had been done with no error;

– *rebuild*: new processes are spawned to replace the failed processes given the same names as the failed ones. As a consequence, the application can recover its state with the same number of processes. A process that has been respawned knows that it is a clone of a former process;

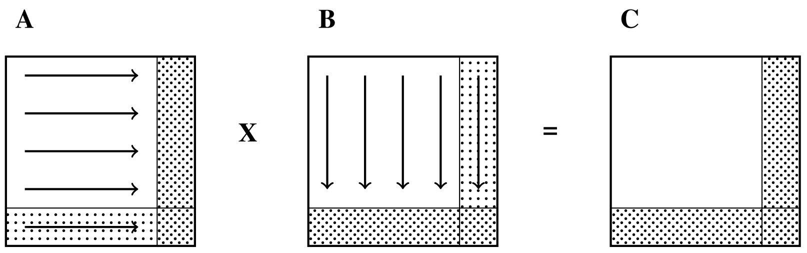


Figure 11.8. *Redundancy for ABFT in iterative matrix-matrix multiplication*

– *abort*: the application is terminated.

Diskless checkpointing [PLA 97] is a possible technique for the application to restore the state of the failed processes. It consists of saving the data that are necessary to restore the state of a given process in the memory of other processes. For example, in many iterative computations, the state of a process can be restored from the entries of a matrix at the end of the previous iteration completed by this process. Processes can send the entries of their matrix at the end of each iteration to a set of distinct processes; the number of processes determines the number of failures that the application can tolerate. When a failure occurs, the state of the application can be restored by spawning a new process to replace the failed one and sending it the corresponding matrix so that it can restore its state.

The FT-LA package [BOS 09] implements a set of fault-tolerant linear algebra kernels using algorithm-based fault-tolerance. Redundant data are stored in additional processes and used to recover upon failures. If processes used for the computation are organized as an $m \times n$ grid, an extra column and an extra line of processes are used to store this redundant data. A checksum is calculated over a set of processes (typically, a line or a column) and stored in one of these additional processes. This approach guarantees the recovery of the data of one process per set of processes (i.e. per line or per column) and detection of flip-bit errors (e.g.communication errors).

Figure 11.8 depicts how fault-tolerant matrix-matrix multiplication can be performed using ABFT. If a matrix A is multiplied by a matrix B, A and B are *pre-conditioned* by calculating the checksum of each line and each column of processes (represented by dotted areas on each matrix). This pre-conditioning introduces the aforementioned redundancy and expands the matrices from over the $m \times n$ grid that contains the initial matrices A and B to over the full set of processes $(m+1) \times (n+1)$. Then the product AB is calculated normally, including the additional processes in the multiplication. The data contained in the additional columns of A and the additional lines of B is not used in the computation. The result of the multiplication gives $C = AB$ with additional lines and additional columns that contain the checksum of each line and each column of processes. If a failure occurs during the computation, the data that should have been contained in the failed process can be retrieved by using the data contained in the processes of its line and the checksum of his line, or doing the same thing in the failed process's column.

Practical experiments and performance measurements on this matrix-matrix multiplication algorithm showed not only that the fault-tolerant approach has a small overhead on failure-free executions, but also that low-performance degradation occurs when failures occur: with one failure, the fault-tolerant matrix-matrix multiplication had 12% overhead with respect to the fastest, non-fault-tolerant implementation.

Whereas the above ABFT approaches apply in the context of crash faults, some recent work has investigated such techniques in the framework of the cheating faults introduced in the section 11.4.3. More precisely, the authors of [VAR 11] show that evolutionary algorithms (EAs), including their distributed implementation, are inherently resilient to a limited number of falsified results produced by cheaters on global computing platforms such as BOINC. This is the first formal analysis of the impact of cheating faults in this context, together with a theoretical proof of convergence towards valid solutions despite the presence of malicious acts. By the variety of problems addressed by EAs, this study will hopefully promote their usage in the future developments around distributed computing platforms.

11.4.6. *Case study*

This section presents implementation of fault-tolerant parallel environments. These three systems implement different strategies to proceed with the execution in spite of failures in the system. We describe here how these approaches are integrated in the parallel model followed by these environments.

MPICH-V

MPI [FOR 94, GEI 96] is the *de facto* standard for programming parallel applications on distributed memory. It defines a standard interface for inter-process communications (point-to-point and collective communications) with the characteristics of portability, flexibility, and performance. However, neither the first [FOR 94] nor the second [GEI 96] release of the standard addresses the issue of recovering from failures. The default behavior consists of terminating the application as soon as a failure has been detected. The MPICH-V framework was designed to permit implementations and experimental evaluations of transparent fault-tolerance protocols for parallel applications written using MPI. Fault tolerance is handled by the MPI middleware: the fault tolerance mechanism is implemented by the communication library and the run-time environment. Hence, the parallel application does not have to be modified nor to support any fault-tolerance feature.

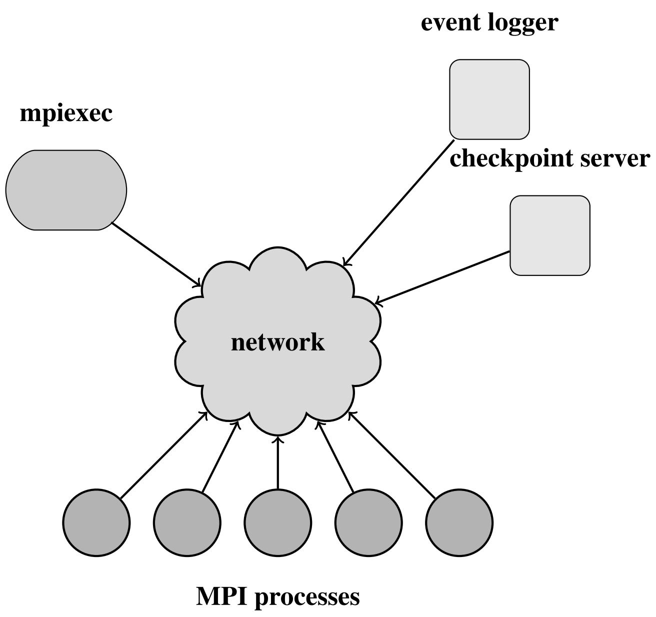


Figure 11.9. *Architecture of the MPICH-V framework*

MPICH-V[21] is based on MPICH [GRO 96] (and MPICH2 in the latest release), one major open-source MPI implementation. It extends the run-time environment of MPICH with additional components that are necessary to support the fault-tolerance protocols. The architecture of MPICH-V is depicted in Figure 11.9. The MPI application is started by the *mpiexec* process, which is a sort of orchestrator of the application and the fault-tolerance protocol. It deploys the *MPI processes*. Rollback recovery protocols require the checkpoints to be stored on a reliable storage support: one or several *checkpoint servers* are added to the run-time environment. Some protocols require information on the messages that are sent between processes to be logged. An *event logger* is used to store these data on a reliable storage device.

MPICH-V1 [BOS 02] implements a non-coordinated checkpointing protocol that logs all the inter-process communications on a reliable storage device called *channel memory*. Each message is sent to the channel memory that saves it and then forwards it to the receiver. MPICH-V2 [BOU 03] implements a pessimistic, sender-based message-logging, non-coordinated checkpointing protocol in order to evaluate the performance of this protocol. MPICH-Vcausal [BOU 05] implements a causal

21. http://mpich-v.lri.fr.

message-logging protocol and several methods to reduce the size of the information piggybacked on messages. Two versions of Chandy and Lamport's coordinated rollback recovery protocol have been implemented. A non-blocking version, which logs all the messages sent during a checkpoint wave, has been implemented in MPICH-*Vcl* [LEM 04]. A blocking implementation is featured in MPICH-*Pcl* [BUN 08]. These protocols follow a simplified version of the architecture depicted on Figure 11.9, and do not require any event logger. These implementations allow experimental comparisons between protocols to be made. These protocols must be compared in two situations: failure-free execution, and in the presence of failures [BOU 06]. Performance evaluation on fault-free executions gives an idea of the overhead induced by the protocol (cost of fault-tolerance). Evaluating the performance with respect to the number of failures gives an idea of the cost of each failure with a given failure-recovery protocol.

Practical comparisons showed that message-logging protocols have a more significant overhead on fault-free executions than coordinated checkpoints, especially on high-speed networks. Large-scale, high-latency systems such as institutional grids must avoid synchronization (such as that involving a blocking implementation of coordinated checkpointing). Smaller systems interconnected by low-latency networks, on the other hand, can be synchronized more quickly and benefit from a simpler protocol that does not log any messages, even those sent during the checkpoint wave of a non-blocking coordinated checkpointing protocol. When failures occur, coordinated checkpointing protocols require all the protocols to rollback, losing all the computation made by the other processes since the latest checkpoint wave. As expected, non-coordinated rollback recovery protocols have better performance when failures occur, except for very low fault frequencies where coordinated checkpointing performs better.

KAAPI

KAAPI [GAU 07] is a scheduling system that executes tasks in distributed systems based on their dependency graph. It uses a dynamic, greedy scheduling algorithm based on work-stealing. A KAAPI application can be reconfigured dynamically at run-time using modifications on the graph of tasks, as long as these modifications respect the semantics of the original program. At a given moment, the state of a program is represented by the current dataflow graph. As a consequence, the state of the application can be recovered from a snapshot of each process and the local state of the dataflow [JAF 05b]. In the context of KAAPI, the snapshot of the local process and the local state of the dataflow will be referred to as the local *checkpoint* of a task.

The main idea of the work-stealing algorithm is that idle tasks steal work from other processes. As a consequence, stealing work from one another is the only possible dependency between two tasks. KAAPI implements a *theft-induced checkpointing* protocol [JAF 05b], which is an adaptation of the *communication-induced checkpointing* protocol [HEL 99] in the context of the KAAPI model. Communication-inducted checkpointing protocols are based on two kinds of checkpoints. Tasks or processes take checkpoints on a regular basis to make sure that the recovery line is progressing: these checkpoints are called *local checkpoints*. The other kind of checkpoints is triggered by communications (i.e. causal dependencies between tasks or processes): these checkpoints are called *forced checkpoints*. With KAAPI, forced checkpoints are triggered by work stealing between tasks.

Distributed applications in KAAPI are assumed to respect the piecewise deterministic assumption (defined in section 11.4.4). We have seen that in the context of KAAPI, the only causal dependencies between tasks are tied to the work-stealing mechanism. As a consequence, non-deterministic events in a KAAPI application are those that require a modification of the dataflow graph.

KAAPI implements a *systematic event logging* protocol [JAF 05a] that logs all the modifications that are made at run-time on the dataflow graph (e.g.creations of new nodes). This protocol is an adaptation of the message-logging protocols used in the message-passing model to the context of KAAPI. Since the state of the application is represented by the data flow graph at a given moment, a global, distributed checkpoint of the system can be taken by saving the state of all the processes of the tasks of a KAAPI application and the state of the graph. KAAPI uses the knowledge it has on future communications provided by the dependency graph of the application to implement the coordination mechanism that is necessary to take a global checkpoint of the application. Coordination is made between processes that will communicate with one another in the future of the execution of the application, rather than with all the processes of the application. This optimization reduces the number of messages sent during the coordination phase while keeping the state of the application consistent [BES 08].

Charm++

Charm++ [Kal 93] is a parallel programming language based on C++. A parallel Charm++ application consists of two kinds of objects: concurrent objects (called *chares*), scheduled by the Charm+ run-time environment, and messages sent between chares. Messages are used to invoke methods on a remote chare. Charm++ takes advantage from C++ features, such as data marshalling, to implement its run-time system efficiently and in a portable way.

Several fault-tolerance strategies are available with Charm++. One particularity of these protocols is that they use *in-memory* checkpointing to speed up the checkpointing process. Rather than transferring the checkpoint on a remote server, processes save them in other processes' memory. The checkpointing time is therefore reduced, since the data transfer can take advantage of the low latency of the local network.

In-memory checkpoints are not as reliable as stable data storage systems. To make this protocol more reliable, *FTC-Charm++* [ZHE 04] is a coordinated rollback recovery protocol that uses *double checkpointing*: checkpoints are replicated on two different processes executed on two different processors, called its *buddy processors*. Double checkpointing is more robust than storing only one copy of a checkpoint on a single process, as long as the three processes that keep a process's state (the process itself and the processes executed on its two buddy processors) are not killed. In-memory checkpointing involves a memory overhead that can be high if the application has a large memory footprint. Another variation of in-memory checkpoints is called *in-disk* checkpointing: checkpoints are stored by processes on a local disk rather than in their volatile memory.

FTL-Charm++ [CHA 04] is a pessimistic sender-based message-logging protocol. Before sending a message, a process coordinates with the receiver to emit a *ticket*, which will be used to replay the sequence of messages in the same order in a similar way to determinants described in section 11.4.4 for non-coordinated protocols. Unlike traditional message-logging protocols, FTL-Charm++ does not save this ticket on an event logger supposed to be located on a stable storage support, but on a *buddy processor*, similarly to the buddy processors used for double checkpointing. The recovery protocols take advantage of the dynamic scheduling and process migration features of Charm++. Since the run-time environment of Charm++ schedules the chares of an application on the available resources dynamically at run-time and, in general, schedules several chares on each physical processor (oversubscription), there is no absolute need for spare resources to restart the failed processes. A fast restart protocol which takes advantage of the process migration and dynamic scheduling features of Charm++'s run-time system, reduces the time required to re-execute a process after it has rolled back to the previous checkpoint in the non-coordinated protocol [CHA 07]. Process migration is also used for another fault-tolerance strategy. Assuming that some failures are predictable (e.g. an abnormal increase of the CPU temperature), processes that are very likely to be hit by a failure can be migrated before the failure actually occurs. This *proactive approach* [CHA 06] takes advantage of alarms raised by hardware signals and fault prediction schemes to *evacuate* processes before the failure actually occurs. As a consequence, the processes do not have to rollback *at all*.

11.5. Ensuring resource security

The security, and especially the integrity, of a resource is one of the user's primary concerns in the distributed system. Indeed the user needs to be sure that the computing resources he is using are not modified, i.e. that no malicious software or backdoors are installed.

11.5.1. *System enforcing*

The problem of how to secure a computing resource has been studied since the early foundations of computer science. It has increased with the development of the networks and especially with the expansion of the Internet. As already mentioned in section 11.4.3, connected systems are subjected to many threats and potential attacks such as DoS, intrusions or theft of information are conducted every day on connected systems. The inherent security of each machine strongly depends on the type of operating system (OS) running on the computer. Nowadays, Unix-based systems are considered the most secure, assuming of course they are correctly configured. We list here the general approaches (not only linked to these systems) used in system enforcing. This applies to every computing component including our commodity machines and it should be seen as advice for protecting computers.

Taking care of the security of each individual computer is one of the challenges for cluster/grid system administrators as a secure distributed system assumes a reasonable security level for each node composing the system. In this context, the following elements should be taken into account:

– *activating the system firewall*. Of course, network flow can be controlled at the backbone level (via the creation of dedicated VLANs with specific access rules) yet it does not prevent a connected node from protecting itself against undesirable network traffic. The network services provided by servers are bound to port numbers. These are like doors the user is connecting to, in order to receive or send information. Without any specific protection, a program running on a computer system can open any port on the machine and let anybody access this system. A way to block any unwanted services to open ports is to block them using a *firewall*. More generally, firewalls control the network flow originating (or targeting) a machine. This is useful to block any port numbers known to be used by common malwares. Figure 11.10 illustrates a firewall activated on a cluster access front-end that typically runs a web server and a SSH server to permit connection from the outside. As all web requests are handled on ports 80 and 443, and the SSH connection is bound by default to port 22, there is no need to open other ports on the machine;

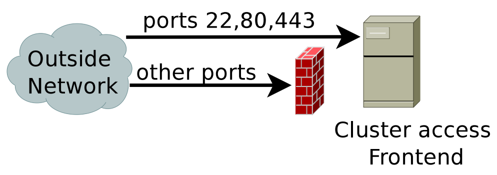


Figure 11.10. *Illustration of a firewall activated on a cluster access front-end*

– *ensuring an up-to-date system*. All OSs come with a way to update the system. It should be done on a regular basis to ensure that discovered vulnerabilities are fixed as soon as a patch is available for the system;

– *restricting the available services and place quotas*. On too many systems, the default installation integrates more services than required, and each of them is a potential source of attacks and vulnerability. Among the exceptions to this behavior, we can cite the Debian OS, which explains why it is one of the favorite Linux distributions for system administrators. In all cases, the running services should be restricted to the minimum. For instance, a cluster access front-end will not need more than a SSH server (for the connection) and eventually a web server (the front-end generally has a public IP, it is available from the Internet so it is probably a good host for a website displaying cluster information). Additionally, it is possible to limit resource usage via quota configuration;

– *use sandbox environments to confine program execution*. This is often used to execute untested code, or untrusted programs from unverified third-parties, suppliers, and untrusted users. Classic examples of sandboxes include virtual machines, kernel jails on FreeBSD systems[22], and cgroups[23], `chroot` environment;

– *monitor resource usage*. The activity of each computing resource should be monitored. Several tools can be used at this level, among which we can mention:

- Ganglia[24], a scalable distributed monitoring system for high-performance computing systems such as clusters and grids,

- Nagios[25], a powerful monitoring system of a complete IT infrastructure, such as a cluster, to ensure systems, applications, and services are functioning properly. In the event of a failure, the technical staff are alerted of the problem, allowing them to begin remediation processes,

- Tripwire[26] is an integrity tool for checking any modifications on the files of the disk. This is done by logging a fingerprint of the binaries and configuration files of the system like their hash-values, and checking afterwards, periodically or on demand, that these fingerprints have not changed, i.e. that the files have not been tampered with. It is useful for detecting rootkits or malwares,

- Logcheck[27], a simple utility that is designed to allow a system administrator to view the logfiles that are produced upon hosts under their control. This is done by mailing summaries of the logfiles to them, after first filtering out "normal" entries.

In addition to these tools, Intrusion Detection Systems (IDS) can be used.

22. http://www.freebsd.org/doc/en/books/arch-handbook/jail.html.
23. http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.
24. http://ganglia.sourceforge.net.
25. http://www.nagios.org.
26. http://sourceforge.net/projects/tripwire.
27. http://logcheck.org/.

11.5.2. *Trusted computing*

The concept of *trusted computing* was introduced in the computer security literature a long time ago and the initial ideas are now incorporated in PCs, mobile phones as well as more dedicated material: servers, disk, and storage infrastructures.

Two recent papers [MAR 08, CHE 09] synthesize the issues of trusted computing: "the re-design of systems architecture in such a way as to support its factorization into relatively discrete components with well-defined characteristics". The definition is about software security in a broad sense and it impacts privacy, digital rights management, and certification. It impacts also on the way we communicate in a trusted network and the way we store data with a trusted storage component.

The *Trusted Computing Group*[28] is an industry consortium promoting the development of Trusted Computing. For instance, it promotes the use of a special hardware, namely the Trusted Platform Module (TPM) and it proposes standards for it.

11.5.3. *TPM (Trusted Platform Modules): Hardware for Trusted Computing*

The naive approach to software integrity is to compare the running software to well-known hash sums. However, this method of software integrity does not resist simple attacks by simulating the target hardware and software in a virtual machine, thereby exposing the software and the processed data to analysis in specific environments.

The media industry have been the main proponents of a solution that could push the encryption to the hardware level and integrity checking to the point where no software could spy upon the target code. Password protection is touted as the main advantage of the proponents of such a hardware protection, but management of licenses (to media content or to software) is clearly the target of such hardware.

The largest and by far most successful hardware elements that are capable of checking the state of the software and hardware stack it is running on (without being fooled) are those responding to the TPM specification [ISO09a].[29]

Other implementations of this concept do exist: for example, the CryptoPage architecture [DUC 06] uses strong cryptography in the processor between the internal cache and the external bus with a cache line cipher (to shuffle the memory addresses in an opaque manner) and a memory verifier based on Merkle tree hash function.

28. http://www.trustedcomputinggroup.org/.

29. The hardware piece implementing the specification is also called the TPM chip.

The goal is to obtain software processes able to resist various attacks, such as replay attacks.

However, TPM chips is the only implementation that received strong support from the industrial community and can now be found in many computers (especially laptops, but also desktop computers and cell phones). Microcontrollers implementing the specification are produced by Intel, Broadcom, and STMicroelectronics, among others.

The heart of the TPM chip comprises three parts: a cryptographic processor, the permanent memory, and the normal memory (plus an I/O bus connected to the rest of the board). The cryptographic processor is a common architecture capable of performing encryption, decryption, signature checking, 2048 bits RSA-key generation, SHA1 hashing, and of course, a good RNG.[30] The normal memory contains various keys used for the normal co-processor operation, such as storage keys, etc. It is partly non-volatile. The permanent memory contains essentially two keys: a 2048 bit RSA key pair (PUBEK and PRIVEK). The private part is built into the chip at the factory and should never be revealed.

With sufficient care (and this is what the TPM specification amounts to), the TPM chip is able to keep PRIVEK private and thus issue unfalsifiable data signatures (which are gathered from the main CPU). This is the key point to authenticating the controlled data. The virtual machine scenario is of course eliminated as the virtual machine cannot use the TPM chip as if it were native.

The use of the key also allows platform authentication to be performed (the system with the TPM can digitally sign a message that can only come from itself, and can thus prove the identity of the system to the authentication system). Other uses are also full-disk encryption.

This TPM module is not compatible with the cloud computing approach, however, and is subject to either cryptanalysis or physical attacks. Cryptanalysis allows PRIVEK (from PUBEK or from messages) to be retrieved; physical attacks allow in-memory data to be found, so that data protected by TPM may still be retrieved, even if only authorized software is executed at the time of the decoding of the data. For example, a physical access to the memory modules after the system has been stopped brutally while operating leaves a window of several hours during which data can still be read when put inside another board.

30. A random number generator is necessary to create key pairs.

For Linux users, we can also depend on different software, among them Trousers and the `tpm-tools`[31] and also the TPM emulator[32], which is a very active project. The objective of this project is the implementation of a software-based TPM/MTM[33] emulator as well as of an appropriate TCG Device Driver Library (TDDL). The TPM emulator package comprises three main parts: a user-space daemon (tpmd), which implements the actual TPM emulator, a TPM device driver library (tddl) as the regular interface to access the emulator, and a kernel module (tpmd_dev), which provides the character device `/dev/tpm` for low-level compatibility with TPM device drivers.

Finally, we must recognize that this kind of technology is not yet integrated in software projects, notably those in grid computing. In February 2010, Christopher Tarnovsky announced that he had been able to hack into an Infineon TPM chip using an electron microscope[34]. TPMs were never intended to be invulnerable to an attacker with physical access to the machine hosting the chip.

11.6. Result checking in distributed computations

This section presents methods for *a posteriori* verification of the integrity of the system by checking the correctness of the results produced by this system.

11.6.1. *Overview*

The integrity of executions is handled at two levels. First, at the scale of the execution itself by the guaranteeing that it is correct despite the dynamicity of the computing resources. This introduces the setting of appropriate fault tolerance mechanisms, such as those presented in this chapter (see section 11.4 typically).

Such mechanisms are not sufficient to ensure integrity, as resources and results produced by the different tasks may be corrupted. Concerning the protection against the corruption of computing resources, several approaches involving both hardware and software mechanisms can be considered, as described in section 11.5.

This section is dedicated to the protection against the second aspect, i.e. results' corruption. We have already introduced this type of fault in section 11.4.3, when classification of the different types of faults was introduced. Result falsifications are often referred to as *cheating faults*. As mentioned in section 11.4.3, such byzantine errors are quite hard to detect in the sense that the data produced by the execution of

31. http://trousers.sourceforge.net/.
32. http://tpm-emulator.berlios.de/.
33. MTM: Mobile Trusted Module.
34. http://abcnews.go.com/Technology/wireStoryid=9780148.

the program, although erroneous, generally respects the expected format and does not raise the alarm on the integrity of the data itself. Again, an eloquent example already mentioned is the famous SETI@home problem: some collaborators have substituted the initial FFT computation with their own code, which is inadequate [MOL 00]. We do not really know if it was a simple blunder or a true malicious attack – the goal was to improve the collaborator's rank in the SETI *hall of fame*. Nevertheless, the forgery of results has caused the loss of several months of computation.

More generally, result forgery i.e. cheating comes from incentives associated with the execution. Some recent analysis has demonstrated that this behavior is not marginal. In [KON 07], it was shown that a remarkably high percentage of hosts – about 35% – in BOINC that were monitored via the XtremLab project [XTR 09] returned at least one corrupt result in the 3 month time frame over which the experiment was conducted. The current trend is an increase of this behavior, mainly because of the enthusiasm around the cloud paradigm. Indeed, incentives at this level have become clearly money-oriented, which is an additional motivation for cheaters and sabotages to intervene.

The presence of cheaters in distributed systems is well-known and many counter measures have been proposed in literature. A major reference in this domain is the work of Sarmenta [SAR 02]. Two complementary strategies can be put forward:

1) prevention of *a priori* forgery by making it harder to perpetrate. The question is then to strengthen the architecture of global computing as expounded in section 11.5, e.g.through quotas (on disks and/or CPU), firewalls, or confinement of execution or appropriate rights on resources;

2) *a posteriori* control of the output generated by the tasks by adopting techniques of quantitative evaluation of the accuracy of results in such a way that corrections may be applied on-the-fly.

We now focus exclusively on this second point and describe the different approaches followed during the result certification step and integrity control mechanisms used in distributed systems.

11.6.2. *Result-checking by post-condition*

A result certification mechanism is composed of two aspects:

1) the program's formal verification [BOY 82] before its execution. It consists of verifying a set of properties on a downloaded application through a formal proof that comes with the application. The conclusions of the proof are then compared with the security policy of the target machine in charge of the execution in order to determine whether running the application is safe or not. We can also use a specific assembler language such as [MOR 99]. This approach remains specific as the existence of a

validity proof is not guaranteed: it depends on the program and is also costly. In addition, it does not improve confidence in the results as they are returned by a potentially corrupted resource;

2) the program testing [BLU 89] or property testing [GOL 98]. In the first case, we evaluate the outputs of the program on predetermined inputs for which we already know the result. In the second case, we check whether the generated outputs are close to a given property established for the program.

If the program testing method can easily be circumvented, the approach by testing a property falls within the general mechanism of *simple checkers* introduced in [WAS 97]. This involves a post-condition checking on the results generated when the cost of this checking is lower than the computation itself. The idea is that for some problems, the time required to carry out the computation is asymptotically greater than the time required to determine whether or not a given result is correct. This is possible thanks to a post-condition the output have to conform to. For instance, let us consider the discrete logarithm problem (DLP): given a group $(G, .)$, a generator g of this group and an element $h \in G$, output x such that $h = g^x$. Checking a given output x is far more simple (using fast exponentiation typically) than computing this output.

The post-condition approach remains the most efficient one for a problem **P** given at the beginning. Yet it remains specific: it is often impossible to automatically extract such a post-condition on a program. Additionally, it does not isolate all the resources responsible for the forgery of the result.

11.6.3. *Result-checking by duplication*

Another method uses duplicated computations to evaluate the correctness of the results produced by computing resources. Generic approaches are based on duplication.

Task replication for batch certification and reinforcement.

At this level, most previous studies [SAR 02, GER 05, KON 07] consider programs with multiple parameters that summarize to a set of *batches* scheduled by a reliable server which can be distributed. A batch is composed of n independent tasks $\{T_i\}_{1 \leq i \leq n}$ each task T_i receiving the inputs $\overrightarrow{e_i}$ and generating the output $\overrightarrow{s_i}$. The tasks are executed in parallel on resources of a global computing system: a set of *workers*. The results of the executions are returned to the scheduler.

In the remainder of this subsection we assume that the workers run at the same speed. When the scheduler has collected all the results $\{\overrightarrow{s_i}\}_{1 \leq i \leq n}$, it validates them, and then generates the next batch. This procedure is repeated until completion of the whole computation. Moreover, due to the execution in an hostile environment, result corruption may occur. In general, it is assumed that the program supports a proportion, denoted δ, of falsified results for each batch. For some applications, this threshold can be relatively high (1% or more). This applies, for example, to applications of image rendering and videos where the presence of a few erroneous pixels remains unseen by the human eye.

Furthermore, result falsification have been often modeled by assuming a fraction of ω saboters behaving as Bernoulli processes with an unknown constant probability s. Consequently, falsified results in a batch of size n is a random variable T_n which conforms to a binomial law $\mathcal{B}(n,p)$ with $p = \omega s$. Using those hypothesis, two approaches are proposed in the literature:

– Germain *et al.* [GER 05] tried to certify the quality of the batch as soon as possible through a test $\mathcal{T}$ and probabilistic checks;

– Sarmenta [SAR 02] enforced the batch quality by making the falsification probability lower than the tolerance threshold δ of the application.

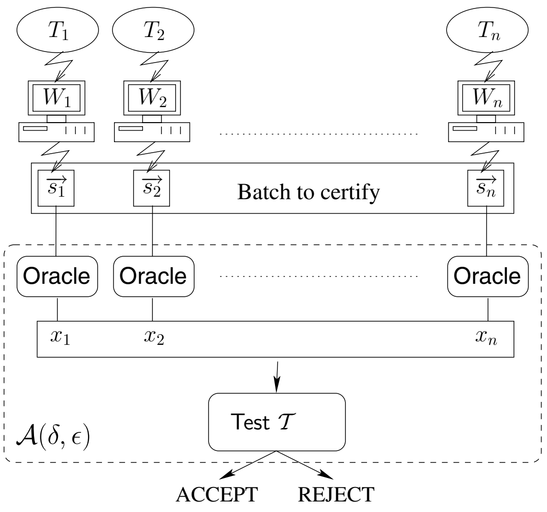


Figure 11.11. *Result certification by task duplication in [GER 05]*

The first approach to ensure the validity of a batch $[\overrightarrow{s_1}, \ldots, \overrightarrow{s_n}]$ is illustrated in Figure 11.11 and consists of certifying the quality of the batch as soon as possible through a tester $\mathcal{T}$ operating a probabilistic hypothesis testing. The correctness of a given result $\overrightarrow{s_i}$ is checked in a deterministic way by an oracle, typically by re-executing the task T_i on reliable resources and giving a binary answer x_i (0=correct,1=falsified). Therefore, given $\epsilon \in [0, 1]$, the certification algorithm $\mathcal{A}(\delta, \epsilon)$ should determine whether the falsification probability p of the initial batch is under the tolerance threshold δ of the application. ϵ corresponds to a "false-positive" (i.e. the probability of answering ACCEPT while $p > \delta$) the user is ready to accept. In the framework of statistical testing, and more precisely of sequential analysis, [GER 05] proposed a probabilistic test that certifies the batch in an adaptive way, without unduly eliminating results which are actually correct and with a relatively low cost (defined as the number of oracle calls).

In the second approach, initiated in [SAR 02], various techniques are proposed to reinforce the quality of the batch. More precisely, concepts like voting, spot-checking, blacklisting, or credibility-based fault-tolerance are applied to decrease the error rate of the batch (characterized by the falsification probability p) under the threshold δ. This approach is illustrated in Figure 11.12.

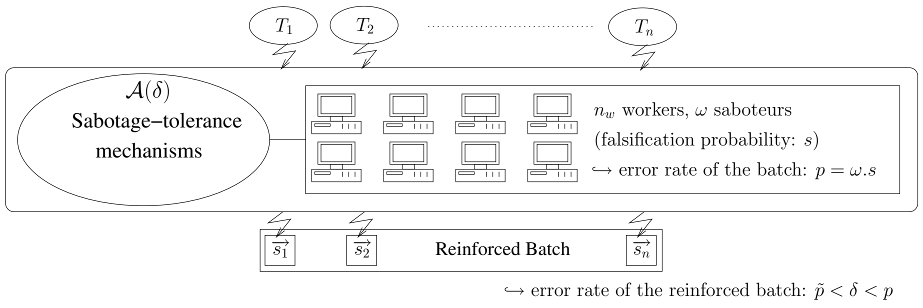

T_1
T_2
T_n
$\mathcal{A}(\delta)$
Sabotage–tolerance mechanisms
n_w workers, ω saboteurs
(falsification probability: s)
$\hookrightarrow$ error rate of the batch: $p = \omega.s$
$\vec{s_1}$
$\vec{s_2}$
Reinforced Batch
$\vec{s_n}$
$\hookrightarrow$ error rate of the reinforced batch: $\tilde{p} < \delta < p$

Figure 11.12. *General configuration of batch reinforcement proposed in [SAR 02]*

Result-checking by partial duplication based on data-flow graphs

The preceding approaches are limited to the restricted context of independent tasks with a modelization of attacker behavior.

Falsification in systems with task dependencies are addressed in [GAO 04] where tasks are determined to execute on reliable or non-reliable nodes in order to maximize the expected number of correct results. The problem is shown to be NP-hard. Whereas the approach considers the critical issue of fault propagation, it is deterministic and therefore could be exploited by a clever adversary.

Indeed, it is possible to extend the probabilistic approaches for direct certification presented in section 11.6.3 to any parallel computation with dependent tasks, making no particular assumption on the attack or on the distribution of errors. The approach is based on an abstract and portable representation of the distributed execution of a parallel program **P** over a fixed input: a bipartite direct acyclic graph $G = (\mathcal{V}, \mathcal{E})$ known as a *macro-dataflow graph*.

Specifically, $\mathcal{V}$ is the finite set of vertices v_j and $\mathcal{E}$ is the set of edges e_{jk}, $j \neq k$, representing precedence relations between $v_j, v_k \in \mathcal{V}$. The vertex set consists of two kinds of tasks. The first class of vertices is associated with the tasks (in the sequential scheduling sense when a task is seen as the smallest program unit of execution) whereas the second represents the parameters of the tasks – either inputs or outputs according to the direction of the edge. The total number of tasks T_j in G is denoted by $|G| = n$. An example of such a dataflow graph is proposed in Figure 11.13 where tasks and data are represented as circles and squares respectively.

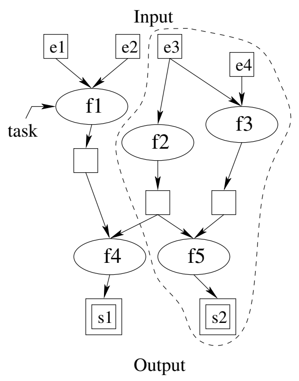


Figure 11.13. *Instance of a data-flow graph associated with the execution of five tasks* $\{f_1, ..., f_5\}$*, with input parameters* $\{e_1, ..., e_4\}$*. The produced results are* $\{s_1, s_2\}$

Modeling an execution by a data-flow graph is part of many parallel programming languages such as Jade [RIN 98] or Athapascan [GAL 98]. Furthermore, some efficient execution engines like KAAPI [GAU 07], introduced in section 11.4.6, use the graph G to build, schedule, and execute programs on distributed architectures. In addition, the graph describes a consistent global state of the execution, which can be used for some checkpointing mechanisms able to ensure fault-tolerance against crash-faults [JAF 04] (see again section 11.4.6 for more details). Finally, the checkpoint could be applied to extract the context of a task in order to re-execute it on safe resources. This assumed a checkpoint server (eventually distributed) hosted on reliable and secured resources. It permits the duplication of a single task for further comparison in a result-checking algorithm, an operation typically conducted by the oracles introduced in [GER 05].

The impact of a result falsification, whether in the context of independent or dependent tasks, is illustrated in Figures 11.14 and 11.15, respectively. Cleverly used, the information presented in the graph limits the overhead of the the tasks re-executed on the oracles. This leads to a *probabilistic certification* that establishes whether the results of the computation are correct or not, as introduced in [VAR 04]. Actually, [VAR 04] but also [KRI 05a, KRI 05b] described various certification algorithms based on macro-dataflow graphs to check the results of an execution composed by dependent tasks. An experimental study conducted in [VAR 06] made use of those results in the framework

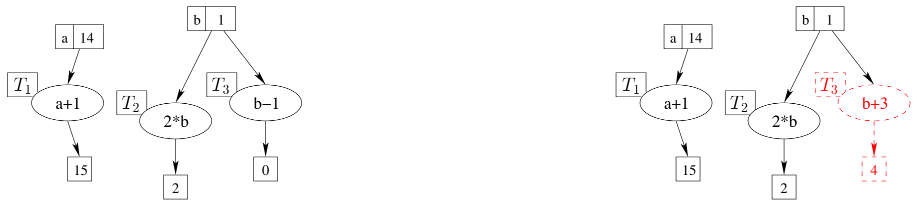


Figure 11.14. *Correct and falsified execution of a parallel application composed of three independent tasks*

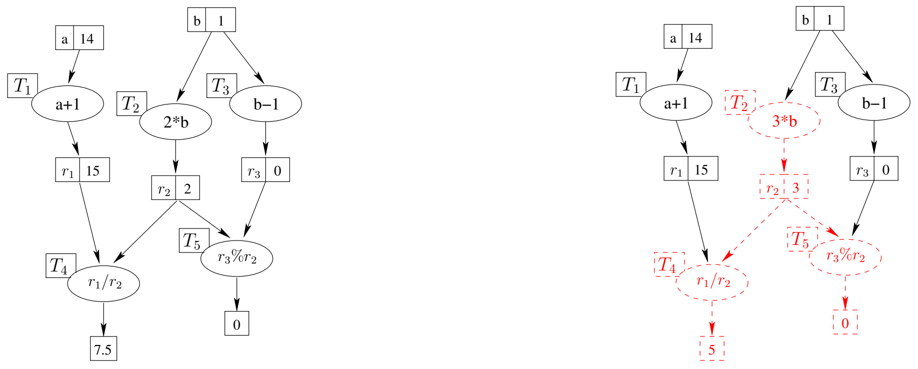


Figure 11.15. *Correct and falsified execution of a parallel application composed of five dependent tasks*

of a medical application. Furthermore, a more developed cost analysis based on on-line scheduling by work-stealing has been conducted in [ROC 07]. Finally, the reader may be interested by the consistent view of these works summarized in [VAR 07].

BOINC case study

The BOINC middleware is a popular volunteer computing system that enables huge computing power using thousands of Internet resources, typically PCs. This has been introduced in Chapter 10. Many types of attacks are possible on volunteer computing systems and more specifically on BOINC. The most common attacks are listed below:

– *result falsification*: attackers return incorrect results;

– *credit falsification*: attackers return results claiming more CPU time than was actually used;

– *malicious executable distribution*: attackers break into a BOINC server and, by modifying the database and files, attempt to distribute their own executable (e.g. a virus program, a malware, and/or a worm) disguised as a BOINC application;

– *denial of service on the data server*: attackers repeatedly send large files to BOINC data servers, filling up their disks, and making them unusable;

– *theft of participant account information by server attack*: attackers break into a BOINC server and steal email addresses and other account information;

– *theft of participant account information by network attack*: attackers exploit the BOINC network protocols to steal account information;

– *theft of project files*: attackers steal input and/or output files.

BOINC provides mechanisms to reduce the likelihood of some of these attacks. For instance, result and credit falsification can be limited by using replication and result checking techniques as described in section 11.6. To be protected against malicious executable distribution, BOINC relies on code signing with a trusted certification authority (CA) independent from the project server so that even if attackers break into a project's BOINC server, they will not be able to cause clients to accept a tampered code file. There is no real solution against denial of service attacks, as for any system connected to the Internet. BOINC attempts to bound output file size and rely on digital certificates for uploading operation but this is clearly insufficient. Finally, theft of information is not directly treated in BOINC.

The different levels protecting the system are the following:

– the security model aims to enforce the trust between volunteers and the project itself. At installation time, the project owner produces a pair of public/private keys and stores them in a safe place, typically, in a machine isolated from the network, as recommended on the BOINC web site. When volunteers contribute for the first time to the project, they obtain the public key of the project. Project owners have to digitally sign the project application files, so that volunteers can verify that the binary codes downloaded by the BOINC client really belong to the project. This mechanism ensures that, if a pirate gets access to one of the BOINC servers, he would not be able to upload malicious code to hundreds of thousands resources. If volunteers trust the projects, the opposite is not true;

– to protect against malicious users [SAR 02], BOINC implements a result certification mechanism, based on redundant computation. BOINC gives the ability to project administrators to write their own custom results certifying code according to their application. As stated in [AND 04], BOINC implements redundant computing using several server daemon processes:

- the *transitioner* implements the redundant computing logic: it generates new results as needed and identifies error conditions,

- the *validator* examines sets of results and selects canonical results. It includes an application-specific result-comparison function,

- the *assimilator* handles newly-found canonical results. It includes an application-specific function that typically parses the result and inserts it into a science database,

- the *file deleter* deletes input and output files from data servers when they are no longer needed.

In this architecture servers and daemons can run on different hosts and can be replicated, so BOINC servers are scalable as stated by D. Anderson in [AND 04]. Availability is enhanced because some daemons can run even while parts of the project are down (for example, the scheduling server and transitioner can operate even if the science database is down).

Moreover, BOINC provides a feature called *homogenous redundancy* for scientific applications, in particular those that generate many floating point values. When this feature is enabled, the BOINC scheduler sends results for a given task only to hosts with the same operating system name and CPU vendor. In this case, strict equality can be used to compare results.

11.7. Conclusion

In this chapter, we have covered many practical and fundamental aspects of safety in a broad sense for one class of distributed systems, namely grid systems. After an introduction to grid technologies, we have visited the main concepts of safety in grid systems: confidentiality, authentication, availability, integrity, and resource integrity. We have also introduced one specific technique used for desktop grids, namely result checking.

As grid systems are now evolving towards cloud systems, we may observe a revitalization of research topics and we are now moving from coordination issues to provisioning issues. People from the community will probably have to envisage security and safety issues according to this new context and paradigm.

In clouds that provide Internet-based services (computing and storage), security issues are becoming a key differentiator and competitive edge between cloud providers. For instance, if we consider a cloud infrastructure for journalists made of services for writing articles, reporting with experts, data consulting, and on-line storage, it can be noted that if not all the previous human activities are secured, democracies are at risk.

To prevent damage, the TPM monitors and reports on what is running on the user's machine. This monitoring and reporting are particularly important in the virtualized environment of cloud computing. In the previous example, trusted network connection is also important, as well as trusted storage: these two issues are inherently tackled with TPM. The Cloud Security Alliance (CSA)[35] released a report that identifies many areas for concern in cloud computing. This new environment cannot be protected using traditional security approaches, even those inherited from grid computing. We would like to mention the separation between customers (TPM could

35. http://cloudsecurityalliance.org.

provide hardware-based verification of hypervisor and virtual machine integrity) and the cloud's legal and regulatory issues (to verify that a cloud provider has a strong policy and practice that address legal and regulatory issues for provisioning customers, for instance). In the areas of data retention and deletion, trusted storage and TPM techniques may play a major role in the future.

11.8. Bibliography

[ABB 09] ABBES H., CÉRIN C., JEMNI M., "BonjourGrid: orchestration of multi-instances of grid middlewares on institutional desktop grids", *IPDPS*, IEEE, p. 1-8, 2009.

[AGU 97] AGUILERA M. K., CHEN W., TOUEG S., "Heartbeat: a timeout-free failure detector for quiescent reliable communication", MAVRONICOLAS M., TSIGAS P., Eds., *Proc. of the 11th Workshop on Distributed Algorithms (WDAG'97)*, vol. 1320 of *Lecture Notes in Computer Science*, Springer, p. 126-140, 1997.

[ALI 04] ALI S., MACIEJEWSKI A., SIEGEL H., K. J.-K., "Measuring the robustness of a resource allocation", *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, p. 630–641, 2004.

[ALV 95] ALVISI L., MARZULLO K., "Message logging: pessimistic, optimistic, and causal", *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS 1995)*, IEEE CS Press, p. 229-236, May-June 1995.

[AND 02] ANDERSON D. P., COBB J., KORPELA E., LEBOFSKY M., WERTHIMER D., "SETI@home: an experiment in public-resource computing", *Commun. ACM*, vol. 45, p. 56-61, 2002.

[AND 04] ANDERSON D. P., "BOINC: a system for public-resource computing and storage", *5th IEEE/ACM Int. Workshop on Grid Computing*, Pittsburgh, USA, Nov 8, 2004.

[AVI 04] AVIZIENIS A., LAPRIE J., RANDELL B., "Dependability and its threats: a taxonomy", *Building the Information Society: IFIP 18th World Computer Congress: Topical sessions August 22-27, 2004, Toulouse, France*, Kluwer Academic Publishers, Page 91, 2004.

[BAR 01a] BARAK B., GOLDREICH O., IMPAGLIAZZO R., RUDICH S., SAHAI A., VADHAN S. P., YANG K., "On the (im)possibility of obfuscating programs", *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, London, Springer-Verlag, p. 1–18, 2001.

[BAR 01b] BARRETT D. J., SILVERMAN R., *SSH, the Secure Shell: the Definitive Guide*, O'Reilly, 2001.

[BES 08] BESSERON X., GAUTIER T., "Optimised recovery with a coordinated checkpoint/rollback protocol for domain decomposition applications", AN L. T. H., BOUVRY P., TAO P. D., Eds., *Second International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences (MCO 2008)*, vol. 14 of *Communications in Computer and Information Science*, Springer-Verlag, p. 497-506, 2008.

[BIR 95] BIRD R., GOPAL I., HERZBERG A., JANSON P., KUTTEN S., MOLVA R., YUNG M., "KryptoKnight family of light-weight protocols for authentication and key distribution", *IEEE/ACM Transactions on Networking*, vol. 3, p. 31–41, 1995.

[BLU 89] BLUM M., KANNAN S., "Designing programs that check their work", *21st ACM Symposium on the Theory of Computing*, ACM Press, p. 86–97, 1989.

[BOI10] "BOINC", 2010, http://boinc.berkeley.edu/.

[BOR 83] BORG A., BAUMBACH J., GLAZER S., "A message system supporting fault tolerance", *SOSP '83: Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, New York, NY, USA, ACM Press, p. 90–99, 1983.

[BOS 02] BOSILCA G., BOUTEILLER A., CAPPELLO F., DJILALI S., FÉDAK G., GERMAIN C., HÉRAULT T., LEMARINIER P., LODYGENSKY O., MAGNIETTE F., NÉRI V., SELIKHOV A., "MPICH-V: toward a scalable fault tolerant MPI for volatile nodes", *High Performance Networking and Computing (SC|02)*, Baltimore USA, IEEE/ACM, November 2002.

[BOS 09] BOSILCA G., DELMAS R., DONGARRA J., LANGOU J., "Algorithm-based fault tolerance applied to high performance computing", *J. Parallel Distrib. Comput.*, vol. 69, num. 4, p. 410-416, 2009.

[BOU 03] BOUTEILLER A., CAPPELLO F., HÉRAULT T., KRAWEZIK G., LEMARINIER P., MAGNIETTE F., "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging", *High Performance Networking and Computing (SC|03)*, Phoenix USA, IEEE/ACM, November 2003.

[BOU 05] BOUTEILLER A., COLLIN B., HERAULT T., LEMARINIER P., CAPPELLO F., "Impact of event logger on causal message logging protocols for fault tolerant MPI", *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Washington, DC, USA, IEEE Computer Society, Page 97, 2005.

[BOU 06] BOUTEILLER A., HÉRAULT T., KRAWEZIK G., LEMARINIER P., CAPPELLO F., "MPICH-V project: a multiprotocol automatic fault-tolerant MPI", *IJHPCA*, vol. 20, p. 319–333, 2006.

[BOY 82] BOYER R. S., MOORE J. S., *The Correctness Problem in Computer Science*, Academic Press, Orlando, FL, 1982.

[BUN 08] BUNTINAS D., COTI C., HERAULT T., LEMARINIER P., PILARD L., REZMERITA A., RODRIGUEZ E., CAPPELLO F., "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI", *Future Generation Computer Systems*, vol. 24, p. 73-84, 2008, DOI: http://dx.doi.org/10.1016/j.future.2007.02.002.

[CAP 05a] CAPPELLO F., CARON E., DAYDE M., DESPREZ F., JEGOU Y., PRIMET P. V.-B., JEANNOT E., LANTERI S., LEDUC J., MELAB N., MORNET G., QUETIER B., RICHARD O., "Grid'5000: a large scale and highly reconfigurable grid experimental testbed", *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing CD (SC|05)*, Seattle, Washington, USA, IEEE/ACM, p. 99–106, November 2005, http://www.grid5000.fr.

[CAP 05b] CAPPELLO F., DJILALI S., FEDAK G., HÉRAULT T., MAGNIETTE F., NÉRI V., LODYGENSKY O., "Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid", *Future Generation Comp. Syst.*, vol. 21, p. 417-437, 2005.

[CAS 90] CASE J., FEDOR M., SCHOFFSTALL M., DAVIN J., A simple network management protocol (SNMP), RFC num. 1157, Report of the Network Working Group, May 1990.

[CHA 85] CHANDY K. M., LAMPORT L., "Distributed snapshots: determining global states of distributed systems", *Transactions on Computer Systems*, vol. 3, p. 63-75, 1985.

[CHA 96] CHANDRA T., HADZILACOS V., TOUEG S., CHARRON-BOST B., "Impossibility of group membership in asynchronous systems", *Proceedings of the 15th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, p. 322-330, May 1996.

[CHA 04] CHAKRAVORTY S., KALÉ L. V., "A fault tolerant protocol for massively parallel machines", *FTPDS Workshop for IPDPS 2004*, IEEE Press, 2004.

[CHA 06] CHAKRAVORTY S., MENDES C. L., KALÉ L. V., "Proactive fault tolerance in MPI applications via task migration", *HiPC*, vol. 4297 of *Lecture Notes in Computer Science*, Springer, p. 485-496, 2006.

[CHA 07] CHAKRAVORTY S., KALÉ L. V., "A fault tolerance protocol with fast fault recovery", *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, IEEE Press, 2007.

[CHE 05] CHEN Z., FAGG G. E., GABRIEL E., LANGOU J., ANGSKUN T., BOSILCA G., DONGARRA J., "Fault tolerant high performance computing by a coding approach", PINGALI K., YELICK K. A., GRIMSHAW A. S., Eds., *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*, ACM, p. 213–223, 2005.

[CHE 09] CHEN L., MITCHELL C. J., MARTIN A., Eds., *Trusted Computing, Second International Conference, Trust 2009, Oxford, UK, April 6-8, 2009, Proceedings*, vol. 5471 of *Lecture Notes in Computer Science*, Springer, 2009.

[CHI 03] CHIEN A. A., CALDER B., ELBERT S., BHATIA K., "Entropia: architecture and performance of an enterprise desktop grid system", *J. Parallel Distrib. Comput.*, vol. 63, p. 597-610, 2003.

[CLA 09] CLAUDEL B., HUARD G., RICHARD O., "TakTuk, adaptive deployment of remote executions", KRANZLMÜLLER D., BODE A., HEGERING H.-G., CASANOVA H., GERNDT M., Eds., *HPDC*, ACM, p. 91-100, 2009.

[COL 02] COLLBERG C. S., THOMBORSON C., "Watermarking, tamper-proofing, and obfuscation – tools for software protection", vol. 28, 2002.

[DEV 98] DEVALE J., *Traditional Reliability*, vol. 18-849b of *Dependable Embedded Systems*, Carnegie Mellon University, 1998.

[DIE 99a] DIERKS T., ALLEN C., The TLS protocol – version 1.0, Report num. 2246, 1999, Outmoded by RFC 4346, updated by RFCs 3546, 5746.

[DIE 99b] DIERKS T., ALLEN C., The TLS protocol – version 1.0, Request for Comments (RFC) num. 2246, Network Working Group, 1999.

[DUC 06] DUC G., KERYELL R., "CryptoPage: an efficient secure architecture with memory encryption, integrity and information leakage protection", *ACSAC*, IEEE Computer Society, p. 483-492, 2006.

[ELN 92] ELNOZAHY E. N., ZWAENEPOEL W., “Manetho: transparent rollback-recovery with low overhead, limited rollback, and fast output commit”, *IEEE Trans. Computers*, vol. 41, p. 526–531, 1992.

[FAG 04] FAGG G. E., DONGARRA J., “Building and using a fault-tolerant MPI implementation”, *International Journal of High Performance Computing Applications*, vol. 18, num. 3, p. 353-361, 2004.

[FED 01] FEDAK G., GERMAIN C., NÉRI V., CAPPELLO F., “XtremWeb: a generic global computing system”, *IEEE CCGrid2001 (Global Computing on Personal Devices)*, IEEE Computer Society, p. 582-587, 2001.

[FIS 85] FISCHER M. J., LYNCH N. A., PATERSON M., “Impossibility of distributed consensus with one faulty process”, *J. ACM*, vol. 32, p. 374-382, 1985.

[FOR 94] FORUM M. P. I., MPI: a message-passing interface standard, Report num. UT-CS-94-230, Department of Computer Science, University of Tennessee, April 1994.

[FOS 97] FOSTER I., KESSELMAN C., “Globus: a metacomputing infrastructure toolkit”, *International J. of Supercomputer Applications and High Performance Computing*, vol. 11, p. 115–128, 1997.

[FOS 01] FOSTER I., KESSELMAN C., TUECKE S., “The anatomy of the grid – enabling scalable virtual organizations”, *International Journal of Supercomputer Applications*, vol. 15, 2001.

[GAL 98] GALILÉE F., ROCH J.-L., CAVALHEIRO G., DOREILLE M., “Athapascan-1: online building data flow graph in a parallel language”, *PACT’98*, Paris, France, IEEE, p. 88–95, October 1998.

[GAO 04] GAO L., MALEWICZ G., “Internet computing of tasks with dependencies using unreliable workers”, *8th International Conference on Principles of Distributed Systems (OPODIS’04)*, vol. 3544 of *LNCS*, Springer-Verlag, p. 443-458, 2004.

[GAU 07] GAUTIER T., BESSERON X., PIGEON L., “KAAPI: a thread scheduling runtime system for data flow computations on cluster of multi-processors”, MAZA M. M., WATT S. M., Eds., *Proceedings of the International Workshop on Parallel Symbolic Computation (PASCO)*, ACM, p. 15-23, 2007.

[GEI 96] GEIST A., GROPP W. D., HUSS-LEDERMAN S., LUMSDAINE A., LUSK E. L., SAPHIR W., SKJELLUM A., SNIR M., “MPI-2: extending the message-passing interface”, BOUGÉ L., FRAIGNIAUD P., MIGNOTTE A., ROBERT Y., Eds., *1st European Conference on Parallel and Distributed Computing (EuroPar’96)*, vol. 1123 of *Lecture Notes in Computer Science*, Springer, p. 128–135, 1996.

[GEN 09] GENTRY C., A fully homomorphic encryption scheme, PhD thesis, Stanford University, 2009.

[GER 05] GERMAIN C., MONNIER-RAGAIGNE D., “Grid result checking”, *Proceedings of the 2nd Conference on Computing Frontiers*, Ischia, Italy, ACM Press, p. 87–96, May 2005.

[GNU 01] GNU, Ed., “*The GNU C library reference manual*”, Chapter System Databases and Name Service Switch, The Gnu Project, 2001, http://www.gnu.org/software/libc/manual/html_node/Name-Service-Switch.html.

[GOL 98] GOLDREICH O., GOLDWASSER S., RON D., “Property testing and its connection to learning and approximation”, *Journal of the ACM*, vol. 4, p. 653–750, 1998.

[GRO 96] GROPP W. D., LUSK E. L., DOSS N., SKJELLUM A., “High-performance, portable implementation of the MPI message passing interface standard”, *Parallel Computing*, vol. 22, p. 789-828, Elsevier, 1996.

[GRO 97] GROUP X., X/Open Single Sign-On Service (XSSO) – pluggable authentication, Report num. P702, X/Open Group, June 1997, http://www.opengroup.org/pubs/catalog/p702.htm.

[HAL 95] HALLER N., The S/KEY one-time password system, Report num. 1760, February 1995.

[HAL 98] HALLER N., METZ C., NESSER P., STRAW M., A one-time password system, Report num. 2289, February 1998.

[HEL 99] HELARY J.-M., MOSTEFAOUI A., RAYNAL M., “Communication-induced determination of consistent snapshots”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, p. 865–877, 1999.

[HOH 98] HOHL F., “Time limited blackbox security: protecting mobile agents from malicious hosts”, VIGNA G., Ed., *Mobile Agents and Security*, vol. 1419 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, p. 92-113, 1998.

[ISO 98] International Organization for Standardization, Geneva, Switzerland, ISO/IEC 9798-3: information technology – Security techniques – Entity authentication – part 3: Mechanisms using digital signature techniques, 2nd edition, 1998.

[ISO 99] International Organization for Standardization, Geneva, Switzerland, ISO/IEC 9798-4: information technology – security techniques – entity authentication – part 4: mechanisms using a cryptographic check function, 2nd edition, 1999.

[ISO 05] International Organization for Standardization, Geneva, Switzerland, ISO/IEC 27001: information technology – security techniques – information security management systems – requirements, 2005.

[ISO 08] International Organization for Standardization, Geneva, Switzerland, ISO/IEC 9798-2: information technology – security techniques – entity authentication – part 2: mechanisms using symmetric encipherment algorithms, 3rd edition, 2008.

[ISO 09a] International Organization for Standardization, Geneva, Switzerland, ISO/IEC 11889-1: information technology – trusted platform module – part 1: overview, 2009.

[ISO 09b] International Organization for Standardization, Geneva, Switzerland, ISO/IEC 9798-5: information technology – security techniques – entity authentication – part 5: mechanisms using zero-knowledge techniques, 3rd edition, 2009.

[ISO 10] International Organization for Standardization, Geneva, Switzerland, ISO/IEC 9798-1: information technology – security techniques – entity authentication – part 1: generalities, 3rd edition, 2010.

[JAF 04] JAFAR S., VARRETTE S., ROCH J.-L., "Using data-flow analysis for resilience and result checking in peer-to-peer computations", *IEEE DEXA'2004 – Workshop GLOBE'04: Grid and Peer-to-Peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems*, Zaragoza, Spain, IEEE, p. 512–516, September 2004.

[JAF 05a] JAFAR S., GAUTIER T., KRINGS A., ROCH J.-L., "A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing", *Proceedings of the 11th European Conference on Parallel and Distributed Computing (EuroPar'05)*, vol. 3648 of *Lecture Notes in Computer Science*, p. 675–684, 2005.

[JAF 05b] JAFAR S., KRINGS A. W., GAUTIER T., ROCH J.-L., "Theft-induced checkpointing for reconfigurable dataflow applications", *IEEE Electro/Information Technology Conference (EIT 2005)*, IEEE, May 2005.

[JOH 87] JOHNSON D. B., ZWAENEPOEL W., "Sender-based message logging", *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing (17th FTCS'87)*, IEEE Comp. Soc. Press, p. 14–19, July 1987.

[JOH 90] JOHNSON D. B., ZWAENEPOEL W. E., "Recovery in distributed systems using optimistic message logging and checkpointing", *Journal of Algorithms*, vol. 11, p. 462–491, 1990.

[KAD 05] "kadeploy", 2005, http://kadeploy.imag.fr.

[KAL 93] KALÉ L., KRISHNAN S., "CHARM++: a portable concurrent object oriented system based on C++", PAEPCKE A., Ed., *Proceedings of OOPSLA'93*, ACM Press, p. 91–108, September 1993.

[KAL 98] KALISKI B., "PKCS #1: RSA encryption – version 1.5", RFC 2313 (Informational), March 1998, Outmoded by RFC 2437.

[KER] KERBEROS, MIT, Massachusetts, USA, http://web.mit.edu/kerberos.

[KON 04] KONDO D., TAUFER M., III C. L. B., CASANOVA H., CHIEN A. A., "Characterizing and evaluating desktop grids: an empirical study", *IPDPS*, IEEE Computer Society, 2004.

[KON 07] KONDO D., ARAUJO F., MALECOT P., DOMINGUES P., SILVA L. M., FEDAK G., CAPPELLO F., "Characterizing error rates in internet desktop grids", *Proceedings of the 13th European Conference on Parallel and Distributed Computing (Euro-Par'07)*, August 2007.

[KRI 05a] KRINGS A., ROCH J.-L., JAFAR S., VARRETTE S., "A probabilistic approach for task and result certification of large-scale distributed applications in hostile environments", *European Grid Conference (EGC2005)*, vol. 3470 of *LNCS*, Amsterdam, Netherlands, Springer Verlag, February 14–16, 2005.

[KRI 05b] KRINGS A. W., ROCH J.-L., JAFAR S., "Certification of large distributed computations with task dependencies in hostile environments", *IEEE Electro/Information Technology Conference, (EIT 2005)*, Lincoln, Nebraska, IEEE, May 2005.

[KWO 07] KWOK Y.-K., "*The Handbook of Computer Networks*", vol. 3, Chapter Incentive Issues in Peer-to-Peer Systems, p. 168–188, John Wiley & Sons, 2007.

[LAM 78] LAMPORT L., "Time, clocks, and the ordering of events in a distributed system", *Commun. ACM*, vol. 21, p. 558–565, 1978.

[LAM 78] LAMPORT L., "Time, clocks, and the ordering of events in a distributed system", *Commun. ACM*, vol. 21, p. 558–565, 1978.

[LAM 81] LAMPORT L., "Password authentication with insecure communication", *Communications of the ACM*, vol. 24, p. 770–772, 1981.

[LEM 04] LEMARINIER P., BOUTEILLER A., HERAULT T., KRAWEZIK G., CAPPELLO F., "Improved message logging versus improved coordinated checkpointing for fault tolerant MPI", *IEEE International Conference on Cluster Computing (Cluster 2004)*, IEEE CS Press, 2004.

[LO 04] LO V. M., ZAPPALA D., ZHOU D., LIU Y., ZHAO S., "Cluster computing on the fly: P2P scheduling of idle cycles in the internet", VOELKER G. M., SHENKER S., Eds., *IPTPS*, vol. 3279 of *Lecture Notes in Computer Science*, Springer, p. 227-236, 2004.

[LOU 02] LOUREIRO S., BUSSARD L., ROUDIER Y., "Extending tamper-proof hardware security to untrusted execution environments", *CARDIS*, p. 111-124, 2002.

[MAR 08] MARTIN A., The ten-page introduction to trusted computing, Report , Software Engineering Group, Oxford University Computing Laboratory, Oxford, CS-RR-08-11, 2008.

[MEN 96] MENEZES A. J., VANSTONE S. A., OORSCHOT P. C. V., *Handbook of applied cryptography*, CRC Press, 1st edition, 1996, http://www.cacr.math.uwaterloo.ca/hac/.

[MOL 92] MOLVA R., TSUDIK G., VAN HERREWEGHEN E., ZATTI S., "KryptoKnight authentication and key distribution system", *Proceedings of the Second European Symposium on Research in Computer Security (ESORICS'92)*, London, UK, Springer-Verlag, p. 155–174, Jan. 1992.

[MOL 00] MOLNAR D., The SETI@Home Problem, http://www.acm.org/crossroads/columns /onpatrol/september2000.html, November 2000.

[MOR 99] MORRISETT G., CRARY K., GLEW N., GROSSMAN D., SAMUELS R., SMITH F., WALKER D., WEIRICH S., ZDANCEWIC S., "TALx86: a realistic typed assembly language", *Second Workshop on Compiler Support for System Software (WCSSS'99)*, Atlanta, Georgia, USA, ACM, May 1999.

[M'R 05] M'RAIHI D., BELLARE M., HOORNAERT F., NACCACHE D., RANEN O., HOTP: an HMAC-based one-time password algorithm, Report num. 4226, 2005.

[NEU 94] NEUMAN C., TS'O T., "Kerberos: an authentication service for computer networks", *IEEE Communications Magazine*, vol. 32, p. 33–38, 1994, http://gost.isi.edu/publications/kerberos-neuman-tso.html.

[NEU 04] NEUMAN C., YU T., HARTMAN S., RAEBURN K., The Kerberos network authentication service (V5) IETF, Report , USC-ISI/MIT, 2004, http://tools.ietf.org/id/draft-ietf-krb-wg-kerberos-clarifications-05.txt.

[NMA] "Nmap – free security scanner for network exploration & security audits".

[NOR 03] NORASWAMY N., HARKINS D., *IPSec*, CampusPress, 2003.

[OTP05] One-Time Password Specifications (OTPS), 2005, http://www.rsa.com/rsalabs/ node.asp?id=2816.

[PAI 99] PAILLIER P., "Public-key Cryptosystems Based on Composite Degree Residuosity Classes", Springer-Verlag, p. 223–238, 1999.

[PLA 97] PLANK J. S., KIM Y., DONGARRA J., "Fault-tolerant matrix operations for networks of workstations using diskless checkpointing", *J. Parallel Distrib. Comput.*, vol. 43, p. 125-138, 1997.

[RES 01] RESCOLA E., *SSL and TLS – designing and building secure systems*, Addison-Wesley, 1st edition, 2001.

[RIN 98] RINARD M. C., LAM M. S., "The design, implementation, and evaluation of Jade", *ACM Transactions on Programming Languages and Systems*, vol. 20, p. 483–545, 1998.

[ROC 07] ROCH J.-L., VARRETTE S., "Probabilistic certification of divide & conquer algorithms on global computing platforms. Application to fault-tolerant exact matrix-vector product.", *ACM International Workshop on Parallel Symbolic Computation'07 (PASCO'07)*, London, Ontario, Canada, ACM, p. 88–92, July 2007.

[SAR 02] SARMENTA L. F. G., "Sabotage-tolerance mechanisms for volunteer computing systems", *Future Generation Computer Systems*, vol. 18, p. 561-572, Elsevier, 2002.

[SCH 07] SCHERRER A., LARRIEU N., OWEZARSKI P., BORGNAT P., ABRY P., "Non-Gaussian and long memory statistical characterizations for internet traffic with anomalies", *IEEE Trans. Dependable Secur. Comput.*, vol. 4, p. 56–70, 2007.

[SIE 91] SIEWIOREK D. P., SWARZ R. S., *Reliable Computer Systems: Design and Evaluation, 2nd Edition*, Butterworth, 1991.

[STR 85] STROM R. E., YEMINI S. A., "Optimistic recovery in distributed systems", *Transactions on Computer Systems*, vol. 3, p. 204-226, 1985.

[THA 05] THAIN D., TANNENBAUM T., LIVNY M., "Distributed computing in practice: the Condor experience", *Concurrent Computing – Practice and Experience*, vol. 17, p. 323-356, John Wiley and Sons Ltd., February 2005.

[TRI 01] TRIVEDI K. S., *Probability and statistics with reliability, queuing and computer science applications*, John Wiley and Sons, New York, 2001.

[TUE 04] TUECKE S., WELCH V., ENGERT D., PEARLMAN L., THOMPSON M., Internet X.509 public key infrastructure (PKI) – proxy certificate profile, Request for Comments (RFC) num. 3820, Network Working Group, June 2004.

[U.S97] U.S. Department of Commerce – N.I.S.T., Geneva, Switzerland, FIPS 196: entity authentication using public key cryptography, February 1997.

[VAR 04] VARRETTE S., ROCH J.-L., LEPREVOST F., "FlowCert: probabilistic certification for peer-to-peer computations", *16th Symposium on Computer Architecture and High Performance Computing, IEEE SBAC-PAD 2004*, Foz do Iguaçu, Brazil, IEEE, p. 108–115, October 2004.

[VAR 05] VARRETTE S., GEORGET S., MONTAGNAT J., ROCH J.-L., LEPREVOST F., "Distributed authentication in GRID5000", *Proc. of OTM Confederated Int. Workshops on Grid Computing and its Application to Data Analysis (GADA'05)*, vol. 3762 of *LNCS*, Ayia Napa, Cyprus, Springer Verlag, p. 314–326, November 1 2005.

[VAR 06] VARRETTE S., ROCH J.-L., MONTAGNAT J., SEITZ L., PIERSON J.-M., LEPREVOST F., "Safe distributed architecture for image-based computer assisted diagnosis", *IEEE 1st International Workshop on Health Pervasive Systems (HPS'06)*, Lyon, France, June 2006.

[VAR 07] VARRETTE S., Security in large scale computing systems: authentication and result-checking, PhD thesis, INP Grenoble and University of Luxembourg, 2007.

[VAR 11] VARRETTE S., TANTAR E., BOUVRY P., "On the resilience of [distributed] evolutionary algorithms against cheaters in global computing platforms", *Proc. of the 14th International Workshop on Nature Inspired Distributed Computing (NIDISC 2011), part of the 25th IEEE/ACM Intl. Parallel and Distributed Processing Symposium (IPDPS 2011)*, Anchorage (Alaska), USA, May 2011.

[WAH 97] WAHL M., HOWES T., KILLE S., RFC 2251 – lightweight directory access protocol (v3), Report, Internet Engineering Task Force, 1997, http://www.ietf.org/rfc/rfc2251.txt.

[WAS 97] WASSERMAN H., BLUM M., "Software reliability via run-time result-checking", *Journal of the ACM*, vol. 44, p. 826–849, 1997.

[WIE 06] WIESMANN M., URBÁN P., DÉFAGO X., "An SNMP based failure detection service", *25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006)*, IEEE Computer Society, p. 365-376, 2006.

[XTR 09] XTREMLAB, "http://xtremlab.lri.fr/", 2009.

[ZHE 04] ZHENG G., SHI L., KALÉ L. V., "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI", *2004 IEEE International Conference on Cluster Computing*, San Diego, CA, p. 93-103, September 2004.

Chapter 12

Enforcing Security with Cryptography

12.1. Introduction

The world famous International Standards Organization (ISO) defines in its norm ISO 27001:2005 (Information technology - security techniques - information security management systems - requirements) the term "confidentiality" as follows: *Confidentiality is a characteristic that applies to information. To protect and preserve the confidentiality of information means to ensure that it is not made available or disclosed to unauthorized entities. In this context, entities include both individuals and processes.*

One way to ensure a high level of confidentiality should be to use some private communication network, with native devices of information protection; for instance some privately operated optical fiber networks between two buildings of a financial institution. Nevertheless, the cost for establishing and maintaining such networks is clearly not compatible with scaling. What is the alternative? *Cryptography*. What is cryptography? A collection of involved mathematical notions, miscellaneous engineering designs, and a large amount of frequently used software that allow thousands of people per hour to buy or sell many articles on the Internet.

The very objective of cryptography is to allow confidential communications between two entities, namely human beings, computers, or any processes, through a public network. A public network is an unrestricted communication medium with

Chapter written by Sami HARARI and Laurent POINSOT.

Distributed Systems: Design and Algorithms Edited by Serge Haddad, Fabrice Kordon, Laurent Pautet and Laure Petrucci
© 2011 ISTE Ltd. Published 2011 by ISTE Ltd.

no access control such as the telephone network or the Internet, for instance. Cryptographic devices should make it impossible to obtain and use information illegitimately "sniffed" out on the public network.

When confidentiality must be guaranteed, for instance for military messages or financial transactions, it is fundamental to employ cryptography. Such techniques may be successfully used to achieve other security requirements, such as integrity, authenticity, and non-repudiation. Even though we acknowledge the previous ideas, this chapter is mainly devoted to the protection of confidentiality.

Generally cryptographic communication between two individuals (or computers or processes), say Alice and Bob, is composed in the following way. Before sending information (on the public network) to Bob, Alice modifies it using a *cryptographic system* (a *cryptosystem* for short), into a new message called *ciphertext* or *cryptogram*, which has the essential property of dissimulating the very nature of the original message, called *cleartext* or *plaintext*, to every entity other than Bob. Bob, the legitimate recipient of Alice's message, retrieves and decrypts the ciphertext in order to recover the original message from Alice. As the network is public and, therefore, freely accessible, any person may be able to intercept the ciphertext. However, because it is encrypted *for* Bob, this message seems to be without meaning and, in fact, unusable. In this way, in principle, the requirement of confidentiality is ensured.

All cryptosystems are quite similar in form and principle, and they all share the same operating process and fulfill similar tasks. Therefore, the first part of this chapter is devoted to the general description of these common features: an accurate definition for cryptosystems will be given as well as a description of high-level functionalities provided by cryptosystem devices. Furthermore the very existence of cryptography is related to threats on communication media; this is the reason why the concept of cryptanalysis is also introduced in the first part. Cryptosystems are classified in one of the following two species: symmetric (or secret-key) cryptosystems, and public-key cryptosystems. A part of this chapter is dedicated to both families. In section 12.3 we deal with the former; we provide a general description for secret-key cryptosystems, and we deal with some popular algorithms, namely the data encryption standard (DES), international data encryption algorithm (IDEA) and advanced encryption standard (AES), in order to illustrate the development of mathematical technologies in this area. The world famous RSA algorithm, as a relevant instance of public key cryptosystems, is detailed in section 12.4. Then we stress the fundamental role played by prime numbers in asymmetric encryption: different techniques to prove primality or to provide prime numbers are presented.

12.2. Cryptography: from a general perspective

In this section we introduce general definitions and the main principles used in the remainder of this chapter.

12.2.1. *Cryptosystems*

The notion of cryptographic systems, although quite imprecise, may be given a rigorous mathematical definition (which is inspired from [STI 06]). Mathematically speaking, an *enciphering algorithm*, also called *cryptosystem* or *cipher system* [1], can be described as a collection of three non-empty sets, $\mathcal{P}$, $\mathcal{C}$ and $\mathcal{K}$ (in general these sets are finite), called sets of *plaintexts*, *ciphertexts*, and *keys*, and two functions [2] $E\colon \mathcal{K} \to \mathcal{C}^{\mathcal{P}}$, which maps a key $k \in \mathcal{K}$ to a *enciphering* (or *encryption*) *function* $E_k\colon \mathcal{P} \to \mathcal{C}$, and $D\colon \mathcal{K} \to \mathcal{P}^{\mathcal{C}}$, which associates with every $k \in \mathcal{K}$ its *deciphering* (or *decryption*) *function* $D_k\colon \mathcal{C} \to \mathcal{P}$, which are required to satisfy a *decryption rule*: for every plaintext $x \in \mathcal{P}$,

$$D_k(E_k(x)) = x\ .$$

This rule is fundamental for the decryption process, and more precisely to make such a process possible. Indeed, let us assume that $y \in \mathcal{C}$ is the ciphertext $E_k(x)$ for some plaintext $x \in \mathcal{P}$ (and key $k \in \mathcal{K}$). The decryption rule indicates that we can recover this plaintext from the ciphertext by an application of the decryption function D_k: $D_k(y) = D_k(E_k(x)) = x$. While quite simple, this is the most important feature of a cryptosystem, and this leads to the search for invertible functions [3] E_k and D_k (for every key k). This is the reason why we will present some cryptographic algorithms emphasizing the property of invertibility satisfied by the encryption/decryption functions.

From this formalism we deduce that Alice must know the map E_k while D_k has to be known by Bob in order to make possible the decryption process. If someone else than Bob is acquainted with the use of D_k, then he is able to decrypt any message enciphered by E_k, and the protection of confidentiality probably fails (except if the person under consideration is Alice!).

1. In the following we will freely use the terms “cryptographic” or “cipher” or “encryption” in order to avoid monotony.

2. Recall that the set of all maps from X to Y is usually denoted Y^X.

3. The decryption rule only implies that for each $k \in \mathcal{K}$, D_k is onto and E_k is one-to-one. Nevertheless, when $\mathcal{P}$ and $\mathcal{C}$ are finite with the same cardinal number, both maps are invertible (we also say that they are “bijective” or “one-one correspondences”).

Example 12.1 The following (secret-key) cryptosystem, called *one-time pad*, was invented by G.S. Vernam in 1917 (while published in 1926 in [VER 26]). Let us denote by $\mathbb{Z}_2$ the set $\{0,1\}$ of bits and let $\oplus$ be the addition of bits modulo two, also called *exclusive or* (shorter: *XOR*), given by the following (addition) table:

$\oplus$	0	1
0	0	1
1	1	0

Vernam's cryptosystem is formalized in the following fashion: $\mathcal{P} = \mathcal{C} = \mathcal{K} = (\mathbb{Z}_2)^\ell$ where ℓ is a positive integer. Therefore plaintexts, ciphertexts, and keys are are ℓ-tuples of bits. For each key $k = (k_1, \dots, k_\ell)$ (where each k_i is a bit) the encryption is defined

$$\begin{array}{llll} E_k : & (\mathbb{Z}_2)^\ell & \to & (\mathbb{Z}_2)^\ell \\ & x = (x_1, \dots, x_\ell) & \mapsto & x \oplus k = (x_1 \oplus k_1, \dots, x_\ell \oplus k_\ell) \; . \end{array}$$

So the ciphertext $E_k(x)$ corresponding to the plaintext x is equal to the component-wise modulo-two sum, which, by abuse, is also called *XOR* (or *exclusive or*), of x and k. The decryption function D_k is equal to E_k. It is quite easy to check that the decryption rule is satisfied. First of all, let us note that for every $x, y \in (\mathbb{Z}_2)^\ell$ it holds that $(x \oplus y) \oplus y = x$ (this is due to the definition of $\oplus$ at the bit level). It follows that

$$\begin{aligned} D_k(E_k(x)) &= D_k(x_1 \oplus k_1, \dots, x_\ell \oplus k_\ell) \\ &= ((x_1 \oplus k_1) \oplus k_1, \dots, (x_\ell \oplus k_\ell) \oplus k_\ell) \\ &= (x_1, \dots, x_\ell) \\ &= x \; . \end{aligned}$$

In order to illustrate the encryption process, let us suppose that $\ell = 4$, $x = (0,1,1,0)$, and $k = (1,1,0,1)$. Then $E_k(x) = (0 \oplus 1, 1 \oplus 1, 1 \oplus 0, 0 \oplus 1) = (1,0,1,1)$.

12.2.2. *Two dissimilar worlds*

As described in the Introduction, there are two principal classes of cryptosystems, distinguished by the management of the secret on E_k and D_k.

Conventional, *symmetric* or *secret-key cryptosystems* are the encryption schemes where nobody knows the key k used to communicate, except the legitimate correspondents, say Alice and Bob. In this context, k is called the *secret key*. Functions E_k and D_k are secret quantities shared by the two interlocutors. In order to use such a cryptosystem Alice and Bob need to choose the secret key together, or at least one of them determines then communicates it to the other. In short Alice and Bob must agree on the choice of the secret key *before* any encrypted communication. In order to make this choice, they must meet physically in a secure area or use a private network.

The one-time pad of example 12.1, and also DES, IDEA and AES (described later) are secret key cryptosystems.

The other main class of encryption processes is given by the so-called *asymmetric* or *public-key cryptosystems*. The key k and the decryption function D_k are secret quantities only known by the receiver of confidential messages, Bob, while the encryption function E_k (and not the key k) is published by Bob (on his web page for instance) so that everybody who wishes to communicate with him can use it. In this situation Bob is the unique individual able to decrypt messages $E_k(x)$ since D_k is its own secret. The different roles played by the public E_k on one side and the secrets k, D_k on the other side, justify the term "asymmetric" for such cryptosystems; obviously "public-key" comes from the existence of this public quantity E_k. The RSA algorithm belongs to this class of algorithms.

We emphasize the fact that both classes of cryptosystems are based on very different mathematical techniques: invertible functions over some algebraic structures, probability theory and statistical analysis usually occurred in conventional cryptography, while prime numbers, computability, and complexity theories are the main ingredients of the mathematical foundation for asymmetric encryption schemes.

12.2.3. *Functionalities provided by cryptographic devices*

The application of cryptographic tools is not restricted to the protection of information confidentiality. It is actually possible to define four primitive functionalities provided by encryption devices: confidentiality, authenticity, integrity, and non-repudiation. Each of them represents a means of defense against a particular kind of threat. These cryptographic characteristics are described below:

– *confidentiality* means that information, after encryption, loses all meaning for all people except the legitimate protagonists of a cryptographic communication. An enemy that intercepts the plaintext must be unable to decrypt it for confidentiality to be preserved;

– *integrity*: in every communication (encrypted or not) it is expected that the message be received with no modifications, exactly as it was sent. Moreover, if a received message is different from the transmitted message, then the receiver must be able to detect it. We say that "integrity of messages against modifications is ensured" if the preservation of these two properties is secured;

– *authenticity*: the mission assigned to authentication consists of guaranteeing that the received message comes from the entity (human, computer, process) that is supposed to send it. If an enemy – playing the role of Alice – sends a message to Bob, then the authenticity of the message must be questioned. Otherwise, Bob, believing the enemy is Alice, would send confidential information to her/him;

– *non-repudiation* is the means that avoid the receiver of a message to deny its transmission. This a fundamental protection, for instance, in the context of financial transactions.

In this chapter we only deal with integrity, which is the heart of cryptography, and we do not develop the other cryptographic notions.

12.2.4. *Cryptanalysis: the dark side of cryptology?*

In the world of cryptography two kinds of entities coexist: the legitimate players of a enciphered communication, Alice and Bob, and an adversary (also called cryptanalyst, enemy, opponent, attacker) who tries to discover the key used to encipher; if he succeeds in this attempt, then the enemy has "broken" the cryptosystem: the *cryptanalysis* is successful.

Cryptography and cryptanalysis form the two sides of cryptology, the *science of secrets*. In appearance, but only in appearance, the dark side of cryptology is cryptanalysis. This notion is also used to design systems to be invulnerable against some classes of cryptanalysis. Then it becomes essential to define models of the strength of an attacker so as to measure how strong the cryptosystem is. At this step *Kerckhoffs' principle* is often assumed. This assumption – defined by A. Kerckhoffs – means that the encryption algorithm is known by the enemy [KER 83a, KER 83b].

There exists a very basic cryptanalysis for every cryptosystem, called *brute-force attack*, which in theory should be able to break any encryption algorithm. It is not sophisticated at all as it consists of trying to decrypt a ciphertext with all possible keys until an understandable plaintext is obtained. An adversary will find the key after an average of $\frac{|\mathcal{K}|}{2}$ attempts[4].

The number $|\mathcal{K}|$ of possible keys is clearly a fundamental parameter to measure how strong a cryptosystem is, with respect to a brute-force attack. Modern cryptosystems with a size of at least 160 bits to encode a key are considered secure against this trivial attack because, even for very advanced computers, an exhaustive search in a set of 2^{159} seems to be impossible in practice. Usually, an attack is considered successful when it requires less time to get the key than a brute-force attack.

4. The quantity $|X|$ is the number of elements or *cardinal* of the finite set X.

Obviously more sophisticated cryptanalysis may be encountered. Their common goal is always to find the key used to encrypt messages. The most common types of attacks are classified by increasing order of adversary's power. The list is given below:

– *known-ciphertext attacks*. The adversary is assumed to only have access to a set of ciphertexts (from unknown plaintexts and a fixed unknown key);

– *known-plaintext attacks*. The enemy has samples of both the plaintext and its encrypted version (by a given and unknown key), the ciphertext, and is free to make use of them to reveal the key;

– *chosen-plaintext attacks*. This mode assumes that the attacker has the capability of choosing arbitrary plaintexts to be encrypted and can obtain the corresponding ciphertexts;

– *chosen-ciphertext attacks*. The opponent collects information by choosing one or several ciphertexts and obtaining their decryption under an unknown key.

This classification allows us to define several degrees of cryptographic resistance. For instance, it is possible to prove that the one-time pad is invulnerable with respect to a known-ciphertext attack while it is easily broken by a known-plaintext attack: let us assume that a plaintext m and its ciphertext $c = m \oplus k$ are known, then the key is immediately found by computing $m \oplus c = k$.

12.2.5. *General requirements to avoid vulnerabilities*

There are three theoretical models to measure the level of security of a cryptographic device. In 1949, Claude Shannon, founder of modern cryptography, gave the mathematical bases of contemporary cryptology in his famous article [SHA 49]. In this paper he introduced the two first criteria for a cryptosystem to be secure: *unconditional security* and *statistical security*.

A cryptosystem is said to provide *unconditional security* when any kind of knowledge of a ciphertext does not reveal any information about the corresponding plaintext. As an example we can prove that for Vernam's cryptosystem such a cryptographic property holds whenever a new randomly chosen key is used for each encryption. This very strong feature ensures invulnerability against every known-plaintext attack.

Nevertheless, one-time pad, as with all secret-key algorithms, involves a key exchange among the legitimate interlocutors, but to satisfy unconditional security they are forced to use a new key for each of their confidential communications. We easily see the limitation of such a process in practice. In order to get around it, Shannon defined another resistance criterion, namely *statistical security*, which is based on two more fundamental properties called *diffusion* and *confusion*.

Using the name *diffusion* Shannon defined the fact that every letter (or more generally symbol) of a ciphertext should be dependent of every letter of the corresponding plaintext and of the key. The goal of this is the following: two ciphertexts, where one of them is due to a modification – even a minimal one – of the plaintext or of the key, must be very different. Therefore, the ciphertexts are dependent on the initial condition (plaintext or key used). A slight difference at the input of a cryptosystem must produce a large difference in its output.

Confusion refers to making the relationship between the key and the ciphertext as complex and involved as possible in order to hide any statistical structures that could be used to discover information from the plaintext without knowledge of the key. For instance, statistics of natural languages must be destroyed during the encryption process so that they become inpractical for an adversary. We will observe soon that these two properties, diffusion and confusion, establish the architectural pattern of modern symmetric encryption algorithms.

The last approach to cryptographic security, called *computational security*, introduced by Whitfield Diffie and Martin Hellman in their joint work [DIF 76], only concerns public-key encryption schemes. Such an algorithm is said to provide *computational security* if the best known attack requires too many computations to be feasible in practice. In general we prove that breaking a cryptosystem is equivalent to solving a problem known to be difficult in the sense that the construction of an explicit solution is impossible in practice (but not in theory!). Notice also that a computationally secure scheme is not unconditionally secure.

12.3. Symmetric encryption schemes

This section is devoted to symmetric encryption schemes: the high-level design is presented at first, followed by famous instances of such schemes.

12.3.1. *The secret key*

Let us briefly recall how a secret-key algorithm is implemented. For a symmetrically ciphered communication, Alice and Bob, and no other entity, have the common secret key. Thus Alice encrypts her message with this key, and sends it to Bob, who can recover the original message from the ciphertext he received by using the key. Even if the cryptosystem used is known by everybody – in accordance with Kerckoffs' principle – an adversary cannot decrypt any intercepted confidential message as he does not possess the key.

The choice of the key by Alice and Bob is a tricky problem. Indeed, either they physically meet or one of them sends the key to the other using a communication network secured in some way, for instance a private optic fiber between two buildings, or by the use of a key-exchange protocol. This problem is not treated in this chapter. See [MEN 97], available on-line at http://www.cacr.math.uwaterloo.ca/hac/, for a good reference on key-exchange protocols.

12.3.2. *Iterated structures and block ciphers*

For the sake of efficiency, encryption processes are performed by computers. Thus, the messages (plain or cipher) are treated as blocks of bits (or bytes) built following an *iterated* architecture that allows a high level of confusion and diffusion. An internal *round function* T is used. It takes two arguments: a message m and a *secret-subkey* or *round subkey* k (both are blocks of bits). The subkey is produced from the secret key, called *master key*, by some *derivation algorithm*. Even though they are required for a symmetric encryption scheme, these algorithms are not treated in more detail in this chapter.

The round function is required to satisfy the following property to make decryption possible. With a fixed round subkey k, the function $T_k\colon m \mapsto T(m,k)$ must be invertible. This is actually the realization of the decryption rule in this particular context. The argument m is called *round plaintext* and $T(m,k)$ is the *round ciphertext*. The round function consists of a sequence of complex mathematical transformations in order to make its result $T(m,k)$ unintelligible. More precisely, T must implement confusion and diffusion requirements. In particular an output block of such a function must be dependent of an important number (at least half the number) of bits of plaintext and round subkey.

In order to confuse and diffuse, the round function is iterated some number r of times as follows. Let m be the message to encrypt. The following sequence of computations is done.

$$\begin{aligned} m_0 &= m; \\ m_{i+1} &= T_{k_{i+1}}(m_i) \text{ for } 0 \leq i \leq r-1 \end{aligned}$$

where k_i denotes the subkey related to the ith round. The ciphertext c, obtained as output of the last round, is given by formulae:

$$\begin{aligned} c &= m_r \\ &= T_{k_r}(m_{r-1}) \\ &= T_{k_r} \circ T_{k_{r-1}} \circ \cdots \circ T_{k_2} \circ T_{k_1}(m) \; . \end{aligned}$$

where "$\circ$" is the usual composition of functions. This iterated architecture turns out to be unavoidable to obtain convenient levels of confusion and diffusion in order to ensure statistical security. More precisely, iteration increases the diffusion.

Let us take a look at the deciphering process. Recall that for a given round subkey k, the function T_k is required to be invertible, which implies that there exists a map T_k^{-1} such that for any block x, $T_k^{-1}(T_k(x)) = x$. Decryption is performed by "reversing the time". More precisely, it is done by replacing the round function T_k by its inverse T_k^{-1}, and running the sequence of subkeys in the reverse order. Formally from the ciphertext c the plaintext m is obtained by:

$$\begin{aligned} c_0 &= c; \\ c_{i+1} &= T^{-1}_{\widehat{k}_{i+1}}(c_i) \text{ for } 0 \leq i \leq r-1 \end{aligned}$$

where $\widehat{k}_i$ denotes the subkey k_{r+1-i} related to the $r+1-i$th round so that:

$$\begin{aligned} \widehat{k}_1 &= k_r \\ \widehat{k}_2 &= k_{r-1} \\ &\dots \\ \widehat{k}_r &= k_1 \,. \end{aligned}$$

According to the invertibility of the round function (with a fixed round subkey), the final block c_r is clearly equal to the original plaintext m.

12.3.3. *Some famous algorithms: a short story of the evolution of mathematical techniques*

Most of the famous symmetric encryption schemes make use of an iterated structure with some possible minor modifications at the first and final rounds. Therefore such cryptosystems only differ from the point of view of the size of data (plain and ciphertext, secret key, subkey), of the number of rounds, of the internal round function, and the derivation algorithm used. In what follows three of the most renowned secret-key ciphers, namely DES, IDEA, and AES, are described, which use the evolution of mathematical constructions in these algorithms.

The 1970s: DES - data encryption standard

DES was designed by IBM during the 1970s, and became an encryption standard in 1977 for United States of America's official documents. Its status as a standard – for 5 years – was evaluated several times: the last time being in 1999.

In [FIP 99] the DES is completely described and in [FIP 87] its different operation modes are presented. This symmetric algorithm operates on 64-bit plaintexts, ciphertexts, and secret-keys. Actually, 8 bits from the key are parity bits: the 8th bit of each byte of the key takes the value such that the number of bits equal to 1 in this byte is an even number. A subkey (for a given round) is given by 48 bits of the master key – except parity bits – in some specific order. The ciphertext is obtained after 16 rounds.

Let us study the round function of the DES. It is formally defined as a *Feistel structure* or *Feistel scheme* named after the American cryptographer Horst Feistel [FEI 73]. In such a scheme, blocks have an even number 2ℓ of bits ($\ell = 32$ in the case of DES). The first ℓ consecutive bits of some block B are denoted, as a block, by L, while R is given by B's last ℓ bits in such a way that $B = (L, R)$. Let f be a function that takes two blocks as input, the first block having length ℓ. This function produces as output also a block of size ℓ. The round function T for the Feistel structure associated with f operates as follows: it takes $B = (L, R)$ and a round subkey k as entries, it flips L and R, and transforms L into $f(R, k) \oplus L$. This can be written in a mathematical form:

$$\begin{aligned} T(B,k) &= T((L,R),k) \\ &= (R, f(R,k) \oplus L). \end{aligned}$$

It can easily be checked that given any map f as above, the round function T, with a fixed round key k, is invertible. This is an essential property for the deciphering process in such a cryptosystem. Let us prove this property. We define the map $U_k(L, R) := (f(L, k) \oplus R, L)$ which will be shown to be the inverse of $T_k \colon (L, R) \mapsto T((L, R), k)$. Notice that if we denote by σ the permutation $\sigma(L, R) = (R, L)$, then $U_k = \sigma \circ T_k \circ \sigma$. Moreover $U_k(T_k(L, R)) = (L, R)$. Indeed let us define $L' = R$ and $R' = f(R, k) \oplus L$.

$$\begin{aligned} U_k(T_k(L,R)) &= U_k(R, f(R,k) \oplus L) \\ &= U_k(L', R') \\ &= (f(L',k) \oplus R', L') \\ &= (f(R,k) \oplus (f(R,k) \oplus L), R) \\ &= (L,R). \end{aligned}$$

As a result, such a Feistel scheme may be used as a round function in an iterated symmetric encryption algorithm.

In order to complete the description of the round function of the DES, a description of the function f used in this system is needed. This is an important function because confusion is based on it, while diffusion is obtained by the iterated structure itself. The function f takes as its first argument a block of size 32 (the 32 first or last bits of the block to encrypt), which is denoted by X. The second argument is a subkey, so here a block, say Y, of 48 bits. The result $f(X, Y)$ is a block of 32 bits (according to the specifications of Feistel structures). The function f carries out a computation in four steps:

1) X is transformed by a function E, that takes 32 bits in input and produces a block of 48 bits, in such a way that $E(X)$ consists of the bits of X in another order

where 16 of them are duplicated. More precisely the 48 bits of $E(X)$ are obtained by selecting the bits of X according to the order induced by the following table:

Function E

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Thus, as an example, the first 4 bits of $E(X)$ are the bits 32, 1, 2, and 3 of X whereas the last 3 bits are the bits 31, 32, and 1 of X;

2) the result $E(X) \oplus Y$ is then computed and written as a concatenation of eight subblocks, each of them consisting of 6 bits:

$$E(X) \oplus Y = B_1B_2B_3B_4B_5B_6B_7B_8$$

where for each $i \in \{1, \ldots, 8\}$, B_i has a length of 6 bits;

3) for each $i = 1, \ldots, 8$, B_i goes through a function S_i, called a *substitution* or an *S-box*. Such a box takes 6 bits in input and gives 4 bits as output. The result of this step is given by the concatenation of the $S_i(B_i)$, i.e. the block of 32 bits:

$$S = S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8).$$

Each S-box S_i is represented by a table with 4 rows and 16 columns. Its rows are indexed from the top to the bottom with integers from 0 to 3 and its columns from the left to the right by integers from 0 to 15. Each entry contains an integer between 0 and 15. For instance, the S-box S_1 is given by the following table:

S_1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Let us see the action of an S-box, say S_i, on a block B_i of 6 bits. The first and last bits of B_i are interpreted as a binary representation of an integer, say a, between 0 and 3. The four other bits represent a binary representation of an other integer, say b, between 0 and 15. The entry (a, b) of the table associated with S_i, i.e. the integer given at the intersection of the ath row and the bth column, may be written as a block of 4 bits in its binary representation (since, by definition, it is an integer between 0 and 15). This block is taken as the output of S_i, or in other terms, the value $S_i(B_i)$. For instance, let B_1 be the block 011011. The corresponding index for the row of S_1 is represented by 01 so it is equal to $0 \times 2^1 + 1 \times 2^0 = 1$ in decimal representation. The corresponding index for the column of S_1 is given by 1101, which is the binary representation of $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$. Therefore, $S_1(B_1)$ is the binary representation of the integer 5 given as the entry $(1, 13)$. Since 5 is represented by 0101, the result is $S_1(B_1) = 0101$.

These S-boxes are nonlinear in the sense that in general $S_i(B_i \oplus B_i') \neq S_i(B_i) \oplus S_i(B_i')$. They destroy the algebraic structure and, therefore, produce confusion for this cryptogram;

4) at the input of these eight S-boxes we have a block S of 32 bits. The last step of internal computations of f is a re-ordering of these bits using a permutation P. It is represented in the table below:

Permutation P

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

The output $P(S)$ is obtained from S by taking the 16th bit of S for the 1st bit of $P(S)$, the 7th bit of S for the 2nd bit of $P(S)$, etc. (at the end, the 25th bit of S is used as the 32nd bit of $P(S)$). $P(S)$ is taken as the result of $f(X, Y)$ for the round function.

In order to summarize this situation, to compute $f(X, Y)$, $B_1, \ldots, B_8$ are defined as blocks of 6 bits each by

$$B_1 B_2 \ldots B_8 = E(X) \oplus Y$$

then the block $f(X, Y)$ is defined by

$$f(X, Y) = P(S_1(B_1) S_2(B_2) \ldots S_8(B_8)).$$

The DES round function is now fully described. We are in position to conclude with the presentation of the encryption process by the DES algorithm. An initial step, before the 16 rounds, is applied to the block that represents the plaintext: it goes through a permutation IP, called the *initial permutation*, the operation of which is given by the following table.

Permutation IP

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	22	14	6
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Therefore, the permuted block has a bit number of 58 from the original block as its first bit, then bit 50 for its second, and so on. This initial step is followed by the 16 iterations of the round function. Finally if (L_{16}, R_{16}) denotes the 64-bits blocks produced at the 16th, and last, round, then the encryption process is ended by applying to $\sigma(L_{16}, R_{16}) = (R_{16}, L_{16})$ the inverse IP^{-1} of IP.

We are now in position to present this algorithm in a more compact way. Let k be the master key, and k_i be the subkey from round number i. Let $E_k^{\mathbf{DES}}$ be the DES encryption function. The ciphertext $E_k^{\mathbf{DES}}(m)$ of a plaintext m is computed as follows.

$$\begin{array}{lcl} (L_0, R_0) & = & IP(m); \\ (L_{i+1}, R_{i+1}) & = & T_{k_{i+1}}(L_i, R_i) \text{ for } i = 0, \ldots, 15; \\ E_k^{\mathbf{DES}}(m) & = & IP^{-1}(R_{16}, L_{16}). \end{array}$$

Using a more condensed notation:

$$E_k^{\mathbf{DES}}(m) = \left(IP^{-1} \circ \sigma \circ T_{k_{16}} \circ \cdots \circ T_{k_1} \circ IP\right)(m) \ .$$

Notice that the final permutation is not applied to (L_{16}, R_{16}) but to $\sigma(L_{16}, R_{16})$, i.e. (R_{16}, L_{16}). Since this permutation, IP^{-1}, is the inverse of IP, in order to perform decryption the same algorithm is applied on $E_k^{\mathbf{DES}}(m)$, subkeys being used in a reverse order from k_{16} to k_1. In other terms, the decryption function is defined by

$$D_k^{\mathbf{DES}}(c) = (IP^{-1} \circ \sigma \circ T_{k_1} \circ \cdots \circ T_{k_{16}} \circ IP)(c) \ .$$

In order to check the decryption rule, we only need to notice that $T_k^{-1} = \sigma \circ T_k \circ \sigma$, and for every (L, R), $(\sigma \circ \sigma)(L, R) = (L, R)$. Therefore,

$$\begin{array}{l} D_k^{\mathbf{DES}}(E_k^{\mathbf{DES}}(m)) = \\ (IP^{-1} \circ \sigma \circ T_{k_1} \circ \cdots \circ T_{k_{16}} \circ IP)(E_k^{\mathbf{DES}}(m)) = \\ IP^{-1} \circ (\sigma \circ T_{k_1} \circ \sigma) \circ \cdots \circ (\sigma \circ T_{k_{16}} \circ \sigma) \circ \sigma \circ IP(E_k^{\mathbf{DES}}(m)) = \\ IP^{-1} \circ T_{k_1}^{-1} \circ \cdots \circ T_{k_{16}}^{-1} \circ \sigma \circ IP(E_k^{\mathbf{DES}}(m)) = \\ IP^{-1} \circ T_{k_1}^{-1} \circ \cdots \circ T_{k_{16}}^{-1} \circ \sigma \circ IP \circ IP^{-1} \circ \sigma \circ T_{k_{16}} \circ \cdots \circ T_{k_1} \circ IP(m) = m \end{array}$$

(eliminating consecutive compositions of a map and its inverse).

Document [FIP 99] also contains the description of another algorithm for symmetric encryption, TDEA (for Triple Data Encryption Algorithm), called *triple DES*. It is defined as an iteration of the original DES. Let $k^{(1)}$, $k^{(2)}$ and $k^{(3)}$ be three master keys subject to particular independence properties (given in [FIP 99]). Let m be a 64 bits long block to encode:

1) encryption algorithm: block m is transformed into a new block c (64 bits) as follows:

$$c = E^{\mathbf{DES}}_{k^{(3)}}(D^{\mathbf{DES}}_{k^{(2)}}(E^{\mathbf{DES}}_{k^{(1)}}(m)));$$

2) decryption algorithm: m is recovered from the ciphertext c by computing:

$$m = D^{\mathbf{DES}}_{k^{(1)}}(E^{\mathbf{DES}}_{k^{(2)}}(D^{\mathbf{DES}}_{k^{(3)}}(c))) \ .$$

The 1990s: IDEA - International Data Encryption Algorithm

The IDEA, invented by Xuejia Lai and James L. Massey, is described in [LAI 90] and [LAI 92].

IDEA was explictly designed to fulfill confusion and diffusion requirements. Similar to DES, it is based on an iterated structure. However, the method used to produce invertible functions – in order to make possible the decryption process – is not based on Feistel structures. IDEA round function relies on more involved mathematical structures, namely the *groups*. An *internal composition law*, denoted by $*$, on a set E is a function that associates an ordered pair (x, y) of members of E with some z that belongs to E: we denote this z by $x * y$. A *group* is then defined as a non-empty set G together with an internal composition law that satisfies the following axioms:

1) associativity: for every x, y, z in G, $x * (y * z) = (x * y) * z$;

2) neutral element: there is some $e \in G$ such that for every $x \in G$, $x * e = e * x = x$;

3) inversion: for every $x \in G$, there is a unique $y_x \in G$ such that $x * y_x = y_x * x = e$. This element y_x is usually denoted by x^{-1}.

For instance if p is a *prime* number – that is a positive integer > 1 with 1 and the number itself as only divisors (such that 2, 3, 5, 7, 11, etc.) – then modulo p multiplication of positive integers is an internal composition group law on the set $\{1, 2, \cdots, p-1\}$. Similarly for every positive integer n, the set $\{0, \cdots, n-1\}$ becomes a group under modulo n addition. Finally, the set of all blocks of n bits with bit-wise modulo 2 sum, that is XOR, is another example of a group. IDEA is precisely based on these three algebraic structures.

In order to describe the round function of IDEA, the following notations will be used. Let n be an integer so that $2^{2^n} + 1$ is a prime number (for instance $n = 1$ or $n = 2$ or $n = 16$).

– As usual the symbol "$\oplus$" is used to denote XOR operation between two blocks of 2^n. For instance with $n = 2$, $(0, 1, 1, 0) \oplus (1, 1, 0, 1) = (1, 0, 1, 1)$;

– each 2^n-bit long block can be identified with a unique integer between 0 and $2^{2^n} - 1$ written in binary representation. More generally, let us assume given a ℓ-bit block $(x_{\ell-1}, x_{\ell-2}, \cdots, x_1, x_0)$, $x_i \in \{0, 1\}$. It represents the integer $x = \sum_{i=0}^{\ell-1} x_i 2^i$, and satisfies $0 \leq x \leq 2^\ell - 1$. It is thereby possible to compute a modulo 2^{2^n} addition under this identification (take $\ell = 2^n$). This operation is denoted by "$\boxplus$". For $n = 2$ so that $2^{2^n} = 16$, $(0, 1, 1, 0)$ represents the integer 6, and $(1, 1, 0, 1)$ the integer 13. Addition modulo 16 of 6 and 13 is, in binary notation, is equal to $(0, 0, 1, 1)$. Therefore $(0, 1, 1, 0) \boxplus (1, 1, 0, 1) = (0, 0, 1, 1)$;

– each 2^n-bit long block, such that at least one of its bits is not zero, represents a unique integer between 1 and $2^{2^n} - 1$. The block, given by 2^n bits equal to zero, is declared to represent the integer 2^{2^n}. Since $2^{2^n} + 1$ is assumed to be prime, the set $\{1, 2, \ldots, 2^{2^n}\}$, under modulo $2^{2^n} + 1$ multiplication of integers, is a group. According to this identification between blocks and integers, we can apply this product, denoted by "$\odot$", to any two blocks (each of them composed of 2^n bits). For instance, $(0, 1, 1, 0) \odot (1, 1, 0, 1) = (1, 0, 1, 0)$ since 6×13 is equal to 10 modulo $2^4 + 1 = 17$, and 10 is represented as $(1, 0, 1, 0)$ in base two.

Basic components of IDEA being known, it is possible to describe the round function. IDEA handles blocks of 64 bits for plain and ciphertexts, and uses a master key of size 128 bits. The derivation algorithm produces at each round, from a given master key, subkeys of 96 bits. The block m_{i-1}, produced at the $(i-1)$th round, is used as the input of the round function for the ith round. It is divided into four blocks, each of 16 bits, while the ith subkey is divided into six blocks of 16 bits, so that $m_{i-1} = m_{i-1}^1\, m_{i-1}^2\, m_{i-1}^3\, m_{i-1}^4$ and $k_i = k_i^1\, k_i^2\, k_i^3\, k_i^4\, k_i^5\, k_i^6$ where m_{i-1}^j and k_i^l are blocks of 16 bits for each $j = 1, 2, 3, 4$ and $l = 1, 2, 3, 4, 5, 6$. Notice that 16 satisfies the requirement that $2^{16} + 1 = 65537$ is a prime number. As a consequence it is possible to use the three group laws previously introduced on blocks of 16 bits. The round function is based on a particular operation, denoted by MA, and called *multiplication-addition* or *MA-structure*, that takes four blocks x_1, x_2, y_1, y_2, each of 16 bits, in input and produces two blocks, $MA_1(x_1, x_2, y_1, y_2)$ and $MA_2(x_1, x_2, y_1, y_2)$, also 16 bits long. Mathematical relations between inputs and outputs of MA are the following:

$$\begin{array}{lcl} MA(x_1, x_2, y_1, y_2) & = & MA_1(x_1, x_2, y_1, y_2)\; MA_2(x_1, x_2, y_1, y_2) \\ MA_1(x_1, x_2, y_1, y_2) & = & MA_2(x_1, x_2, y_1, y_2) \boxplus (x_1 \odot y_1) \\ MA_2(x_1, x_2, y_1, y_2) & = & ((x_1 \odot y_1) \boxplus x_2) \odot y_2 \end{array}$$

where the second member of the first equality represents the concatenation of the blocks $MA_1(x_1, x_2, y_1, y_2)$ and $MA_2(x_1, x_2, y_1, y_2)$. The MA-structure is therefore composed of sophisticated and involved use of two of the three group operations, multiplication $\odot$ modulo $2^{16} + 1$ and addition $\boxplus$ modulo 2^{16}. The MA-structure plays the same role as the function f from DES in fulfillment of confusion and diffusion, but unlike the latter, it is invertible whenever the inputs y_1 and y_2 are fixed. Indeed from the knowledge of the outputs $z_1 = MA_1(x_1, x_2, y_1, y_2)$, $z_2 = MA_2(x_1, x_2, y_1, y_2)$ of the MA-structure together with y_1, y_2, it is possible to recover x_1, x_2. The equation $z_1 = MA_2(x_1, x_2, y_1, y_2) \boxplus (x_1 \odot y_1)$ leads to the value of x_1. In fact, let us denote by a^{-1} (respectively $-a$) the inverse of a with respect to the operation $\odot$ (respectively $\boxplus$).

$$\begin{array}{rlcl} & z_1 & = & z_2 \boxplus (x_1 \odot y_1) \\ \Leftrightarrow & -z_2 \boxplus z_1 & = & x_1 \odot y_1 \\ \Leftrightarrow & (-z_2 \boxplus z_1) \odot y_1^{-1} & = & x_1. \end{array}$$

Then, injecting this value for x_1 into the equation $z_2 = ((x_1 \odot y_1) \boxplus x_2) \odot y_2$, recovers the value of x_2. Indeed,

$$\begin{array}{rlcl} & z_2 & = & ((x_1 \odot y_1) \boxplus x_2) \odot y_2 \\ \Leftrightarrow & z_2 \odot y_2^{-1} & = & (x_1 \odot y_1) \boxplus x_2 \\ \Leftrightarrow & -(x_1 \odot y_1) \boxplus (z_2 \odot y_2^{-1}) & = & x_2 \\ \Leftrightarrow & -(((-z_2 \boxplus z_1) \odot y_1^{-1}) \odot y_1) \boxplus (z_2 \odot y_2^{-1}) & = & x_2. \end{array}$$

IDEA does not use any Feistel structure, invertible by construction, but we will see later the round function of IDEA to be also invertible. Surprisingly, invertibility of the MA-structure does not play any role in the deciphering process.

Let us precisely examine any round of IDEA. The ith round produces a block c_i of 64 bits divided into four blocks (16 bits each of them), which we denote by c_i^1, c_i^2, c_i^3 and c_i^4 such that $c_i = c_i^1\ c_i^2\ c_i^3\ c_i^4$. From a purely mathematical point of view, an IDEA round is given by the following formulae.

$$\begin{array}{rcl} c_i^1 & = & MA_2 \oplus (m_{i-1}^3 \boxplus k_i^3); \\ c_i^2 & = & MA_1 \oplus (m_{i-1}^4 \boxplus k_i^4); \\ c_i^3 & = & MA_2 \oplus (m_{i-1}^1 \odot k_i^1); \\ c_i^4 & = & MA_1 \oplus (m_{i-1}^2 \odot k_i^2) \end{array} \tag{12.1}$$

where we define

$$\begin{array}{rcl} MA_1 & = & MA_1((m_{i-1}^1 \odot k_i^1) \oplus (m_{i-1}^3 \boxplus k_i^3), (m_{i-1}^2 \odot k_i^2) \oplus (m_{i-1}^4 \boxplus k_i^4), k_5^i, k_6^i); \\ MA_2 & = & MA_2((m_{i-1}^1 \odot k_i^1) \oplus (m_{i-1}^3 \boxplus k_i^3), (m_{i-1}^2 \odot k_i^2) \oplus (m_{i-1}^4 \boxplus k_i^4), k_i^5, k_i^6). \end{array} \tag{12.2}$$

The IDEA enciphering process is given as a sequence of eight rounds for which the output c_i from the ith round is chosen as input for the following round. The ciphertext corresponding to the plaintext $m = m_0$ is not the block c_8, output of the eighth round. Indeed there is a final step: the ciphertext $c_9 = c_9^1\ c_9^2\ c_9^3\ c_9^4$ is computed by

$$\begin{array}{rcl} c_9^1 & = & c_8^1 \odot k_9^1; \\ c_9^2 & = & c_8^2 \odot k_9^2; \\ c_9^3 & = & c_8^3 \boxplus k_9^3; \\ c_9^4 & = & c_8^4 \boxplus k_9^4 \end{array} \tag{12.3}$$

where $k_9 = k_9^1\, k_9^2\, k_9^3\, k_9^4$ is a subkey of 64 bits (each of the k_9^i being composed of 16 bits), which also comes from the derivation algorithm applied to the master key.

Let us review the decryption process. First of all, let us explore how to recover the inputs m_{i-1}^j for $j = 1, \dots, 4$ of round number i (for $1 \le i \le 8$) from the outputs c_i^j and subkeys k_i^l ($l = 1, \dots, 6$). Recall that the inverse of a block x under $\oplus$ operation is x itself. In particular, $x \oplus x$ is equal to the block with all bits equal to zero, neutral element for $\oplus$. From the definitions of c_i^j given by equations (12.1), the following result can be checked.

$$\begin{array}{rrcl} & c_i^1 \oplus c_i^3 & = & MA_2 \oplus (m_{i-1}^3 \boxplus k_i^3) \oplus MA_2 \oplus (m_{i-1}^1 \odot k_i^1) \\ \Leftrightarrow & c_i^1 \oplus c_i^3 & = & (m_{i-1}^3 \boxplus k_i^3) \oplus (m_{i-1}^1 \odot k_i^1). \end{array}$$

Similarly

$$\begin{array}{rrcl} & c_i^2 \oplus c_i^4 & = & MA_1 \oplus (m_{i-1}^4 \boxplus k_i^4) \oplus MA_1 \oplus (m_{i-1}^2 \odot k_i^2) \\ \Leftrightarrow & c_i^2 \oplus c_i^4 & = & (m_{i-1}^4 \boxplus k_i^4) \oplus (m_{i-1}^2 \odot k_i^2). \end{array}$$

Then notice that $(c_i^1 \oplus c_i^3)$ (respectively $c_i^2 \oplus c_i^4$) is the first (respectively the second) argument of the MA function according to equations (12.2). From this we see that under knowledge of all c_i^j and k_i^5, k_i^6, MA_1 and MA_2 can be computed. Finally using formulae (12.1) inputs from round number i, namely m_{i-1}^j, can be deduced since subkeys k_i^l are also known. For instance,

$$\begin{array}{rlcl} & c_i^1 & = & MA_2 \oplus (m_{i-1}^3 \boxplus k_i^3) \\ \Leftrightarrow & c_i^1 \oplus MA_2 & = & m_{i-1}^3 \boxplus k_i^3 \\ \Leftrightarrow & (c_i^1 \oplus MA_2) \boxplus (-k_i^3) & = & m_{i-1}^3. \end{array}$$

From c_9^j and subkeys $k_9^1, k_9^2, k_9^3, k_9^4$, $c_8 = c_8^1\, c_8^2\, c_8^3\, c_8^4$ is recovered: the equations (12.3) are used. It can be easily shown that:

$$\begin{array}{rcl} c_9^1 \odot (k_9^1)^{-1} & = & c_8^1; \\ c_9^2 \odot (k_9^2)^{-1} & = & c_8^2; \\ c_9^3 \boxplus (-k_9^3) & = & c_8^3; \\ c_9^4 \boxplus (-k_9^4) & = & c_8^4. \end{array}$$

As previously claimed, invertibility of the MA-structure is not involved in the decryption process. In other terms, if f is any function that takes four blocks of 16 bits as input and produces two blocks of 16 bits as outputs, the encryption algorithm obtained, after substitution of the MA-structure by f in the IDEA algorithm, remains invertible and allows decryption process. So after all, what is the role of this function ? Actually diffusion requirement is based on MA. Indeed, each output subblock of MA depends on all input subblocks, in such a way that it ensures diffusion in a number of rounds less than DES.

Confusion is obtained by the mixed use of the three group operations, which, in a specific sense, are mutually incompatible, and thus allow the algebraic structures used to be hidden. As an example, the following properties can be listed. Let $\#$ and $\bigstar$ be two different operations from $\oplus$, $\boxplus$ and $\odot$ (for instance $\#$ is $\oplus$ and $\bigstar$ is $\boxplus$, or $\#$ is $\boxplus$ and $\bigstar$ is $\odot$).

1) No such pairs of operations $\#$, $\bigstar$ satisfy distributivity from one over the other, that is there are at least three blocks x, y, z, each of 16 bits, such that:

$$x\#(y\bigstar z) \neq (x\#y)\bigstar(x\#z);$$

2) no such pairs of operations $\#$, $\bigstar$ satisfy associativity, that is there are at least three blocks x, y, z of 16 bits such that:

$$x\#(y\bigstar z) \neq (x\#y)\bigstar z.$$

Presently: AES - Advanced Encryption Standard

On September 2, 1997, the NIST (National Institute of Standards and Technology) launched a call for proposals about a new cryptosystem to replace DES as a standard. The requirements were the following: a symmetric encryption algorithm, called the AES, supporting blocks of size 128 bits, and keys of lengths 128, 192, and 256 bits. On August 20, 1998, the NIST announced the application of 15 algorithms from twelve countries. A year later, after a detailed review of candidates, the NIST retained only five proposals, namely MARS, RC6, Rijndael, Serpent, and Twofish. A second and last round was led by the NIST with help from the worldwide cryptographic community in order to select the winner. Dramatically no attacks were able to break any of the five last candidates [NEC 01]. Nevertheless, in 2000, other criteria, as algorithmic complexity or implementation characteristics, were applied to select Joan Daemen and Vincent Rijmen's Rinjdael algorithm as the new encryption standard AES. The official document [FIP 01], dated from November 26, 2001, approved AES as a cryptographic protection of sensitive electronic data (unclassified) of Federal agencies and departments of the US government. In the same document are presented the full AES specifications in detail.

AES supports 128, 192, or 256 bits long blocks as key formats, and plaintexts, ciphertexts have 128 bits. The choice of key length depends on the level of protection needed by the communications (the longer they are, the greater the security); for instance, in a note from the American federal government [NSA 03], the National Security Agency (NSA) recommends the use of 192 or 256 bit keys for top-secret documents. Notice that the original Rinjdael was conceived to also manipulate other lengths for blocks and keys, but these were not retained in the final AES version. Similar to its predecessor DES, AES operates on a certain number of rounds that depends on the length of the keys in a way described in the following table.

Keys	Number of rounds
128	10
192	12
256	14

Contrary to DES or IDEA, plaintexts and ciphertexts are not treated as blocks of bits, but as matrices of bytes, called *states*: the input of a round is a 4×4 matrix (four rows, and four columns) of bytes entries. Thus a state is represented as the following matrix

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

where $a_{i,j}$ is a byte. Such a state represents the block

$$a_{0,0}\, a_{1,0}\, a_{2,0}\, a_{3,0}\, a_{0,1}\, a_{1,1}\, a_{2,1}\, a_{3,1}\, \ldots\, a_{0,3}\, a_{1,3}\, a_{2,3}\, a_{3,3}.$$

Whatever the size chosen for the secret key, round subkeys are also represented by such 4×4 arrays of bytes (so they contain 128 bits).

In order to describe the AES round function in detail, some mathematical notions are required. We use the same notations as in the official document [DAE 99] available at http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf.

AES uses operations that are defined on a finite field. A (commutative) *field* $\mathbb{K}$ is a set with at least two distinct elements, 0 and 1, and equipped with two internal composition laws, $+$ and $\times$ (to denote the second law, juxtaposition will also be used as the usual multiplication), such that

1) $\mathbb{K}$ with $+$ is a group with 0 for its neutral element (the inverse of x under this law will be denoted by $-x$, that is $x + (-x) = (-x) + x = 0$);

2) *addition* $+$ is commutative: for every x, y in $\mathbb{K}$, $x + y = y + x$;

3) the set $\mathbb{K}^*$ of elements of $\mathbb{K}$ distinct from 0 is a group for *multiplication* $\times$ and its neutral element is 1 (the inverse of a non zero x in $\mathbb{K}$ for multiplication is denoted by x^{-1} and thus $xx^{-1} = x^{-1}x = 1$);

4) 0 is an absorbing element for multiplication: for every $x \in \mathbb{K}$, $x0 = 0x = 0$;

5) multiplication is commutative: $xy = yx$ for every $x, y \in \mathbb{K}$;

6) multiplication distributes over addition: $x(y + z) = xy + xz$ for all x, y, z in $\mathbb{K}$.

Among the class of all fields some have an infinite cardinality, while others, more interesting from an implementation point of view, have only finitely many elements. They are called the *finite fields*. A byte may be represented as an element of the finite field $\mathsf{GF}(2^8)$ with $2^8 = 256$ elements ("GF" stands for "Galois Field"). Thus a state may be seen as a 4×4 array with entries in this field.

The AES round function has four basic components. Each of them is invertible, which is an important difference compared to DES and IDEA. Indeed, the AES round function is invertible as the composite of invertible maps while the corresponding property in DES or IDEA is not based upon invertible internal components. The first three components are independent of the round subkey, while the fourth is just entry-by-entry addition of bytes from the current state and those of the subkey. Each of these operations acts in a specific way on a state and promotes diffusion. A round thus consists of four stages as follows:

1) the function `ByteSub` (for *Byte Substitution*) is applied to a matrix A, which represents the state at the input of the round. This map acts independently on each entry of A via an invertible transformation $\mathtt{S_{RD}} : \mathsf{GF}(2^8) \to \mathsf{GF}(2^8)$. This function is essentially defined using

$$\mathsf{inv} : x \mapsto \begin{cases} 0 & \text{if } x = 0, \\ x^{-1} & \text{if } x \neq 0 \end{cases}$$

where x is a byte seen as en element of $\mathsf{GF}(2^8)$. More precisely $\mathsf{S_{RD}} = \lambda \circ \mathsf{inv}$ where λ is an *affine* and invertible transformation of bytes; the term "affine" means that there exists a function α such that for all bytes x, y, $\alpha(x+y) = \alpha(x) + \alpha(y)$ (α is said to be *linear*), and a fixed byte β, interpreted as en element of $\mathsf{GF}(2^8)$, such that $\lambda(x) = \alpha(x) + \beta$. The linear map α also is invertible, and $\lambda^{-1}(x) = \alpha^{-1}(x - \beta)$. Therefore the inverse of $\mathsf{S_{RD}}$ is obtained as ${\mathsf{S_{RD}}}^{-1} = \mathsf{inv} \circ \lambda^{-1}$ since, as easily checked, inv is its own inverse.

Graphically, ByteSub may be described as follows:

$$\mathtt{ByteSub}\left[\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}\right] = \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}$$

where each $b_{i,j} = \mathsf{S_{RD}}(a_{i,j})$.
Theoretical results assert that this function provides a very good level of confusion [DAE 02]; in particular, $\mathsf{S_{RD}}$, like DES S-boxes, is not linear since in general, $\mathsf{S_{RD}}(x+y) \neq \mathsf{S_{RD}}(x) + \mathsf{S_{RD}}(y)$. This first step "destroys" the group structure of $\mathsf{GF}(2^8)$ under addition. It is therefore an essential part in achieving the confusion within the cryptosystem. We also notice that ByteSub is invertible: in order to recover an input state A from $B = \mathsf{ByteSub}(A)$, it is sufficient to apply ${\mathsf{S_{RD}}}^{-1}$ on each entry of B; in other terms, the transformation obtained from ByteSub after substitution of $\mathsf{S_{RD}}$ by its inverse ${\mathsf{S_{RD}}}^{-1}$ is the inverse of ByteSub;

2) a shift, from left to right, is applied on the rows of the matrix $B = \mathsf{ByteSub}(A)$, output of ByteSub. These are cyclic shifts: for instance applying shift operation two consecutive times on the row

x_1	x_2	x_3	x_4

gives

x_3	x_4	x_1	x_2

This operation on the rows of the current state is called ShiftRow. The way the shift operates on a row depends on the row index. The first row is not shifted, while the second is shifted one step to the right, the third two steps, and the fourth, three steps. Graphically ShiftRow acts on a matrix B with entries $b_{i,j}$ as follows.

$$\texttt{ShiftRow}\left[\begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}\right] = \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,1} & b_{1,2} & b_{1,3} & b_{1,0} \\ b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ b_{3,3} & b_{3,0} & b_{3,1} & b_{3,2} \end{pmatrix}.$$

Like ByteSub, the ShiftRow function is invertible. In order to recover the output matrix B of ByteSub it is sufficient to apply ShiftRow modified by the value of the applied shifts: the first row is not shifted, the second is now shifted three steps to the left, the third, two steps, and the fourth, only one step. This transformation aims to promote diffusion in the cryptosystem;

3) the third step of the round, called MixColumn, operates at the column level of the current state. It is a matrix multiplication of each column of the state by the same invertible 4×4 matrix M. Let $C = \mathsf{ShiftRow}(B)$ be the current state, result of ShiftRow, with entries denoted by $c_{i,j}$ ($i = 0, \ldots, 3$, $j = 0, \ldots, 3$). Let C_0, C_1, C_2, C_3 be the four columns of C, from left to right, in such a way C can be seen as concatenation, $[C_0 \mid C_1 \mid C_2 \mid C_3]$, of its columns. It follows that column number j (for $j = 0, \ldots, 3$) has the following form:

$$C_j = \begin{pmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{pmatrix}.$$

The multiplication of the column C_j by the matrix M (its entries, $m_{i,j}$, belong to the field $\mathsf{GF}(2^8)$) gives an other column $D_j = \begin{pmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{pmatrix}$. So $D_j = \mathsf{M}C_j$, and in matrix representation:

$$D_j = \mathsf{M}C_j \Leftrightarrow \begin{pmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{pmatrix} = \begin{pmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{pmatrix} \begin{pmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{pmatrix}.$$

Matrix multiplications lead to the following result concerning the values of $d_{i,j}$ for $i = 0, \ldots, 3$.

$$d_{i,j} = m_{i,0}c_{0,j} + m_{i,1}c_{1,j} + m_{i,2}c_{2,j} + m_{i,3}c_{3,j}.$$

The matrix multiplication of the columns of C produces four new columns D_0, D_1, D_2, D_3 that can be concatenated to build a 4×4 matrix D as $D = [D_0 \mid D_1 \mid D_2 \mid D_3]$. In short, MixColumn is formally defined as

$$D = \mathsf{MixColumn}(C) = [\mathsf{M}C_0 \mid \mathsf{M}C_1 \mid \mathsf{M}C_2 \mid \mathsf{M}C_3] .$$

The fact that M is an invertible matrix is essential for the invertibility of MixColumn. For the matrix M to be invertible it means that there is another 4×4 matrix, denoted by M^{-1}, such that for every 1×4 matrix X (that is a matrix of one row and four columns, similar to a column C_j for instance)

$$\mathsf{M}^{-1}(\mathsf{M}X) = X. \tag{12.4}$$

In other terms, if M^{-1} is multiplied by the column that results from the product $\mathsf{M}X$, then X is obtained. Using this property we can prove MixColumn to be invertible. Indeed, let $\mathsf{MixColumn}^{-1}$ be the operation obtained from MixColumn after replacement of M by its inverse M^{-1}. Let us check that by applying $\mathsf{MixColumn}^{-1}$ to the state $D = \mathsf{MixColumn}(C)$, C is recovered.

$$\begin{aligned}
\mathsf{MixColumn}^{-1}(D) &= [\mathsf{M}^{-1}D_0 \mid \mathsf{M}^{-1}D_1 \mid \mathsf{M}^{-1}D_2 \mid \mathsf{M}^{-1}D_3] \\
&\quad \text{(because } D = [D_0 \mid D_1 \mid D_2 \mid D_3]\text{)} \\
&= [\mathsf{M}^{-1}(\mathsf{M}C_0) \mid \mathsf{M}^{-1}(\mathsf{M}C_1) \mid \mathsf{M}^{-1}(\mathsf{M}C_2) \mid \mathsf{M}^{-1}(\mathsf{M}C_3)] \\
&\quad \text{(since } D = \mathsf{MixColumn}(C)\text{)} \\
&= [C_0 \mid C_1 \mid C_2 \mid C_3] \\
&\quad \text{(by property (12.4)).}
\end{aligned}$$

4) The fourth and final step of an AES round is given by an addition of the MixColumn result D and the round subkey k, seen under the form of a 4×4 matrix of bytes. Therefore the following holds:

$$\begin{aligned}
D + k &= \begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{pmatrix} + \begin{pmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{pmatrix} \\
&= \begin{pmatrix} d_{0,0}+k_{0,0} & d_{0,1}+k_{0,1} & d_{0,2}+k_{0,2} & d_{0,3}+k_{0,3} \\ d_{1,0}+k_{1,0} & d_{1,1}+k_{1,1} & d_{1,2}+k_{1,2} & d_{1,3}+k_{1,3} \\ d_{2,0}+k_{2,0} & d_{2,1}+k_{2,1} & d_{2,2}+k_{2,2} & d_{2,3}+k_{2,3} \\ d_{3,0}+k_{3,0} & d_{3,1}+k_{3,1} & d_{3,2}+k_{3,2} & d_{3,3}+k_{3,3} \end{pmatrix}.
\end{aligned}$$

So the result of $D + k$ is a 4×4 matrix E with byte entries $e_{i,j}$ given by

$$e_{i,j} = d_{i,j} + k_{i,j}.$$

This operation of round subkey addition is also invertible because

$$E = D + k \Leftrightarrow D = E + k$$

since addition in $\mathsf{GF}(2^8)$ is nothing else than a usual XOR, and thus $(D+k)+k = D$.

12.4. Prime numbers and public key cryptography

This section is dedicated to the notion of public key cryptography (through one of its famous instance: the RSA) and its relation with arithmetic and, more precisely, prime numbers.

12.4.1. *Introduction*

Many cryptosystems use prime numbers. Most of them belong to the class of asymmetric encryption algorithms. In this section is briefly described one of the most famous, namely the RSA cryptosystem, and then the construction of "cryptographic" prime numbers will be reviewed.

The very principle of public-key encryption was introduced by Diffie and Hellman [DIF 76] in 1976, but the authors were unable to provide an example of such algorithms. In asymmetric cryptography, two kinds of keys are involved: a *public key* and a *private key*. The first is used for encryption, while the second enters the scene during the decryption process.

Let us review a communication between Alice and Bob, made confidential using a public-key algorithm. Let us assume that Alice wishes to send to Bob a confidential message. Alice finds Bob's public key k_B^{pub} which, as public, is known to everybody. Then, she encrypts her message, with this key, and sends it to Bob. Bob, who is the only one to know his private key k_B^{priv}, recovers the plaintext as a result of a decryption algorithm. Notice that for Bob to send a confidential message to Alice, he must use Alice's public key k_A^{pub} who will recover Bob's message with her own private key k_A^{priv}. In such a scenario, everybody has access to public keys, while private keys are kept secret.

Security of public-key encryption process is provided by infeasibility to solve some mathematical problem in a practical way, that is without requiring too much memory or time. In particular, the RSA algorithm is based on the problem of integer factorization or prime factorization of a number into its prime factors.

12.4.2. *The RSA*

One of the first public-key algorithms, the RSA, was developed by Rivest, Shamir and Adleman [RIV 78].

Given two prime numbers p, q, it is easy to compute their product $n = pq$, even if both of them a very large. However, it is difficult to *factorize* n, that is to recover its prime divisors when p, q are large enough (one thousand of decimal digits). The security of RSA is based on this assumption.

Keys generation step

Let p, q be two distinct prime numbers. Let $n = pq$ be the *RSA modulus*. An integer $e < (p-1)(q-1)$, coprime[5] to $(p-1)(q-1)$, is chosen. It is called the *encryption exponent*. Its inverse, d, modulo $(p-1)(q-1)$ is computed: d is the unique positive integer $x < (p-1)(q-1)$ such that

$$ex = a(p-1)(q-1) + 1 \tag{12.5}$$

for some integer a. The number d is called the *decryption exponent*. The ordered pair (n, e) is the public key, while (p, q, d) is the private key.

Enciphering step

A plaintext m is a non-negative integer $< n$. Its corresponding ciphertext is the non-negative integer c, $0 \leq c < n$, given by

$$c = m^e \ mod \ n \ .$$

Notice that everybody is able to compute c from m, because n, e are given in the public key.

Deciphering step

In order to recover the plaintext m from the ciphertext c, computed as above, it is sufficient to compute

$$m = c^d \ mod \ n \ .$$

We notice that only a person with knowledge of d, part of the private key, is able to compute c^d.

5. Two integers a, b are coprime if, and only if, their greatest common divisor is 1.

Let us check the equality above to be true:

$$\begin{aligned} c^d \, mod \, n &= (m^d)^e \, mod \, n \\ &= m^{ed} \, mod \, n \\ &= mm^{a(p-1)(q-1)} \, mod \, n \text{ (according to (12.5))} \\ &= m(m^{(p-1)(q-1)})^a \, mod \, n \\ &= m \, mod \, n \\ &= m \end{aligned}$$

since it can be checked that $m^{(p-1)(q-1)} = m \, mod \, n = m$.

12.4.3. *Primality and pseudo-primality*

Many asymmetric encryption schemes use prime numbers. It follows that the ways to test a number to be prime, or to construct prime numbers are very important issues in this context. The end of this chapter is devoted to a short presentation of some of these methods.

A problem must be solved in order to use a given prime number p in a public-key encryption: one must be sure that p is prime. This problem is obviously solved for small numbers, but it requires some techniques, called *primality tests*, for cryptographic relevant numbers (i.e. very large numbers).

In order to know whether a given number is a prime number or not, it is possible to factorize this number. The Sieve of Eratosthenes is one of the oldest methods to achieve this. Nevertheless, it becomes unusable when the number of decimal digits is large. The best factorization algorithms are able to compute the prime factors of numbers with an order of 200 decimal digits, but they are much lower than those needed in cryptography, and criteria, other than factorization, to determine an integer to be prime must be used.

Fermat's theorem states that if p is a prime number, then for every integer a, $1 < a < p - 1$, the following holds

$$a^{p-1} = 1 \, mod \, p.$$

Its reciprocal is false but may be used to develop weak and strong pseudo-primality tests [MEN 97]. Guaranteed primality tests have also been worked out. They are based on another reciprocal due to Lehmer and put into practice by Pocklington. Some of them will be reviewed; however, they are only used on pseudo-prime inputs.

12.4.4. *Pseudo-primality test*

The reciprocal of Fermat's theorem is false: for a non-prime integer n there is at least one integer a such that

$$a^{n-1} = 1 \; mod \; n.$$

This allows a base a weak pseudo-primality test for n to be defined:

1) compute $a^{n-1} mod \; n$;

2) if the result is different from 1, then the integer n is a composite (it is not a prime number);

3) if it is equal to 1, then n is said to be a base a weak pseudo-prime.

Any prime number obviously is a pseudo-prime with respect to every base.

An algorithm, due to Strassen [SCH 71], computes $a^{n-1} mod \; n$ with a complexity $O(log_2 \; n)$, which is a multiplicative version of "Russian peasant" algorithm. It is an iterative program that connects the computation of a^n to that of $a^{n/2}$.

Strong pseudo-primality

Let n be an integer, pseudo-prime of base a. Let $n - 1 = d2^s$, with an odd d. If $a^d = 1 \; mod \; n$ or $a^{d2^r} = 1 \; mod \; n$ with $r < s$, then n is said to be *strong pseudo-prime of base* a. Experimentally, it is known that non-prime strong pseudo-prime are less numerous than non-prime weak pseudo-prime numbers. For instance, there are only 13 non-prime strong pseudo-prime numbers of bases 2, 3, 5 smaller than 25 10^9. Therefore, as primality test, strong pseudo-primality is better than weak pseudo-primality tests (for instance, for the same bases, they are an order of 2600 non-prime weak pseudo-prime smaller than 25 10^9).

12.4.5. *Guaranteed primality tests*

There is a theoretic primality test based on a reciprocal of Fermat's theorem, which was conjectured and proven by Lehmer [LEH 35]. An implementation has been developed by Pocklington [POC 14].

Let N be an integer for which primality should be proved. Let $N - 1 = R \times F$ be a partial factorization of $N - 1$, F being a product of prime divisors of $N - 1$, while R is not factorized. Let us assume that R and F are coprime, and $R < F$. Let $F = \prod_{j=1}^{n} q_j^{\beta_j}$ with prime numbers q_j. If there is some integer a such that $a^{(N-1)/q_j} - 1$ and N are coprime for every $j = 1, \cdots, n$, and $a^{N-1} = 1 \; mod \; N$, then N is a prime number.

The implementation is difficult to use because the choice of a is not deterministic. Nevertheless, only a partial number of prime factors of $N - 1$ are needed to prove N to be prime (without any error).

Associated construction algorithm

It is possible to use primality test in order to construct prime numbers relevant for a cryptographic use. Let R be a prime integer, for instance obtained by a sieve method. Then smaller prime integers (one of them is 2) are chosen to constitute the part F. Using the previous algorithm, it is possible to find a new prime integer of length the double of that of R. Iterating this process, prime numbers p of arbitrary large value may be constructed with the property that $p - 1$ has a large prime factor which is interesting to avoid factorization.

A method to obtain prime numbers is then the following:

1) let $j = 1$;

2) let p_j be a prime integer with 10 digits, obtained by a sieve method;

3) compute a set of small prime numbers, and some of their powers, called a *base of primes*;

4) pick at random in the base of primes some integers such that their product F_j is even, $F_j > p_j$;

5) test weak pseudo-primality of $p_{j+1} = F_j.p_j + 1$;

6) if weak pseudo-primality does not hold, then change F_j and start again;

7) change pseudo-primality base and check pseudo-primality with respect to this new base;

8) iterate 10 times instruction (7);

9) increment j. Iterate instructions (2)–(8) until a number p_j of expected size is obtained;

10) test strong pseudo-primality of p_j. If it does not hold, then return to instruction 2;

11) apply Pocklington's algorithm to p_j.

The integer p_j is a prime number with the expected number of digits.

12.5. Conclusion

In this chapter we focused on general principles of cryptography and on some of the famous encryption algorithms. It should be clear to everybody that an algorithm may be considered as reliable only for a short period of time, and never in an absolute fashion. Indeed, cryptanalytic techniques are developed to break cryptosystems.

The evolution of mathematical technology to produce invertible functions relevant for a cryptographic use was highlighted in the second part of this chapter, where DES, IDEA, and AES were presented in detail and in chronological order. This evolution follows the discovery of new mathematical objects (Feistel structures, more involved group structures, computations in finite fields, and so on). Therefore, it is quite clear that new cipher algorithms will be designed in the future just as new mathematical objects will be discovered.

A new direction will perhaps be followed, namely *quantum cryptography* [BEN 84]. It is different from the kind of cryptography presented in this chapter, as it is based on principles of quantum mechanics rather than on mathematics. Security of those cryptosystems is ensured by the impossibility to duplicate an unknown wave function (Heisenberg uncertainty principle), or in other terms, the impossibility to perform non-perturbative measures on a quantum system. Thus an adversary will measure some quantities (such as the spin of photons) in order to spy on confidential communications. Therefore, the system will be perturbated so that Alice and Bob will be aware of the attack.

12.6. Bibliography

[BEN 84] BENNETT C., BRASSARD G., "Quantum cryptography: public key distribution and coin tossing", p. 175–179, 1984.

[DAE 99] DAEMEN J., RIJMEN V., "AES proposal: Rijndael", 1999.

[DAE 02] DAEMEN J., RIJMEN V., *The Design of Rijndael: AES - the Advanced Encryption Standard*, Springer, 2002.

[DIF 76] DIFFIE W., HELLMAN M., "New directions in cryptography", *IEEE Transactions on Information Theory*, vol. 22, num. 6, p. 644–654, 1976.

[FEI 73] FEISTEL H., *Cryptography and Computer Privacy*, Scientific American, 1973.

[FIP 87] FIPS P., "81: DES modes of operation", *National Bureau of Standards*, vol. 1, 1987.

[FIP 99] FIPS P., "46-3: data encryption standard", *National Institute for Standards and Technology*, vol. 25, 1999.

[FIP 01] FIPS P., "197: advanced encryption standard", *National Institute of Standards and Technology*, vol. 26, 2001.

[KER 83a] KERCKHOFFS A., "La cryptographie militaire (première partie)", *Journal des Sciences Militaires*, vol. IX, p. 5–38, 1883.

[KER 83b] KERCKHOFFS A., "La cryptographie militaire (seconde partie)", *Journal des Sciences Militaires*, vol. IX, p. 161–191, 1883.

[LAI 90] LAI X., MASSEY J., "A proposal for a new block encryption standard", *Proc. EUROCRYPT*, vol. 90, p. 389–404, Springer, 1990.

[LAI 92] LAI X., MASSEY J., MURPHY S., "Markov ciphers and differential cryptanalysis", *Advances in Cryptology - Eurocrypt*, vol. 91, p. 17–38, Springer, 1992.

[LEH 35] LEHMER D., "On Lucas's test for the primality of Mersenne's numbers", *J. London Math. Soc.*, vol. 10, p. 162–165, 1935.

[MEN 97] MENEZES A., OORSCHOT P. V., VANSTONE S., *Handbook of Applied Cryptography*, CRC Press, 1997.

[NEC 01] NECHVATAL J., BARKER E., BASSHAM L., BURR W., DWORKIN M., FOTI J., ROBACK E., "Report on the development of the Advanced Encryption Standard (AES)", *Journal of Research of the National Institute of Standards and Technology*, vol. 106, p. 511–576, 2001.

[NSA 03] NSA, "15, Fact sheet no. 1. National policy on the use of the Advanced Encryption Standard (AES) to protect national security systems and national security information. CNSS", 2003.

[POC 14] POCKLINGTON H., "The determination of the prime or composite nature of large numbers by Fermat's theorem", *Proc. Cambridge Phil. Soc.*, vol. 18, p. 29–30, 1914.

[RIV 78] RIVEST R. L., SHAMIR A., ADLEMAN L., "A method for obtaining digital signatures", *Commun. ACM*, vol. 21, p. 120–126, 1978.

[SCH 71] SCHONHAGE A., STRASSEN V., "Schnelle Multiplikation grosser Zahlen", *Computing*, vol. 7, p. 281–292, 1971.

[SHA 49] SHANNON C., "Communication theory of secrecy systems", vol. 28, Bell Telephone Laboratories, 1949.

[STI 06] STINSON D., *Cryptography: Theory and Practice*, Chapman and Hall/CRC, 2006.

[VER 26] VERNAM G., "Cipher printing telegraph systems for secret wire and radio telegraphic communications", *Journal of the American Institute of Electrical Engineers*, vol. 45, p. 109–115, 1926.

List of Authors

Benoît BERTHOLON
CSC
University of Luxembourg
Luxembourg

Xavier BONNAIRE
Universidad Técnica
Federico Santa María
Chile

Etienne BORDE
LTCI
Telecom ParisTech
France

Christophe CÉRIN
LIPN
University of Paris 13
France

Camille COTI
LIPN
University of Paris 13
France

Julien DELANGE
European Space Agency
Netherlands

Jean-Christophe DUBACQ
LIPN
University of Paris 13
France

Emanuel GROLLEAU
LISI/ENSMA
ENSMA
Poitiers
France

Serge HADDAD
LSV
Ecole Normale Supérieure de Cachan
France

Sami HARARI
ISITV
Sud-Toulon University
France

Jérôme HUGUES
Institut Supérieur de l'Aéronautique
et de l'Espace
Toulouse
France

Fabrice KORDON
LIP6
Pierre & Marie Curie University
Paris
France

Olivier MARIN
LIP6
Pierre & Marie Curie University
Paris
France

Sébastien MONNET
LIP6
Pierre & Marie Curie University
Paris
France

Laurent PAUTET
LTCI
Telecom ParisTech
France

Maxime PERROTIN
European Space Agency
Netherlands

Laure PETRUCCI
LIPN
University of Paris 13
France

Laurent POINSOT
LIPN
University of Paris 13
France

Michaël RICHARD
LISI/ENSMA
ENSMA
Poitiers
France

Pascal RICHARD
LISI/ENSMA
University of Poitiers
France

Pierre SENS
LIP6
Pierre & Marie Curie University
Paris
France

Gael THOMAS
LIP6
Pierre & Marie Curie University
Paris
France

Sébastien VARRETTE
CSC
University of Luxembourg
Luxembourg

Index