University of Oslo
Department of Informatics

Real-Time Scheduling Support for Multimedia Communication

Hans Kristian Fjeld
hansfj@ifi.uio.no

Cand. Scient. Thesis

1st August 2001

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Distributed multimedia applications have Quality of Service (QoS) requirements that current communication systems are not able to handle. At The Center for Technology at Kjeller (UNIK) there is an ongoing research effort to develop a communication system that can meet these requirements. The system is centered around Dynamic Configuration of Protocols (Da CaPo), a highly flexible framework for communication protocols. One possible way to develop Da CaPo is to make it take advantage of real-time scheduling in an Operating System (OS) that supports this. Some work had already been done is this regard.

Da CaPo is a design for dynamically configurable lightweight communication protocols. Distributed multimedia applications have a high demand for QoS from the operating system and its communication subsystem. There are no commercial systems available that provide this kind of QoS. Da CaPo enables support for QoS from within protocols. Da CaPo can fully or partly replace existing protocol stacks in a system. The current version of Da CaPo was implemented on Sun OS. It has since been partially ported to Sun Solaris. This version has several limitations; it only runs on top of the systems native Transport Control Protocol/Internet Protocol (TCP/IP) network protocol stack, it does not support dynamic reconfiguration, and it is is not multi-threaded.

Real-Time Linux (RTLinux) is a variant of the Linux operating system that offers hard real-time capabilities. This is done by inserting a small real-time executive between the Linux kernel and the hardware. In practice, the real-time executive acts as a virtual machine for the Linux kernel. The real-time executive is implemented as a standard Linux kernel module, which means that it can be loaded into and removed from the kernel at run time. Real-time tasks run directly under the real-time executive. Linux retains all it's features as a best-effort operating system. RTLinux adds communication

facilities between real-time tasks and Linux processes.

1.2 Motivation

A few years ago the bottleneck in communication networks was low bandwidth. Available bandwidth in networks have increased tremendously in the recent years. The network is no longer considered to be the bottleneck in distributed computing. But, the protocols, that were designed for networks with low bandwidth, are still in use, and have not changed much since. The bottleneck has moved, and throughput is now dependent on the amount of computing that has to be done in the protocols. This problem is sometimes referred to as the "slow-software-fast-transmission" problem [37][11].

Da CaPo provides a framework for lightweight protocols and presents a possible solution to the "slow-software-fast-transmission". Currently, work is in progress on improving the Da CaPo design. One idea is to make the design take advantage of resource handling in in the OS, in order to be more efficient [69]. A multi-threaded design and a specialized task scheduler are both elements of this idea.

Initially, the idea was to use the micro-kernel Real-Time Operating System (RTOS) ChorusOS when creating a multi-threaded Da CaPo. However, recently there have emerged real-time variants of Linux. Unlike ChorusOS, they provide both RTOS features and standard desktop OS features in the same OS. Most applications that is in the target range for Da CaPo will run on desktop systems. Thus, an OS that mixes these properties is more suitable for an the implementation.

1.3 Problem Definition

The starting point for this thesis was to continue the work on a multi-threaded version of Da CaPo. Bjørn Volden [69] had previously done simulations on various multi-threaded designs. Tom Kristensen [25] had already begun the work on a multi-threaded implementation for ChorusOS. The general approach chosen for this thesis was to provide better operating system support for Da CaPo. More specifically, by optimizing execution scheduling in the operating system. The general goal that motivates this thesis can be summed up by one sentence:

> "How can resource management support in the operating system be utilized to improve Da CaPo?"

This is a very broad goal which certainly encompasses a lot more research possibilities than can be contained within the scope of one master thesis. In order to build on the work done previously by Bjørn Volden and

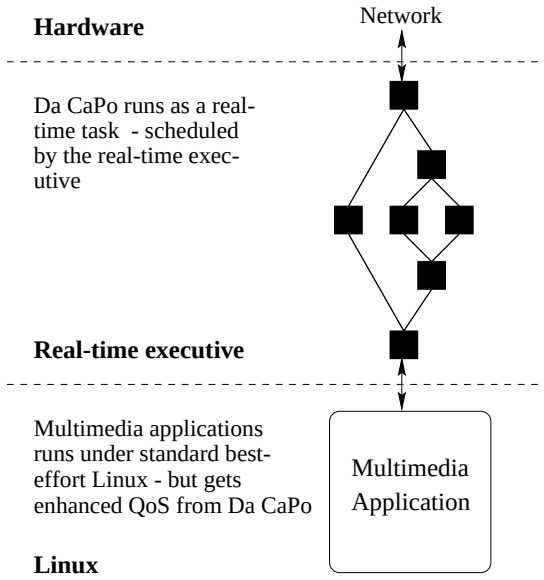

Figure 1.1: Real-time support for Da CaPo

Tom Kristensen, it was decided to concentrate on task management. And combined with a personal interest in real-time operating systems, the unified goal of this thesis can be summed up in one sentence:

> "How can Da CaPo be implemented in order to take advantage of the task management offered by a real-time operating system?"

The problem definition can be further by splitting it into three concrete part problems.

> **Problem 1:** "What is required of a real-time operating system for it to be suitable as a base for a multi-threaded Da CaPo?".

Which features is it necessary that the OS supports in order to take advantage of a multi-threaded design for Da CaPo? A multi-threaded version of Da CaPo is not yet available, but the design considerations that have been made in, amongst others, [69] and [25], are sufficient to enable a study to answer this question. Based on this answer, an OS will be chosen for the implementation of a multi-threaded Da CaPo.

> **Problem 2:** "What should the design of a multi-threaded Da CaPo look like?".

The design needs to take into consideration the features and limitations of both Da CaPo and the OS. A thorough study of both is necessary. An

overall design which includes solutions to problems with the design is the goal.

> **Problem 3:** "Is it possible to implement the new design on the chosen OS?".

The only way to completely answer this problem is to actually do the implementation of the design. This thesis only concerns itself with parts of the design of a multi-threaded Da CaPo, and the other parts of such a design are not available. Thus. a complete implementation is not possible. It will be the goal of this thesis to implement sufficiently of the code to prove that a complete implementation of the arrived at design is possible.

1.4 Method

The Association for Computing Machinery (ACM) task force report [10] creates a framework for defining the field of computing by dividing it into three different parts - theory, abstraction, and design. *Theory* involves using strict mathematical methods and formal specification to define theories and then prove or disprove them. This can be used in areas of computing where where it is practically feasible to mathematically prove a theory. For example to prove the validity of an algorithm. *Abstraction* deals with modeling and analysis of potential implementation. It involves using experimental scientific methods to construct intermediate models, and then experiment with and analyze the models. This is useful and often done prior to and in preparation of the implementation. *Design* focuses on real world implementation. Design is an iterated process that investigates alternative solutions until a maintainable, reliable, well documented and tested design is achieved.

The approach chosen for the work on this thesis was the design method. The design approach is well suited when implementation is one of the goals. This thesis involves investigation of various design considerations for the next generation Da CaPo implementation. The eventual goal is an implementation based on the results of these investigations.

The organization of the thesis is influenced by the methods described in [9]. The core structure is based on the Introduction, Methods, Results, and Discussion (IMRAD) method, which is as an American National Standards Institute (ANSI) standard.

1.5 Outlook

The thesis has been organized into 10 different chapters. Figure 1.2 gives a visual outlook of the thesis.

Chapter 1 is this introduction, where an overview of the thesis elements, the motivation behind it, and its goals are presented.

Chapter 2 presents background material on distributed multimedia and real-time, which is background material for later chapters.

Chapter 3 explains the concept of QoS. This is background material which is important in order to explain the motivation behind the Da CaPo project, and also some of the choices made in the research part of the thesis.

Chapter 4 is an introduction to Da CaPo. The chapter begins with some background material in the general area of communication protocols in order to explain the reasons for wanting to design new communication protocols specialized for multimedia

Chapter 5 deals with process management in general. Resource management is a crucial point in the development of the next version of Da CaPo. There are complex issues which are important in both design and implementation.

Chapter 6 presents an overview of the real-time Linux variants that were considered as OS base for this thesis. They are compared to each other, and also measured against the features of ChorusOS.

Chapter 7 is a more in-depth analysis of RTLinux. It examines features and limitations that is important in both implementation and design.

Chapter 8 sums up the major design considerations for the implementation of Da CaPo for RTLinux, and the choices that were made. It goes on to present the final overall design.

Chapter 9 describes the realization of the design in Chapter 8. Problems with the implementation and remaining tasks are also described.

Chapter 10 sums up the goals and content of the thesis and discusses the results.

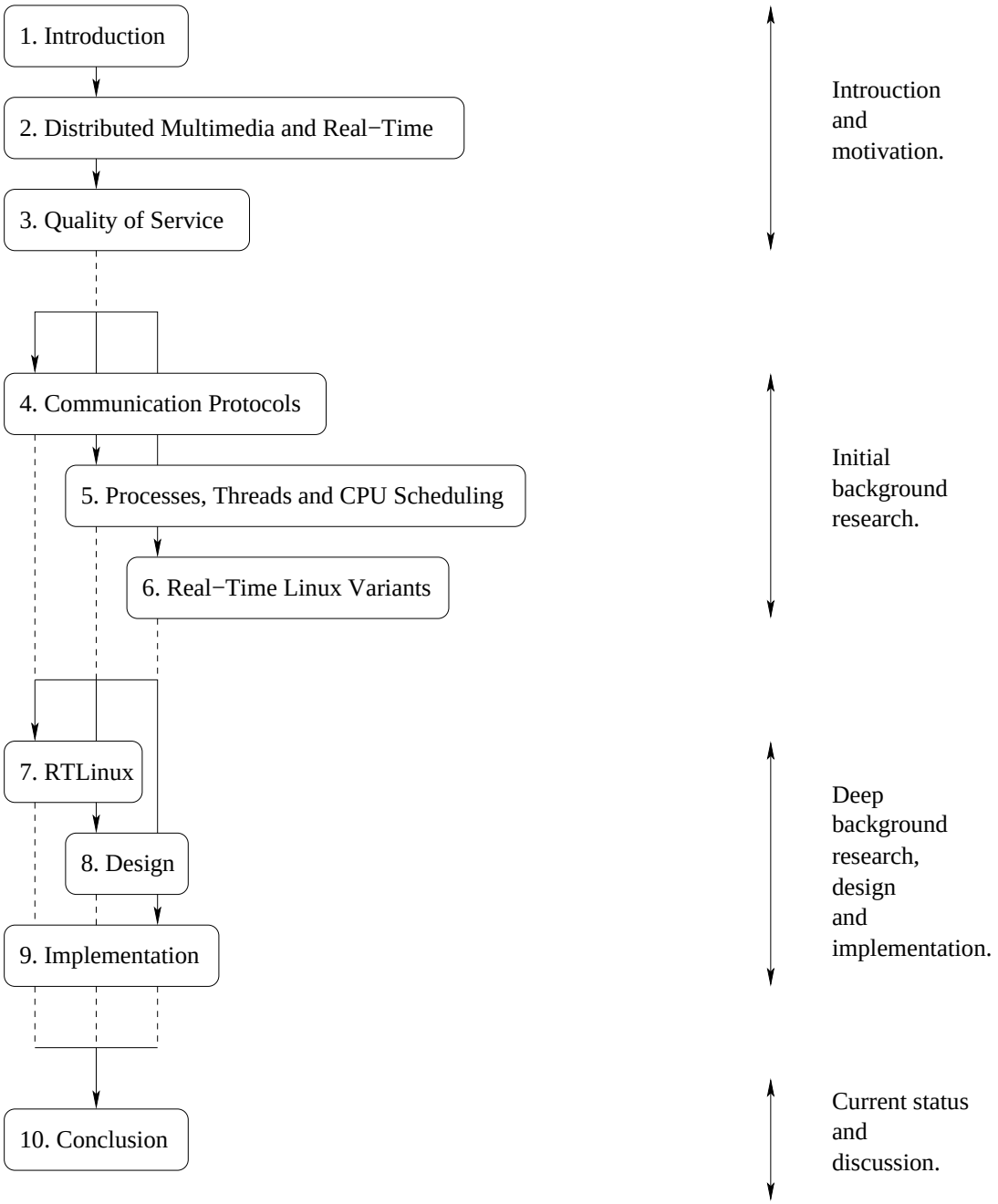

Figure 1.2: Thesis outlook.

Chapter 2

Distributed Multimedia and Real-Time

This chapter briefly introduces the concepts of multimedia and real-time. Distributed multimedia applications have QoS requirements of the communications system. In turn, in order to be able to fulfill these requirements, the communication system have requirements that demand guarantees from, and fine grained resource control in the OS. Commodity OS'es are based on best-effort resource allocation schemes, and are not able to sufficiently accommodate the needs of distributed multimedia applications.

2.1 Multimedia

There have been a variety of attempts on a definition of multimedia. The area of multimedia is a moving target, the science is still young, and there will yet be new inventions that can render older definitions obsolete. In this thesis, the following definition by Steinmetz [59] is used:

> "A multimedia system is characterized by computer-controlled, integrated production, manipulation, presentation, storage and communication of independent information, which is encoded at least through a continuous (time-dependent) and a discrete (time-independent) medium."

This thesis is concerned with *distributed multimedia*, which is basically multimedia over some form of computer network, i.e., multimedia that involves interaction between more than one computer.

2.1.1 Requirements of Multimedia

In [67], distributed multimedia applications are separated into *presentational* or *conversational* applications:

- **Presentational applications**. Presentational applications provide remote access to multimedia documents such as video-on-demand services. This represents a client-server model. Typically, there is a human user on the client system at one end, accessing multimedia data, and a multimedia content providing server at the other end.

- **Conversational applications**. Conversational applications, such as Computer-Supported Cooperative Work (CSCW), represent a peer-to-peer model. It means that human users are using a distributed multimedia communication for communicating with each other.

The common denominator is that there are human users involved, demanding of the application a certain level of quality on the content being delivered. The application, in turn, translate these demands into demands of the OS. The application demands that its data (audio, video or other) is presented to the human user in a natural, error free way. Continuous media arrives from sources like microphones, cameras and storage medias. From these sources, the data is transferred to destinations like loudspeakers and computer screens. On the way from source to destination, the digital data are processed by at least some type of move, copy or transmit operation. Plagemann et.al. [39] have made a summary of the general requirements of multimedia. The following list is an excerpt from [39].

- *High data throughput:* audio streams with telephony quality require 16 Kbit/s and audio streams with CD-quality require 1.4 Mb/s. Typical video data rates range from approximately 1.2 Mb/s for Moving Picture Experts Group (MPEG), 64 Kbit/s to 2 Mb/s for H.261, 20 Mb/s for compressed High Definition Television (HDTV), and more than 1 Gb/s for uncompressed HDTV.

- *Low latency and high responsiveness:* end-to-end delay for audio streams ought be below 150 ms to be acceptable for most applications. However, without special hardware echo cancellation, the end-to-end delay ought to be below 40 ms. In order to achieve acceptable lip synchronization between corresponding video and audio data, maximum skew is 80 ms. The maximum synchronization skew for music and pointing at the corresponding notes is +/- 5 ms. Audio samples are typically gathered in 20 ms packets, ie. 50 packets per second have to be handled per audio stream.

- *QoS guarantees:* to achieve a quality level that satisfies user requirements, the system has to handle and deliver multimedia data according to negotiated QoS parameters, eg. bounded delay and bounded jitter.

Interrupt latency, context switching overhead, and data movement are the major bottlenecks in OS's that determine throughput, latency, and respons-

iveness. In order to provide QoS guarantees, advanced management of system resources is required. All or most of the resources involved are under the control of the OS. Multimedia make new kinds of demands of the OS, and distributed multimedia adds to these demands by having additional requirements of the communication subsystem.

2.2 Real-Time Systems

Real-Time systems are characterized by the fact that the correctness of a computation is dependent on logical correctness as well as timing correctness. There is no uniformly agreed upon definition of a real-time system. This thesis uses this definition from [56]:

> "Real-Time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced."

In other words, real-time systems are used in situations where applications have deadlines to meet. The following features distinguish a real-time system [59]:

- **Predictability**. Predictably fast response to time-critical events and accurate timing information.
- **Schedulability**. Schedulability refers to the degree of resource utilization at which, or below which, the deadline of each time-critical task can be taken into account. Real-time systems offer a high level of schedulability.
- **Stability**. Real-time systems maintain stability under transient overload. The processing of critical tasks is always ensured.

The traditional real-time system often consists of a *controlling* system and a *controlled* system. For example, in an automated factory the controlled system is the factory floor with its robots, assembling stations, and the assembled parts. The controlling system is the computer and human interfaces that manage and coordinate the activities on the factory floor.

This thesis aims to use real-time systems as end-points in distributed multimedia. Thus, it can be claimed that it's an untraditional use for real-time systems.

2.2.1 Timing Constraints

Non-real-time systems usually follows the best-effort policy. The idea is that the system is shared equally by all tasks running on it. Tasks are executed

so as to minimize their average response time. The Round Robin (RR) scheduling policy (Section 5.2.1) is often used. The level of schedulability is low. In addition, these systems often have features that can result in *open ended computing*. That is, features where it is not possible to tell in advance how much time is needed when a task uses the feature.

In non-real-time computing, correctness is dependent on the logical sequence of instructions execution, but not *when* they are executed. It is required that the result of a computation is correct, but there's no deadline for the delivery of the result. In a real-time system, the result of a computation must be delivered at the right time in order to be correct.

Timing constraints for real-time tasks can be arbitrarily complicated, but the most common timing constraints for tasks are either periodic or aperiodic [56]:

- **Aperiodic task**. An aperiodic task has a deadline by which it must finish or start, or it may have timing constraints on both start and finish times.
- **Periodic task**. In the case of a periodic tasks, a period might mean once per period T or exactly T units apart.

This separation is important when choosing scheduling algorithm (Section 5.2). In distributed multimedia, there are both tasks that are aperiodic and tasks that are periodic. Thus, scheduling algorithms that handle both types of timing constraints will be more suitable.

2.2.2 Hard or Soft Real-Time

What happens when timing constraints are not met depends on the application? In some contexts, missing a deadline might have disastrous real-life consequences, but there are also situations that offer some leeway. Real-time systems are commonly split into two categories, depending on how serious it is to miss a deadline [56]. The categories are referred to as *hard real-time* and *soft real-time*. Some have attempted to define other categories. Usually these new categories fall somewhere in between hard and soft real-time. One good example of this is Kansas University Real-Time (KURT) Linux (Section 6.3), which defines a third type called *firm real-time*.

- **Hard Real-Time**. In hard real-time systems, it as unacceptable to miss a deadline. The required level of service and bounded response time must be guaranteed.

  The is required of computer systems where failure to meet a deadline could cause disastrous real-life consequences. An example of this is a computer system that controls fuel rod adjustments in a nuclear reactor. If a calculation is one microsecond late, the reactor might

go critical. But hard real-time is also used in less dramatic contexts. Missing a deadline might be unacceptable, even if the consequences are not disastrous. One example of this could be a robot in a research lab, where timing precision is critical. If the system that control the robot is late, it might render the experimental results useless. This might not be disastrous, but unacceptable nonetheless.

There are several RTOS designed especially to provide hard real-time. This is because hard real-time systems have demands that are so special and rigid that they cannot be met by normal OS'es. These RTOS'es are typically designed for use in embedded systems. ChorusOS, QNX, and LynxOS are all examples of such RTOS'es. Not until recently has anyone successfully modified a desktop OS to support hard real-time. RT-Linux (Chapter 7) and Real-Time Application Interface (RTAI) Linux (Chapter 6.2) are both examples of Linux modified in order to support hard real-time.

- **Soft Real-Time**. Soft real-time systems are used in contexts where the requirements for bounded response time is not quite as rigid. It might be acceptable to miss an occasional deadline, if for example, this improves the performance in other areas, like throughput and fairness.

  Multimedia applications normally fall into the soft real-time category. This can be well illustrated in many different examples. If we are to watch streaming video on our desktop computer we expect it to be shown at a *fairly* constant frame rate. Usually though, it is not acceptable that the system freezes all other activities in order to maintain the constant frame rate. As an example of this, imagine that a desktop computer displays a Digital Versatile Disk (DVD) movie perfectly, but at the same time does not respond to mouse clicks.

  Soft real-time requirements are not as rigid as hard real-time requirements. As a result, it is not uncommon to use standard desktop OS'es, or modified versions of these, in soft real-time systems. KURT Linux (Chapter 6.3) and RED Linux (Chapter 6.4) are both examples of a desktop OS modified to support soft real-time.

The Portable Operating System Interface (POSIX) real-time extensions [49] were developed to enable a compliant system to be able to provide the services expected in a soft real-time systems. UNIX systems and other regular fully featured OS'es that incorporate these extensions can be used in soft real-time systems.

2.2.3 Real-Time OS Support

When making a real-time application, there have been three categories of RTOS'es to choose from. Features and limitations wary from category to

category.

- **Embedded micro-kernel**. Using embedded micro-kernel, application programmers are given limited support on real-time scheduling and must implement many real-time primitives themselves. Examples are QNX, PSOS, and ChorusOS (Section 6.1).
- **Real-time extensions**. Traditional non real-time OS's with real-time extensions provide some limited support for real-time applications, but often suffer performance deficiencies due to the original non real-time kernel architecture. Examples of OS'es that have been extended in this way are Microsoft Windows NT and Sun Solaris.
- **Commercial RTOS**. Commercial RTOS's provide the best real-time features, but are generally expensive and not open, which often make them ill suited in many research situations. Examples are Lynx and RT-Mach. (ChorusOS used to be in this category as well, but has since dropped out of that market segment.)

For research projects on real-time scheduling and applications, it is best if an open yet popular RTOS is available. This is one of the motivations behind the many projects now concerned with turning Linux into an RTOS. The real-time Linux variants fall into the category of "real-time extensions", but with some new ideas that make them more similar in features to the commercial RTOS'es. Some of these real-time Linux variants are discussed and compared with a commercial RTOS in Chapter 6. RTLinux is explored in more detail in Chapter 7.

2.3 Summary

This chapter has given brief presentations of distributed multimedia and real-time computing as seen in the context of this thesis. The motivation to put these two subjects together arises from the desire to add specialized scheduling support to Da CaPo. Da CaPo aims to fill the need for communication systems that can meet the requirements of distributed multimedia, and real-time systems specialize in scheduling support. The problem, is that the host systems in distributed multimedia usually need to be desktop systems, while most RTOS'es perform badly or not at all as desktop systems. The real-time Linux variants are real-time extensions to a desktop system, but have features that make them comparable to commercial RTOS'es. Thus, real-time Linux seems to be an ideal choice in this thesis.

Chapter 3

Quality of Service

This chapter explains what QoS is and why it is important in distributed multimedia. It briefly presents the historical conception of the term QoS, and explains what makes the early definitions insufficient for multimedia systems. This is followed by some more detail on what has been done to develop the QoS concept for multimedia. And finally there is a section on resource management and QoS in operating systems. This chapter is included because QoS is very important in the motivation for developing Da CaPo, and the term is used extensively throughout the thesis.

3.1 The QoS Concept

Beyond the intuitive meaning of QoS as system characteristics that influence the perceived quality of a service, there is little consensus on the precise meaning, let alone the formal definition of QoS. However, there have been several proposals for such a definition. The view in this thesis fits the definition proposed by Vogel [67]:

> "QoS represents the set of those quantitative and qualitative characteristics of a distributed multimedia system necessary to achieve the required functionality of an application."

Multimedia applications pose requirements to the supporting system in order to be able to run. The following paragraphs explains how QoS requirements are propagated through, and handled by the system, and how they can be expressed.

To simplify the QoS concept, it's common to view the multimedia system as only two parts; multimedia applications as the part requesting QoS, and the supporting system as the part fulfilling QoS. In QoS research, a more detailed approach is necessary. QoS requirements are interpreted and handled differently within different parts of the multimedia system. When the goal is to be able to provide improved QoS support, it is necessary to see this

from a more detailed perspective. Thus, QoS requirements must be specified differently in the different parts of the system. QoS parameters must be translated, or mapped into different parameters by one level for them to be meaningful at the next level. For example, a user might request something like "a better quality picture" for the video stream he is watching. To the application, this could be translated into a higher resolution with more colors. Finally, this could again be translated into a request for more bandwidth in the communication subsystem. With this layering in mind, this definition by Tanenbaum [64] explains the concept of *service* well:

> "A service is a set of primitives (operations) that a layer provides to the layer above it. The service defines what operations the layer is prepared to perform on behalf of its users, but it says nothing at all about how these operations are implemented. A service relates to an interface between two layers, with the lower layer being the service provider, and the upper layer being the service user."

Plagemann et. al [45] divide, for the purposes of QoS, a multimedia system into five functional layers, between which the QoS values must be translated to fit the service provided in each:

- **Networks**. In this context, networks means the physical networks that connect end-systems.
- **Operating systems**. All the other elements, except networks, are dependent on the OS.
- **End-to-end protocols**. The protocols in the end-systems that enable applications to communicate across the networks.
- **Data management systems**. Applications in the end systems that comprise a repository of functions to store, retrieve, and change persistent data (that is, a file system, or a database system).
- **Applications**. Applications that are written to solve specific multimedia related problems. This includes multimedia applications, but also applications that are more indirectly involved, like for example a voice to text translator.
- **Human Computer Interaction (HCI)**. The interface through which human users and multimedia systems communicate. This includes the devices that for sensory output or input, like a monitor or a keyboard, but also interfaces at a lower level, like a graphical user interface displayed on a monitor.

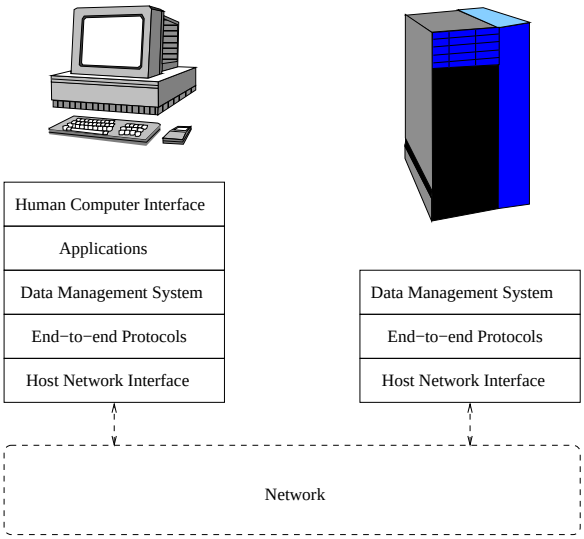

Figure 3.1: Layered architecture of a sample multimedia system.

In this thesis, the focus is on QoS the type of support provided by Da CaPo, which belongs in the end-to-end protocols layer, but the concept is best explained with a general overview.

Figure 3.1 illustrates how two connected, albeit different, multimedia systems are separated into functional layers after the approach described by Plagemann et. al. The system on the left represents a presentational application (Section 2.1.1). It is a desktop multimedia system, which eventually presents the user with human recognizable QoS. The system on the right illustrates a data management system, a multimedia storage server. In order to meet the QoS requirements by the user at the desktop, these requirements must be propagated through all the necessary layers.

3.2 Modern QoS

The term QoS was first used in network protocols [64]. It was provided by the network layer, and enhanced in the transport layer. QoS parameters were *defined*, but might or might not be supported, and were neither monitored nor enforced. Table 3.1 shows a table from [64] that lists typical transport layer QoS parameters. The protocols that were designed then are the same that are being used today. But, the term QoS has since been broadened to encompass a lot more than it did when first used. The following describes, in general terms, some of the most important areas in which the traditional QoS concept have had to be augmented in order to support modern QoS

Connection establishment delay
Connection establishment failure probability
Throughput
Transit delay
Residual error ratio
Protection
Priority
Resilience

Table 3.1: Typical transport layer QoS parameters [64].

requirements:

- **QoS Specification**. A much broader range of parameters is necessary to specify the QoS. Chapter 3.2.2 discusses QoS specification in more detail.
- **QoS Guarantees**. Due to its nature, multimedia systems need to be able to offer *guarantees* about the QoS it offers. If a certain minimum level of QoS cannot be guaranteed, then it will simply not be acceptable for the human user of the system.
- **QoS Negotiation**. A multimedia system must incorporate mechanisms for negotiating QoS with the application, between layers in the system, with other end-systems, and with the network. Chapter 3.2.3 discusses QoS negotiation in more detail.
- **QoS Monitoring**. QoS must be monitored by the system, in order to be maintained and enforced.
- **QoS Enforcement and Adaptation**. A multimedia system must be able to respond to changes in the system and/or the QoS requirements. Chapter 3.2.4 discusses QoS enforcement and adaptation in more detail.

The main motivation behind the current development of Da CaPo is to provide better QoS in the communication subsystem. Hence, ultimately, QoS is at the heart of the motivation behind this thesis, as well. However, it is not the intention to go into great detail on all aspects of modern QoS. Thus, only some of the more relevant key elements are briefly explained.

3.2.1 QoS Semantics

The semantics of QoS has come to mean the degree of commitment in the QoS offered by the service provider. Table 3.2 shows the four levels of QoS

QoS Semantics	Description
Guaranteed QoS	The service provider guarantees QoS through strict resource allocation.
Best-effort QoS	All parties do the best they can to meet the QoS requirements, but there are no guarantees.
Compulsory QoS	If at any time the service provider is unable to meet the QoS requirements, the connection is aborted.
Threshold QoS	If at any time the service provider is unable to meet the QoS requirements, the service user is warned.

Table 3.2: QoS Semantics [45].

Category	Example Parameters
Performance-oriented	End-to-end delay and bit rate
Format-oriented	Video resolution, frame rate, storage format, and compression scheme
Synchronization-oriented	Skew between the beginning of audio and video sequences
Cost-oriented	Connection and data transmission charges and copyright fees
User-oriented	Subjective image and sound quality

Table 3.3: The five categories of QoS parameters [67].

semantics. Best-effort QoS is the weakest form of QoS, and generally what is offered by most communication subsystems today. Guaranteed QoS is the strongest form of QoS. The semantics are useful in explaining various QoS approaches. However, current development suggests that most multimedia applications will require guaranteed QoS.

3.2.2 QoS Specification

QoS is usually expressed as a set of parameter and value pairs, or tuples. Generally, there are two approaches to quantifying QoS parameters [45]. The service user specifies either a target value, or a target range for each QoS parameter. All QoS requirements, both those made by the user, and those internal to the system must be quantified and expressed like this. Vogel et. al [67] suggests that QoS parameters can be separated into the five different categories shown in Table 3.3, based on certain characteristics.

3.2.3 QoS Negotiation

The goal of QoS negotiation is for the service user and the service provider to agree on QoS level. Three basic negotiation schemes and one reconfiguration scheme have been identified for Da CaPo [45][37], they are illustrated in Figure 3.2, and briefly explained below.

- **Unilateral negotiation**. The calling service user proposes a QoS specification that cannot be changed by the service provider or the called service user. There is no real negotiation, the request can only be accepted or rejected.

- **Bilateral negotiation**. The negotiation takes place between the service users. The service provider is not allowed to change the QoS specification. However, service provider as well as called service user may reject the request.

- **Combined negotiation**. This involves the calling user, service provider, and called service user. The calling service user passes a QoS specification to the service provider, which may downgrade the values of the QoS parameters. The resulting QoS specification is passed to the called service user, which in turn may also weaken the QoS parameters. This final QoS specification is returned to the calling service user. The service provider as well as the called user may reject the request.

- **Reconfiguration**. Reconfiguration is triggered when a monitored connection is affected by a change in the available resources. The connection might have to be degraded or terminated due to negative changes, or it could be upgraded if more resources become available. Reconfiguration can be triggered in either system. The initiating system sends a reconfiguration request, followed by suggestions for new protocol configurations, to which the peer system responds to.

In Da CaPo, QoS negotiation is the domain of the configuration manager (Section 8.1).

3.2.4 QoS Realization and Enforcement

In traditional QoS, the approach was to specify QoS requirements when a connection is first set up, then attempt to provide that level of QoS in a best-effort manner. There were no guarantees, and no further monitoring of QoS.

In modern QoS, realization and enforcement have become important issues. Monitoring is required when offering stronger than best-effort QoS.

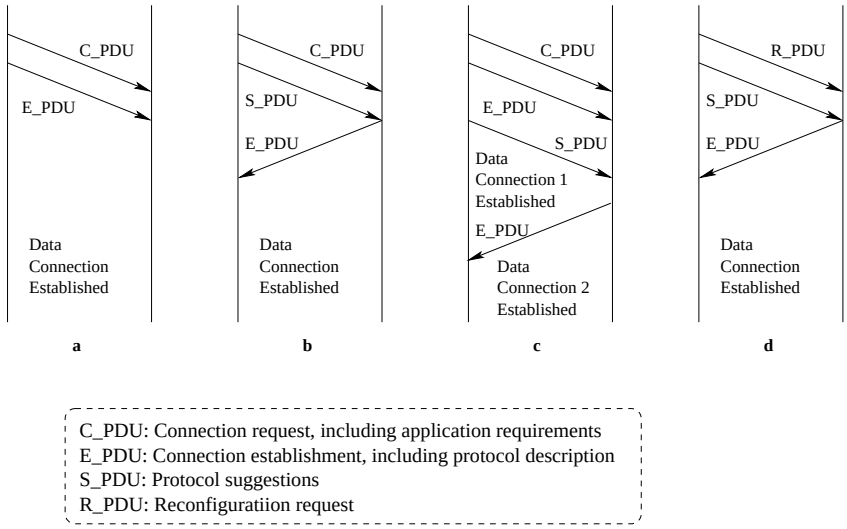

Figure 3.2: Unilateral negotiation (a), bilateral negotiation (b), combined negotiation (c), and reconfiguration (d).

Resource reservation is required in order to offer guaranteed QoS. Adaptation enables the system to allow re-negotiation of QoS. Da CaPo implements flexible protocols that can be dynamically reconfigured in order to adapt to required or forced changes in QoS.

OS support for QoS has also become a focus for research. Guaranteed QoS requires reservation of resources managed by the operating system. And, even weaker QoS semantics are a lot simpler to implement with better operating system support. Commodity OS'es are best-effort systems, and have not been designed to offer this kind of detailed resource management.

RTLinux (Chapter 7) offer a detailed level of resource management that can be used by Da CaPo to monitor, realize and enforce QoS.

3.3 Resource Management and QoS

One of the main tasks of an OS is the management and distribution of OS-managed resources between applications. The traditional approach is to let the OS respond to requests by allocating resources as *fairly* and *efficiently* as possible (best-effort, Section 2.2). As discussed in Chapter 2, this is not sufficient for distributed multimedia. These applications are more rigid in their demands for resource allocation in order to function correctly.

The resource requirements are specified with QoS parameters. Typical application-level QoS specifications include parameter types like frame rate, resolution, jitter, end-to-end delay, and synchronization skew [39]. These

high-level parameters are translated into low-level parameters that maps to OS-managed resources, like CPU time per period, amount of memory, and average and peak network bandwidth.

Plagemann et. al have identified that QoS actions that are related to resource management in the OS can be defined by the following tasks [39]:

- **Specification and allocation of resources**. Prior to the execution of a task, it must be ensured that there are sufficient resources to fulfill its QoS requirements. There must be a way to specify the resource requirements so that they can be allocated by the OS.
- **Admission control**. Test whether enough resources are available to satisfy the request without interfering with previously granted requests. The way a test is performed depends on requirement specification and allocation mechanism used for this resource.
- **Allocation and scheduling**. There must be mechanisms that ensure that a sufficient share of the resource is available at the right time. The type of mechanism depends on the resource type. Resources that can only exclusively be used by a single process at a time have to be multiplexed in the temporal domain. In other words, exclusive resources, like CPU or disk Input/Output (I/O), have to be scheduled. Basically we can differentiate between fair scheduling, real-time scheduling, and work and non-work conserving scheduling mechanisms. So-called shared resources. like memory, basically require multiplexing in the spatial domain, which can be achieved, for example, with the help of a table.
- **Accounting**. The actual amount of resources that was consumed in order to perform the task is recorded in some way. Accounting information is often used in scheduling mechanisms to determine the order of waiting requests. Accounting information is also necessary to make sure that no task consumes more resources negotiated and steals them (in overload situations) from other tasks. Furthermore, accounting information might trigger system-initiated adaptation.
- **Adaptation**. Adaptation might be initiated by the user/application or the system and can mean to downgrade QoS and corresponding resource requirements, or to upgrade them. Adaptation leads in any case to new allocation parameters. Accounting information about the actual resource consumption might be used to optimize resource utilization.
- **Deallocation**. The task of freeing up the resources.

Since this thesis is concerned with real-time scheduling support for Da CaPo, allocation and scheduling is the central one among theses tasks. But

because of their detailed resource management, RTOS'es are well suited in the performance of all these tasks. Using an RTOS as the base for an implementation of Da CaPo offers a lot of possibilities for improved resource management in protocols. For example, more detailed control of Da CaPo as a set of tasks enable monitoring, which in turn enable accounting and then adaptation. Chapter 8 explains the suggestions in this thesis for how some of these possibilities can be exploited.

3.4 Summary

QoS is central in the motivation, and the term is used extensively in discussions and arguments in the thesis. This chapter introduces and gives a brief overview of the concept. It explains why QoS is necessary in the context of multimedia. The most relevant key elements of QoS are explained. Improved resource management is a key element in the desire to re-design and implement Da CaPo for RTLinux. For this reason, this chapter also includes a discussion on the relationship between resource management and QoS.

Chapter 4

Communication Protocols

Distributed applications continue to grow more sophisticated, and to demand more and more of the computer networks. In the last few years, the development in network technology have accelerated tremendously, much thanks to the expansion of the Internet. But in the same time frame, the technology in the in the end-systems, the communication protocols, have not changed much. This stagnation has made the protocols the current bottleneck in computer networks. This chapter gives a brief historical overview of the traditional protocols. It subsequently explains what is wrong with them, and why Da CaPo was developed to provide the QoS required in distributed multimedia. This includes an overview of the original Da CaPo design, which was the natural starting point for the design work of this thesis.

4.1 Traditional Protocols

The term *protocol* was first introduced in the layered reference models. Halsall [19] define a protocol like this:

> "A set of rules formulated to control the exchange of data between two communicating parties."

In this thesis, *traditional* protocols refers to all protocols that are based on the layered model of Open Systems Interconnection Reference Model (OSI-RM) or similar models. OSI-RM is described here because it is a useful tool for logical abstraction when designing and explaining any communication subsystem, and is used in that capacity later in the chapter when explaining Da CaPo.

4.1.1 OSI-RM

The International Organization for Standardization (ISO) created Open Systems Interconnection (OSI) reference model as part of an effort to standardize

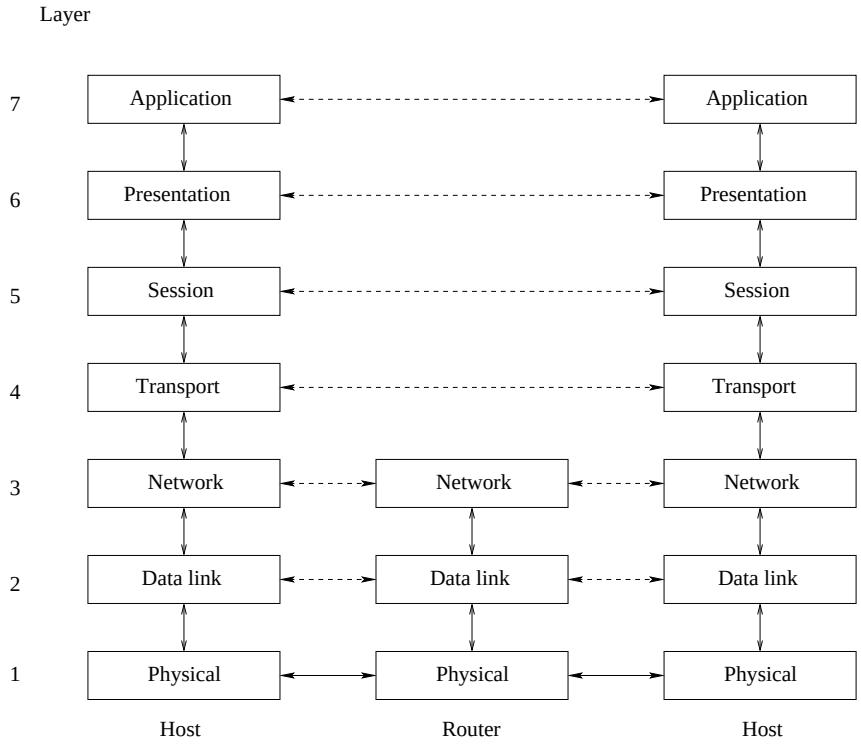

Figure 4.1: The OSI reference model.

the protocols used in communication subsystems. The idea was that applications should be able to communicate with other applications across networks, regardless of operating system, computer architecture, and network architecture. The layered model of the OSI-RM has become the standard reference model for any communication subsystem [64]. Figure 4.1 shows the layered logic of the OSI-RM. Each layer performs a well defined function, and there is a clear, logical separation between each layer.

- **Physical layer**. The physical layer is concerned with transmitting raw bits over a communication channel. This part of the specification largely deals with mechanical, electrical, and procedural interfaces, and the physical transmission medium, which lies below the physical layer.
- **Data link layer**. The main task of the data link layer is to take a raw transmission facility and transform it into a line that appears free of undetected transmission errors to the network layer.
- **Network layer**. The network layer is concerned with controlling the operation of the sub-net. The main tasks of this layer are key elements

of inter-networking, and include routing, congestion control, accounting, and address mapping.

- **Transport layer**. The basic function of the transport layer is to accept data from the session layer, split it up into smaller units if need be, and pass these to the network layer. The transport layer is is also the layer responsible for the establishment and deletion of connections across the network. At connection time it must also determine what type of service (and the QoS) it is able to provide to the layer above. The transport layer might also multiplex several transport connections onto the same network connection.

- **Session layer**. The session layer can be used to establish sessions between two machines. The main services offered in a session is dialog control and synchronization of connections.

- **Presentation layer**. Unlike the lower layers, which are just interested in moving bits, the presentation layer is concerned with the syntax and semantics of the information transmitted. A typical example of a service in this layer is encoding/decoding of data in an agreed upon format.

- **Application layer**. The application layer contains a variety of protocols which provide services that are commonly needed by applications. The applications use the network through these protocols.

Not all layered protocol designs use all the layers present in the OSI-RM. In these designs, some of the layers are combined and/or left out. A good example of this is the TCP/IP protocol suite, which is briefly presented next.

4.1.2 TCP/IP

This brief presentation of TCP/IP is included because this set of protocols is what the Internet uses. That means that it is the most widely used set of protocols yet. According to Tanenbaum [64], the TCP/IP model were first defined in 1974. This was before the OSI-RM was published.

Figure 4.2 shows how the TCP/IP reference model corresponds to the OSI-RM. There are no presentation or session layers in the TCP/IP model. The functionality of the data link layer and the physical layer have been combined into one host-to-network layer. The host-to-network layer protocol is often completely or mostly implemented in hardware. The remaining three layers are quite similar to their counterparts in the OSI-RM.

- **Internet layer**. TCP/IP is a packet-switching network based on a connection-less inter-network layer. This layer, called the Internet layer is the linchpin that holds the whole architecture together. It's task is

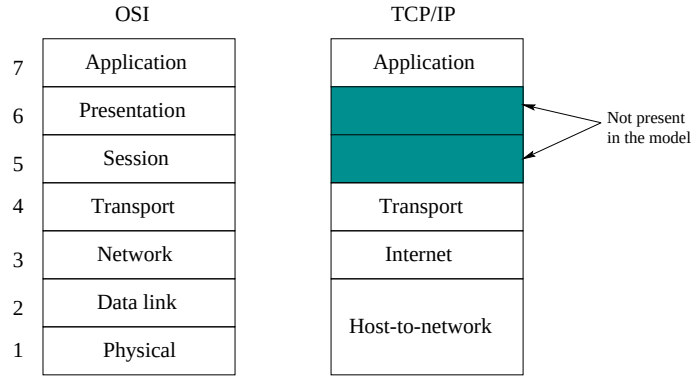

Figure 4.2: The TCP/IP model compared to the OSI reference model.

to guide every individual packet through the network from source to destination. The Internet layer defines an official packet format and protocol called the Internet Protocol (IP).

- **Transport layer**. The layer above the Internet layer is called the transport layer, the same as its OSI-RM counterpart. Its tasks are also the same; establish end-to-end connections, etc. Two protocols have been defined for this layer. Transport Control Protocol (TCP) is a reliable connection oriented protocol. User Data-gram Protocol (UDP) is an unreliable, connection-less protocol.

- **Application layer**. The application layer contains all the higher level protocols that applications interface with. This includes an ever increasing number of protocols with special functionality, like Simple Mail Transfer Protocol (SMTP) for electronic mail, File Transfer Protocol (FTP) for file transfer, Hyper-text Transfer Protocol (HTTP) for World Wide Web (WWW), etc.

4.2 New Approaches to Protocols

It has been and still is a focus of research to invent new ideas and designs for communication protocols. The need for new protocols arises in different settings. In this thesis, the focus is on the requirements of distributed multimedia (Chapter 2). Two recurring ideas in much of this research is the notion of moving away from the layered models, and the concept of light-weight protocols to remove redundancy. Both are ideas that are used in the Da CaPo design.

4.2.1 Model vs Implementation

As shown above, the layered models of OSI-RM and TCP/IP work well as models for abstracting the functionality of the communication system. In the models, each layer has clearly separate functions and there are well defined interfaces between the layers. However, basing an implementation straightforwardly on the reference models will not necessarily yield the best performance. Several research efforts in the last few years have concluded that the bottleneck in the communication system is in the end systems, and is due to costly computations in the end-system protocols, for example [44],[68] and [11].

When the traditional protocols were designed, the bottleneck was in network infrastructure. This led to a protocol design where reducing the number of bits on the wire was emphasized at the expense of having more computing done in the end-systems. A new approach is to abandon the layered model altogether, and instead decompose the communication subsystem into *functions* that execute required functions only, possibly with specialized hardware. Several research efforts show that this strategy can be used to decrease computations in the end-system protocols [11][67]. This is done in Da CaPo, as is explained below.

4.2.2 Lightweight protocols

The general idea behind lightweight protocols is based on the experience that traditional layered protocol stacks, like the OSI and TCP/IP models, cause overhead due to redundant functionality, like error control, flow control, connection multiplexing and segmentation/re-assembly, to appear in more than one protocol in the same stack. When these protocols were designed, it was based on assumptions that the networks had low bandwidths and high error-rate probabilities. This situation has changed as network technology has improved a lot. In lightweight protocols, the idea is that each protocol provide the bare minimum of functionality that is necessary to execute a single protocol task. In this way, redundant overhead is avoided. The Da CaPo design is a framework for lightweight protocols, as is explained below.

4.3 Da CaPo

The Da CaPo design is central in this thesis because the original design is starting point for the design of a Da CaPo version with real-time scheduling support. What subsequently follows is an overview of the original Da CaPo design. [16],[41], [68], [46], [44], [43], and [37] was used as sources.

Dynamic Configuration of Protocols (Da CaPo) is a framework that provides an environment which allows dynamic configuration, adaptation, and reconfiguration of protocols. Configuration of protocols is done by Da

Property	**Improvement**
Dynamic configuration	Enhanced QoS
More flexible protocol	Enhanced QoS
Reduced redundancy	Better performance
Reduced complexity	Better programming

Table 4.1: Summary of motivations for Da CaPo.

CaPo based on application requirements, available resources, and properties of the available network services.

Protocols are made up of protocol modules. Each module incorporates a basic function. Complex functionality is accomplished by assembling modules in specific graph compositions. Da CaPo is able to provide enhanced QoS to applications because modules are basic functions that can be assembled as needed, and thus very flexible, and because the assembled graphs are dynamically re-configurable. Da CaPo can achieve good performance because redundancy is avoided by building protocol graphs that only incorporate the necessary protocol functions (the concept of lightweight protocols, as explained in Section 4.2.2). Furthermore, some of the complexity is shifted from protocols to the framework, making the implementation of the modules themselves less complex. Table 4.1 summarizes these key motivations behind the design of Da CaPo.

4.3.1 Overview of the Da CaPo model

Da CaPo is based on a three-layer model, as shown in Figure 4.3.

- **A layer**. The top layer is the A layer. It represents the distributed applications, which access the C layer through the AC-interface. The AC interface must be designed so that it enables the applications to specify and request QoS.

- **C layer**. The middle layer, the C layer, represents the end-to-end communication protocols. An optimal protocol stack is constructed from available modules, in order to support the requirements of the A layer in a best possible manner, based on the services that are available in the T layer. Normally this involves enhancing the services of the T layer to satisfy the requirements of applications.

- **T layer**. The T layer, at the bottom of the model, represents the transport infrastructure. In this context, transport infrastructure means simply infrastructure over which the end-systems can communicate. The T layer is *not* equivalent to the transport layer of the OSI-RM (Section 4.1.1). Da CaPo *can* use a high level service like Internet

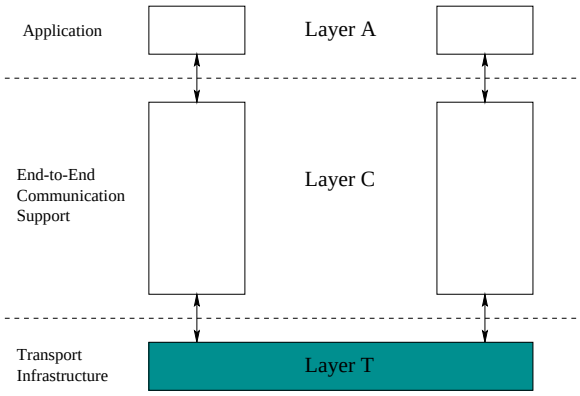

Figure 4.3: The three layered Da CaPo model [16]

Protocol (IP), or a lower level service, like the Media Access Control (MAC) layer in the Institute of Electrical and Electronics Engineers (IEEE) 802.3 (Ethernet) protocol, for this purpose.

4.3.2 Modules as Building Blocks for Protocols

The three layered model of Da CaPo is fundamentally different from the layered models used in OSI-RM and TCP/IP. In Da CaPo, the layered model is used to logically separate Da CaPo's application interface, the A layer, and the transport infrastructure interface, the T layer, from the "real" protocol functionality, which is located in the C layer.

The Da CaPo design uses the idea of decomposing the communication subsystem into protocol functions. Protocol functions are used in protocol graphs for abstract protocol specification. Figure 4.4 shows the relationship between the abstract *functions*, the intermediary *mechanisms*, and the implemented *modules*.

- **Protocol functions**. Protocol functions describe typical protocol communication tasks, e.g., error control, flow control, presentation coding, en- and decryption, etc. In general, protocol functions can be realized by different protocol mechanisms.

- **Protocol mechanisms**. Protocol mechanisms specify the rules supporting the data transfer from one service user to its peer, and they specify the data handling. Communicating entities have to use the same protocol mechanisms, otherwise they cannot understand each other.

- **Modules**. Implementations of protocol mechanisms in Da CaPo consists of sending and receiving modules. The modules are the real build-

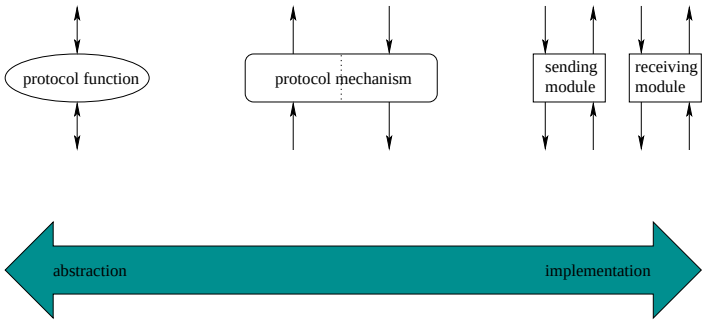

Figure 4.4: Implementing protocol functions [44].

ing blocks for protocol configuration.

4.3.3 Architecture

The architecture for the framework of Da CaPo consists of four cooperating active entities and a passive database. Figure 4.5 shows the relationships between the various components. The following list, based on [16], gives a short overview of each component:

- **CoRA**. Configuration and Resource Allocation (CoRA) is the heuristic program that determines appropriate protocol configurations and QoS for the requested layer C connection. For the determination process, CoRA uses the specification of the layer C services, the description of the properties of modules and layer T services, as well as information about available system and network resources. [41] describes the internals of CoRA.

- **Connection manager**. The connection manager assures that communicating peers are using the correct protocols for data transfer between applications. It negotiates a common protocol configuration and QoS for a layer C connection via a standardized protocol configuration called Management Transfer Protocol (MTP). Besides the negotiation, the connection manager initiates the establishment and release of connections, and coordinates the reconfiguration of existing connections.

- **Resource manager**. The resource manager provides an efficient runtime environment for Da CaPo protocols. It establishes and removes modules, executes the corresponding module functions and offers several services (e.g. timer-, signal-, and buffer-services).

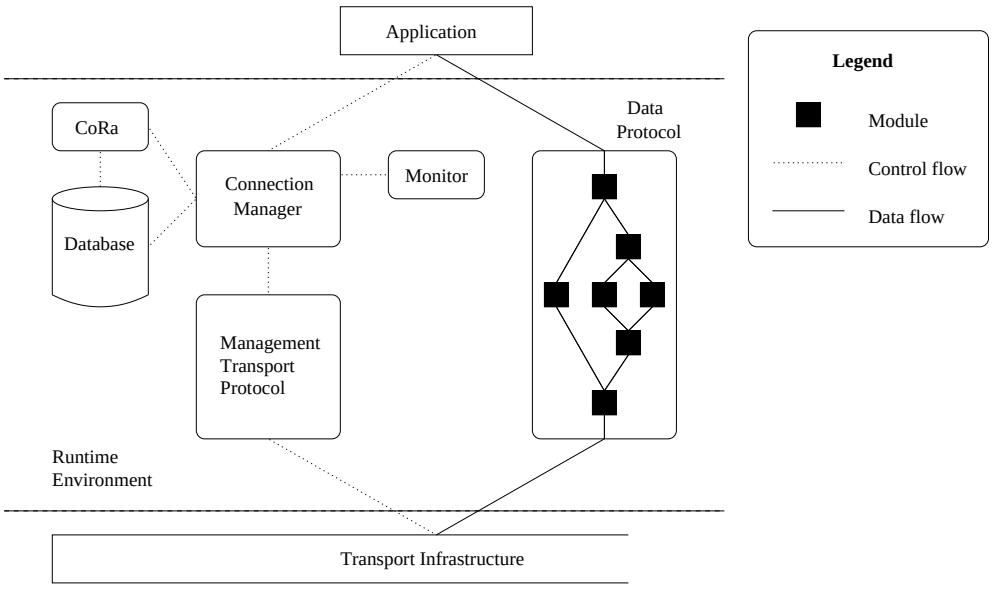

Figure 4.5: Architecture of Da CaPo [37].

- **Monitor**. The monitor supervises the properties of layer C connections. CoRA determines a protocol configuration that satisfies the application requirements. When the monitor detects that requirements can no longer be met, it informs the connection manager, which in turn asks the configuration manager to reconfigure or terminate the protocol.

- **Database**. A database stores information needed by the Da CaPo system. Examples of such information are the specification of protocol graphs, protocol functions with their realizing protocol mechanisms, modules with their properties, layer T services with their default properties as well as available resources. Particularly CoRA and the resource manager rely heavily on the stored information.

4.4 Summary

In Da CaPo, module graphs are constructed based on the requirements of the applications. The resulting protocols should be optimal in each situation. When the applications require better QoS support than the transport infrastructure can offer, modules that enhance the QoS are used in the module graph. However, enhancing QoS uses resources, thus the resulting QoS offered by Da CaPo is also influenced by the availability of resources. Figure 4.6 shows how multiple factors influence the resulting QoS. This shows

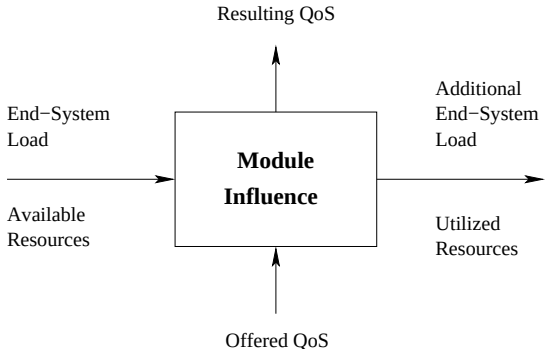

Figure 4.6: Factors of module properties [37].

how the availability of resources affects the QoS, and that explains why it is important to be able to manage these resources as efficiently as possible.

Chapter 5

Processes, Threads and CPU Scheduling

This chapter provides a necessary introduction of process management. This is background research that was required in order to evaluate the various real-time Linux variants (Chapter 6), choose the right design alternatives (Chapter 8), and perform a correct implementation of the design (Chapter 9).

5.1 Threads and Processes

This section is intended as a discussion on what threads are, and why multi-threaded programming is preferable. A variety of sources was used; [55], [10], [28], [5], [14], [65], [6], [62], [50], and [57].

At the lowest level, a Central Processing Unit (CPU) is not able to distinguish between logical tasks, it simply continuously executes a sequence of simple instructions. At a higher level, the OS distinguishes between tasks by making them entities called *processes*. The OS is able to perform several tasks simultaneously by alternating between which process gets executed. The task of alternating between processes, and deciding when and for how long processes should execute is called *CPU scheduling*.

The switch from one process to another is called a *context switch*. The term includes all that has to be done that is directly related to the switch. When a process is "switched out" before it has completed all its execution, it is queued to be resumed at a later time. For this to be possible, the OS must store all information about the current process' state. When "switching in" a resumed process, that process' stored state needs to be reloaded. This is all part of the context switch.

Processes are relatively heavyweight in terms of the overhead incurred at context switches, due to the amount of process context that must be saved and restored at context switches. Processes are also heavyweight with respect

to communication, since processes can share data only through the explicit use of inter-process communication mechanisms such as message passing.

Threads add a second level of concurrency. Applications are allowed to establish concurrent threads of control within process. Threads of control are variously referred to as threads, C threads, pthreads, tasks or Lightweight Processes (LWP)'s. Like processes, each thread has its own program state, but all threads belonging to the same process share the same address space, plus some additional data that are shared between all threads owned by the same process. These data need not be stored for each thread, instead they are stored only with the owning process. Thus, a context switch between threads of the same process incur significantly less overhead. Threads have been identified as being particularly important in real-time systems because they increase the level of control of the processes in a system.

5.1.1 Advantages to Using Threads

Multi-threaded programming is generally considered a to be better way to program because it leads to programming that is more structured, and programs that are more efficient. There is no point in making a multi-threaded program of a simple single process program. But complex applications, perhaps consisting of more than one process, will be simpler to program and provide much better performance. The following summarizes some of the most important advantages to using threads:

- **Better responsiveness**. Blocking a thread will not necessarily block the entire process. When a function call in a single-threaded process blocks, the process will be frozen until that function call is finished. When a function call in a thread blocks, other threads of the same process may still execute. This advantage does not usually apply to user-level threads.

- **Better Inter-process Communication (IPC)**. Communication between threads with the same parent process is usually a lot faster than traditional inter process communication. An application that uses multiple processes to accomplish its tasks can be replaced by an application made up of multiple threads instead of processes. This will actually by simpler to program and provide much better performance.

- **More efficient use of system resources**. For each registered process the system maintains a significant amount of data. A thread is a lot cheaper to maintain. This goes especially for user-space threads, where all the thread specific data are kept in user space. But in general threads use only a fraction of the system resources used by processes.

All of these advantages are arguments for implementing Da CaPo as a multi-threaded application.

5.1.2 Common Problems with Threads

This section describes the most significant problems that are encountered when programming with thread. They can all be avoided with careful programming.

- **Critical section**. A critical section is a part of the code in which data that is shared between threads can be inconsistent. At a higher level it can be viewed as a section of code in which there is no guarantee for other threads that the state of some data is true. If other threads access these data during a critical section, it could cause crash, lock up, produce incorrect results, or just about any other unpleasant thing. The solution is generally to deny other threads access to these data while a thread is a critical section. Mutexes and semaphores [62] are mechanisms used for this purpose.

- **Priority inversion**. This problem that can occur when using fixed priority scheduling (section 5.2.1). Priority inversion happens when a low-priority thread prevents a high-priority thread from running - as a result of an interaction between scheduling and synchronization. The scheduler decides that one thread should run, but synchronization requires that another thread runs first, so that the priorities of the two threads appear to be reversed. The occurrence is best explained with an example:

  1. A low-priority thread acquires a shared resource.
  2. A thread with a higher priority thread arrives.
  3. The scheduler preempts the low-priority thread, and lets the new thread run.
  4. The high-priority thread attempts to acquire the shared resource, but blocks because it is already held by the low-priority thread.
  5. The low-priority thread gets to run until it releases the shared resource. The high-priority thread must wait for the low-priority thread.

  From this point, the problem can get even worse. Theoretically, there could be a large number of low-priority threads that all hold shared resources that the high-priority thread needs to be able to run. In effect causing the only high-priority thread to run last.

- **Asynchronous thread cancellation**. It is usually very difficult to guarantee that the recipient of an asynchronous cancellation request will not be in a critical section. If a thread should die in the middle of a critical section, this could leave some critical data in an inconsistent state. Code that can sensibly deal with asynchronous cancellation

requests is *not* referred to as async-safe. The best solution is to use deferred cancellation.

These problems do not constitute good reasons to avoid using threads. The only reasons for not using threads when they are supported are that they may add complexity to simple programs, and that portability might be slightly weakened.

5.1.3 User-space vs. Kernel-supported Threads

The threads packages available can be divided into two main types; *user-space threads*, also called soft threads, and *kernel-supported threads*. This section describes some of the typical characteristics of the two different types.

- **Kernel awareness**. User-space threads run without any special support from the kernel. The kernel sees only the process that implements the threads support. This process is scheduled by the kernel as one process, according to whatever scheduling support is implemented in the kernel. Kernel-supported threads falls into two sub-categories. One is "pure" kernel-supported systems, where the kernel is aware and responsible for scheduling of all threads. The other is a *hybrid* system, where the kernel cooperates with a user-space process when scheduling the threads.

- **Performance difference**. How much time is spent on the context switches makes up most of the performance difference. In most systems, a context switch between user-space threads is faster than between kernel-supported threads. On some systems, however, the difference is not great. Linux is one of these systems.

- **Problems with functionality**. One major problem with user-space threads, is with blocking system calls. Because the kernel doesn't know about the threads, a blocking system call will not just block the calling thread, it will block the whole process.

Based on these characteristics, the only good reason for using user-space threads is if threads are not supported in the kernel.

5.2 CPU Scheduling Algorithms

The goal of traditional scheduling in time-sharing OSes is optimal throughput, optimal resource utilization and *fairness* (section 2.2). In real-time systems, the main goal of the scheduler is to provide a schedule that allows all, or as many time-critical tasks as possible to be processed in time, according to their deadlines.

Most commodity OSes perform priority scheduling and provide time-sharing and real-time priorities. The typical approach to ensure that a time critical multimedia task receives sufficient CPU time is to assign a real-time priority to this task. The POSIX real-time extensions [49] takes this approach for adding real-time scheduling to a UNIX system. Several real-time OSes offers real-time priority driven scheduling and possibly other scheduling policies, after the POSIX fashion.

There have been a lot of attempts to invent improved algorithms for real-time scheduling. Many of them are variations of basic algorithms. The two most widely used [59] basic algorithms are Earliest Deadline First (EDF) and Rate Monotonic (RM). They are both *reservation-based* algorithms, which means that they are useful (even optimal) in systems where tasks are relatively periodic. It was long held that multimedia computing mostly generated periodic tasks.

It has later been shown [39] that it is not necessarily correct that multimedia generates mostly periodic tasks. This has led to a lot of research on scheduling algorithms that are able to adapt to the unpredictable resource requirements of some multimedia applications. In this thesis, this new branch of scheduling algorithm is refereed to as *proportional share resource allocation* algorithms. These algorithms do not provide any guarantees about the predictability of the schedule.

5.2.1 Priority Scheduling

Most multitasking operating systems today use some sort of priority driven scheduling. This includes all the desktop operating system widely in use today, such as Microsoft Windows, UNIX, and Linux. The most common scheduling policies are based on RR or First-In, First-Out (FIFO) priority driven scheduling.

In priority scheduling, tasks are assigned priorities according to how important/urgent they are. The task with the highest priority gets to run first.

In the early multitasking systems, simple scheduling algorithms like RR were employed in order to share CPU time fairly between tasks. This type of algorithm by itself assumes that all tasks are equally important. In real life, this was not the case. Thus, priorities were added to the simple algorithms.

The real-time scheduling options defined in the POSIX real-time extensions are based on priority driven scheduling (use of other algorithms is allowed, but none are defined [48]. More specifically, POSIX defines the FIFO and RR scheduling algorithms with priorities:

- **First In First Out**. FIFO is possibly the simplest algorithm for CPU scheduling. It provides no fairness. In FIFO, the system maintains a list of the runable tasks. When the CPU is ready, the first task in the

list gets to run. When a new task is ready to run, it is placed at the end of the list of runable tasks. A scheduled task runs until completion. There is no preemption of tasks, no time-sharing, no fairness. First come - first serve applies.

- **Round Robin**. RR is one of the oldest, simplest, fairest and most widely used [62] algorithms. It is similar to FIFO, but a little more complex. In order to provide fairness, each task is assigned a time interval called its *quantum*, which it is allowed to run. When a task has used up its quantum, it's preempted and put at the end of the list of runable tasks.

When priorities are added to FIFO and RR, there is one list of runable task for each priority level. Tasks with higher priority always gets to run before tasks with lower priority. Running tasks can be preempted by higher priority tasks. To prevent a high-priority task from running indefinitely, thereby starving other tasks, there are mechanisms for adjusting its priority level, even as it runs.

A fundamental problem with priority-based scheduling in a multimedia context, is that it is hard to properly allocate resources for a job based purely on the job's priority. Priority-based schemes have several potential problems, including starvation, priority inversion, and lack of fine-grain allocation [58]

5.2.2 Reservation Based Algorithms

Reservation based algorithms are widely used in real-time systems. Algorithms in this category are extremely efficient, in terms of CPU usage, in systems where tasks are periodic [59]. In this context, this means that tasks appear at a regular interval, have a regular processing time, and a regular deadline. Reservation based algorithms can be used to implement guaranteed QoS, but at the cost of fairness. They are also less flexible in the sense that tasks are required to have periodic behavior.

The two most widely used algorithms in this category, EDF and RM, are both based on the same periodic task model; tasks are characterized by a start time s, the time at which the task requires the first execution, and a fixed period p in which the task requires execution time e, and a deadline d (Figure 5.1).

In multimedia, there are encoding schemes for continuous audio and video that generate constant bit rate streams. Reservation-based scheduling is optimal in these cases. However, reservation-based scheduling will probably never be widely accepted for general purpose systems, because of the difficulty of correctly estimating a thread's required portion and period [58].

- **Earliest Deadline First**. EDF is a dynamic and preemptive algorithm [59]. At every new ready state, the scheduler selects the task

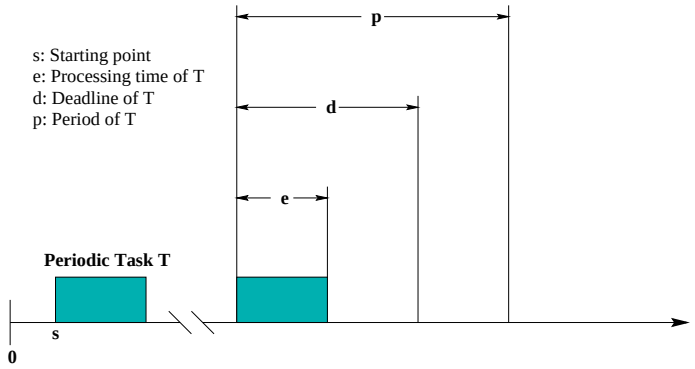

Figure 5.1: Periodic task model

with the earliest deadline among the tasks that are ready and not fully processed. Whenever a new task becomes ready to run, it's deadline must immediately be computed by the EDF algorithm. This means that the running task is preempted, and a new scheduling order is computed. The new task is scheduled immediately, if it's deadline is earlier than that of the interrupted task. The processing of the interrupted task is continued later on, according to the EDF algorithm. EDF is not only an algorithm for periodic tasks, but also for tasks with arbitrary requests, deadlines and service execution times. In this case, no guarantee about the processing of any task can be given.

EDF can be implemented on top of a priority driven scheduler by assigning priorities based on deadlines. The highest priority is assigned to the task with the earliest deadline; the lowest to the one with the furthest. This implicates that priorities of all tasks might have to be readjusted whenever a new task arrives, which may cause considerable overhead.

- **Rate Monotonic**. The RM principle was introduced by Liu and Layland in 1973. It is an optimal static, preemptive priority driven algorithm for periodic jobs [59]. Optimal in this context means that there are no other static algorithm that it is able to schedule a task set which cannot also be scheduled by RM. It's been widely used in real-time scheduling for a long time. And there are numerous algorithms that are variations on the same principle.

  Scheduling of a task is only done right before it starts executing. At this time, the task is assigned a priority. Tasks are executed according to the priorities already assigned to them. There is never any rescheduling of processes.

  The following five assumptions are necessary prerequisites in order to

apply RM [59]:

1. The requests for all tasks with deadlines are periodic, ie. have constant intervals between consecutive requests.
2. The processing of single task must be finished before the next task of the same data stream becomes ready for execution. Deadlines consists of run-ability constraints only, ie. each task must be completed before the next request occurs.
3. All tasks are independent. This means that the requests for a certain task do not depend on the initiation or completion of requests for any other task.
4. Run-time for each request of a task is constant. Run-time denotes the maximum time which is required by a processor to execute the task without interruptions.
5. Any non-periodic task in the system has no required deadline. Typically, they initiate periodic tasks or are tasks for failure recovery. They usually displace periodic tasks.

5.2.3 Proportional Share Resource Allocation

A few years ago, it was believed that distributed multimedia would mainly consist of continuous media streams with constant bit rates [59]. That is, distributed multimedia applications would have periodic behavior, and reservation based algorithms that use the periodic system model would be optimal. However, recent development in multimedia have produced more efficient encoding schemes for audio and video, that produce media streams with variable bit rates. This leads to multimedia applications that have a much less predictable behavior.

It is very likely that this trend will continue in the future and make resource requirements even harder to predict. The latest development in operating system support of multimedia still emphasize QoS, but integrate often adaptability and support both real-time and best-effort requirements [39].

In proportional share allocation, resource requirements are specified in shares of the respective resource types available in the system. In a dynamic system, where tasks dynamically enter and leave the system, a share depends both on the current system state and the current time. When a resource is requested, the share allocated will always be in proportion to the share requirements of the other tasks. In a dynamic system, pure proportional resource allocation can give guarantees about the proportional shares, but the shares may be arbitrarily low. Thus, a pure proportional share algorithm can't be used to implement guaranteed QoS. But this type of algorithms can be made very flexible and even adaptive, as is shown by for instance the Adaptive Rate Controlled (ARC) [73] algorithm.

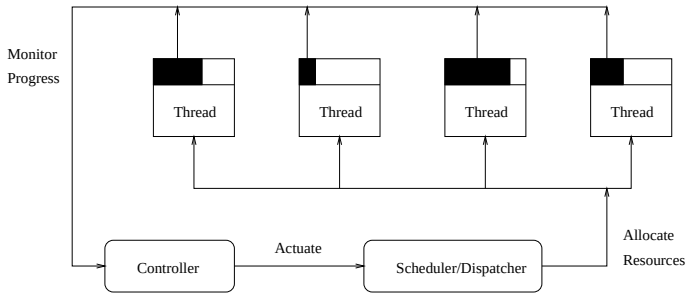

Figure 5.2: High-level diagram of the real-rate scheduler.

- **Real-Rate Scheduling**. This scheduler is a good example of a proportional share allocation scheduling algorithm design. A prototype has been implemented, and is described in [58]. The approach is to provide a scheduler that performs better for *real-rate* applications. Real-rate applications are defined, in [58], as "applications with specific rate or throughput requirements in which the rate is driven by real-world demands".

  The scheme is based on the notion of progress. The *scheduler* dispatches threads in order to ensure that they receive their assigned proportion of the CPU during their period. A *controller* periodically monitors the progress made by the threads, and adjusts each job's proportion automatically. Figure 5.2 shows the rough architecture of the scheduler. A feedback controller monitors the rate of progress of job threads, and calculates new proportions and periods based on on the results. Actuation involves setting the proportion and period of the threads. The scheduler is a standard proportion/period reservation-based scheduler. The controller's execution period and the dispatch period can be different. The most difficult part in this type of scheme is how to estimate the progress of the individual threads. This particular scheduler solves this by defining several classes of application and suitable metrics for each class.

5.3 Summary

The chapter had described what multi-threading is and why it is generally desireable to program using threads. It has also presented the concept of CPU scheduling and some of the most important scheduling algorithms available. This chapter contains necessary background information for the evaluation of real-time Linux variants (Chapter 6), and for the design and implementation of a multi-threaded Da CaPo (Chapters 8 and 9).

Chapter 6

A Survey of Real-Time Linux Variants

Earlier chapters have explained why we want to design a version of Da CaPo for a real-time operating system to take advantage of the resource control and real-time scheduling facilities offered in such systems.

Lately there has been a tremendous development in equipping Linux with real-time capabilities. Thus, it was perceived to be of interest to determine whether Linux could now be an suitable as operating system for further research with Da CaPo. A three step process was chosen as the approach to achieve this goal. The first step was a general study of all real-time Linux projects. The second step was to do a comparison between the different variants, and also put them up against a commercial RTOS. The third step was to, based on the findings from the first two steps, pick one of the real-time Linuxes for more in-depth study.

This chapter presents an overview of the first two steps. ChorusOS was chosen as the commercial RTOS to compare the real-time Linuxes with. In order to present the comparisons in a natural way, a general overview of ChorusOS is given first. Preceding that is general descriptions of the real-time Linux projects RTAI, KURT Linux and RED Linux. RTLinux is described in more detail in Chapter 7. Finally, there is a summary of the comparisons, and a conclusion. RTLinux was found to be the most likely candidate for our purposes.

6.1 ChorusOS

ChorusOS is a micro-kernel based real-time operating system. It was originally developed at Institut National de Recherche en Informatique et en Automatique (INRIA), in France. Later it was commercialized by Chorus Systemés, which was acquired by Sun Microsystems in 1997. Sun is marketing ChorusOS as an embedded real-time operating system, especially tar-

geted at use in public switches and PBXes, and supplies tools designed for development of embedded applications. Sun also have plans for ChorusOS as a part of JavaOS.

ChorusOS comes with a POSIX Application Program Interface (API), including the POSIX real-time extensions, as well as a native API for additional advanced real-time functions. The operating system itself is implemented mostly in C, with some parts written in C++.

6.1.1 Goals of ChorusOS

The goals of the Chorus project have evolved along with the system itself. When the work on this thesis commenced, ChorusOS was quite ideally suited, as can be seen by summarizing the goals of the Chorus project at the time:

- When ChorusOS was commercialized, Chorus Systemés put a lot of effort into equipping ChorusOS with high performance UNIX emulation. The company wanted Chorus to be seen as an alternative to AT&T UNIX, re-engineered, easier to maintain, and oriented to future user requirements [63]. A ChorusOS subsystem, called MiX, enabled it to run UNIX System V binaries.

- Heavy emphasis was also put on making ChorusOS a distributed operating system. The intention was to allow UNIX programs to run on a collection of machines connected by a network. Various extensions have been added to support distributed applications.

- Another goal was to support real-time applications. This is done by allowing real-time programs to have direct access to the micro-kernel, and provide real-time scheduling mechanisms.

- Chorus Systemés also introduced object-oriented programming into ChorusOS.

Unfortunately, when Sun acquired Chorus Systemés, it precipitated a radical alteration of theses goals. ChorusOS is today specialized as an embedded operating system (Section 2.2) for telecom equipment.

6.1.2 System Structure

ChorusOS is structured in layers, as shown in Figure 6.1. At the bottom is the micro-kernel, called the nucleus. The idea is to move as much functionality as possible away from the kernel and into user space. Thus, the micro-kernel only provides minimal management of names, processes, threads, memory and communication. The rest of the operating system is provided by processes in the higher layers. ChorusOS separates between three different types of processes:

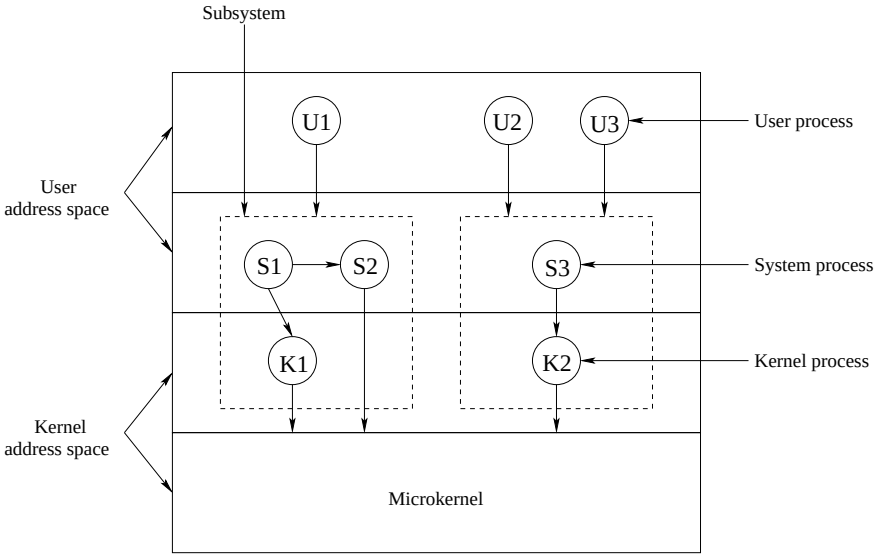

Figure 6.1: The layered structure of ChorusOS, with a micro-kernel, subsystems and processes. The figure is from [64].

- **Kernel processes**. Kernel processes run right on top of the micro-kernel, and share the kernel address space with the micro-kernel. These processes can be dynamically loaded and removed during system execution and provide a way to extend the functionality of the kernel without increasing its size and complexity.

- **System processes**. System processes run in the next layer. They use the user address space, but are allowed to send messages to kernel processes and make calls to the micro-kernel.

- **User processes**. User processes run in the top layer. They share address space with system processes. These processes run on top of a specific subsystem, and all services they require is provided via that subsystem.

A collection of kernel and system processes can work together to form a subsystem. Chorus Systemés used to provide two subsystems, MiX, a UNIX subsystem, and COOL, an object oriented subsystem.

Real-time processes run as system processes, in order for them to be able to make full use of the micro-kernel without intervention or overhead.

6.1.3 ChorusOS Abstractions

The micro-kernel provides and manages six key abstractions that together form the basis for ChorusOS. These concepts are processes, threads, regions, messages, ports, and unique identifiers.

- **Actors**. Processes are referred to as actors. They are essentially the same as processes in other operating systems; containers that encapsulate resources.
- **Threads**. A process can have one or more threads. Each thread is similar to a process in that it has its own stack, stack pointer, program counter, and registers. However, all threads in a process share the same address space, and some other resources.
- **Regions**. A consecutive range of addresses is called a region. Each region is associated with some piece of data, such as a program or file. Regions play a major role in memory management in ChorusOS.
- **Messages**. The basic communication paradigm is ChorusOS is message passing. A thread can communicate to any other thread by passing a message to it.
- **Ports**. Messages are not addressed to a thread, but to an intermediate structure called a port. A port is a buffer for incoming messages. Each port belongs to a thread. Ports can be grouped to form extended communication facilities.
- **Unique identifiers**. The last kernel abstraction relates to naming. Most kernel resources are named by a 64-bit unique identifier. Once a unique identifier is assigned to a resource it is guaranteed never to be reused for another resource.

Three other abstractions are used by ChorusOS, which are jointly managed by the micro-kernel and subsystems. They are *capabilities*, which is a name for a resource normally managed by a subsystem, *protection identifiers*, and *segments*, which is a concept used in memory management.

6.1.4 Kernel Structure

The ChorusOS micro-kernel consists of four pieces, as shown in Figure 6.2. They are:

- **Supervisor**. The supervisor is at the bottom, closest to the hardware and thus machine dependent. It manages raw hardware and catches traps, exceptions, interrupts, and other hardware details, and handles context switching.

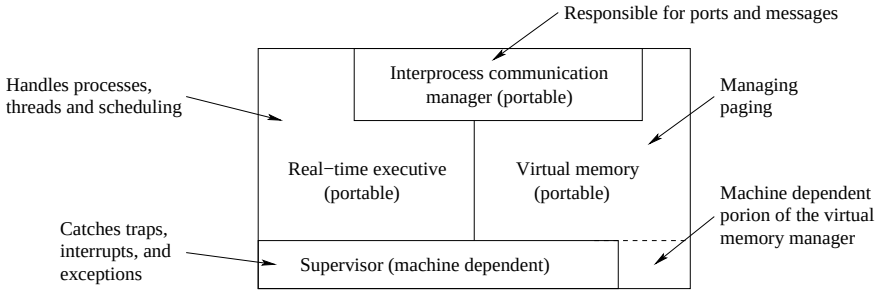

Figure 6.2: Structure of the ChorusOS kernel, according to [64].

- **Virtual memory manager**. The virtual memory manager handles the low level part of the paging system. Most of the virtual memory manager is machine independent.
- **Real-time executive**. The real-time executive is responsible for managing processes, threads, and scheduling. It also takes care of arranging for synchronization between threads of for mutual exclusion and other purposes.
- **Inter-process communication manager**. The inter-process communication manager handles unique identifiers, ports and message passing. It relies on the real-time executive and virtual memory manager.

According to ChorusOS documentation, the four parts of the kernel are constructed to be modular, so that changes to one do not affect any of the others.

6.1.5 Process Management and Scheduling

A process in ChorusOS is a collection of active and passive elements that work together to perform some computation. The active elements are the threads. The passive elements are an address space and a collection of ports. There are three different kinds of processes. They are summarized in Table 6.1:

- **Kernel processes**. Kernel processes run in kernel mode and all share the same address space with each other and the micro-kernel. They can be loaded and unloaded during execution, but other that can be thought of as extensions to the micro-kernel.
- **System process**. Each system process has its own address space. They are unprivileged, which means that they cannot execute I/O or

Type	Trust	Privilege	Mode	Space
User	Untrusted	Unprivileged	User	User
System	Trusted	Unprivileged	User	Kernel
Kernel	Trusted	Privileged	Kernel	Kernel

Table 6.1: The three kinds of processes in ChorusOS [63]

other protected instructions. However, they are trusted by the kernel to make kernel calls.

- **User processes**. User processes are untrusted and unprivileged. This means that they cannot perform I/O or kernel calls directly. They must get these services via their respective subsystem.

Every active process in ChorusOS has one or more threads that execute code. Each thread has its own private context, which is saved when a thread blocks and restored again when it resumes execution. A thread belongs to one specific process for its entire life cycle. ChorusOS defines four execution states for threads:

- **ACTIVE**. A thread in the *ACTIVE* state is either currently running or awaiting its turn for a free CPU. In both cases it is unblocked and ready to run.
- **SUSPENDED**. When a thread is in the *SUSPENDED* state it means that it has been suspended by itself or another thread
- **STOPPED**. A thread is put the *STOPPED* state when the owning process is suspended.
- **WAITING**. And when a thread performs a blocking operation, it is put in the *WAITING* state until the operation finishes.

The synchronization mechanisms provided for threads are semaphores and mutexes. Communication between threads is usually done by passing messages, but two threads in the same process can also communicate using shared memory.

CPU scheduling is done using priorities on a per-thread basis. Each process has a priority and each thread has a relative priority within its process. The kernel keeps track of the priority of each thread in ACTIVE state and runs the one with the highest priority. If there are more than one thread with the same priority, they each get to run for a given time period, a time-slice, after which must surrender the CPU to another thread. This is consistent with the Round Robin scheduling algorithm described in Section 5.2.

To accommodate for real-time applications, an additional feature has been added to the scheduling algorithm. A distinction is made between

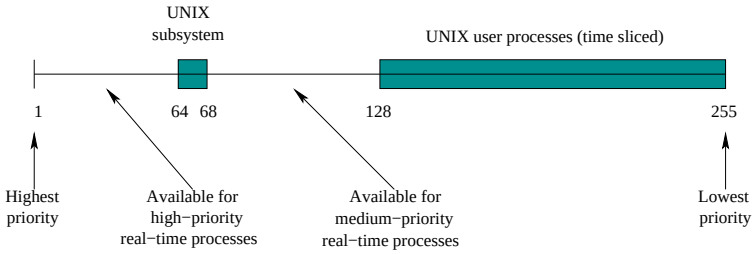

Figure 6.3: ChorusOS - priorities and real-time processes.

threads whose priority is above a certain level and threads whose priority is below it. The threads with priority above this level are not time-sliced (Figure 6.3). These threads run until they finish, or voluntarily releases the CPU.

6.2 RTAI

RTAI, is being developed at Department of Aerospace Engineering, Politecnico de Milano (DIAPM). Zentropix is a US based company that offers commercial support for real-time Linux. Zentropix's solutions are based on RTAI, and they also contribute to the development of RTAI.

The work on RTAI started in 1996. The first version was based on an early version of RTLinux (Section 7). In 1998, the Linux kernel version 2.2 was released. The new kernel introduced a cleaner and more featured interface to the kernel [29], and RTAI was rewritten for the new kernels. The motivation to proceed with development of RTAI was provided by the emergence of applications requiring periodic schedulers, semaphores, and messages in combination with enhanced one-shot timer and Floating Point Unit (FPU) support, which at that time were not supported by RTLinux [29]. As a result of this on-going work, RTAI now supports Linux kernel 2.2 on both single processor systems (UP) and Symmetric Multiprocessing (SMP) systems based on either Intel 486 or Pentium class CPU's. RTAI's architecture provides guaranteed, hard real-time scheduling of selected tasks on a system which retains all the features and services of standard Linux.

The RTAI developers initial involvement with RTLinux has made RTAI a solution that has many similarities with RTLinux. Like RTLinux, RTAI is based on a design where a small real-time executive is loaded into the Linux kernel, and Linux is scheduled as the lowest priority task by this real-time executive.

The fundamental difference between RTAI and RTLinux is rooted in a different attitude toward features. The RTLinux developers emphasize that

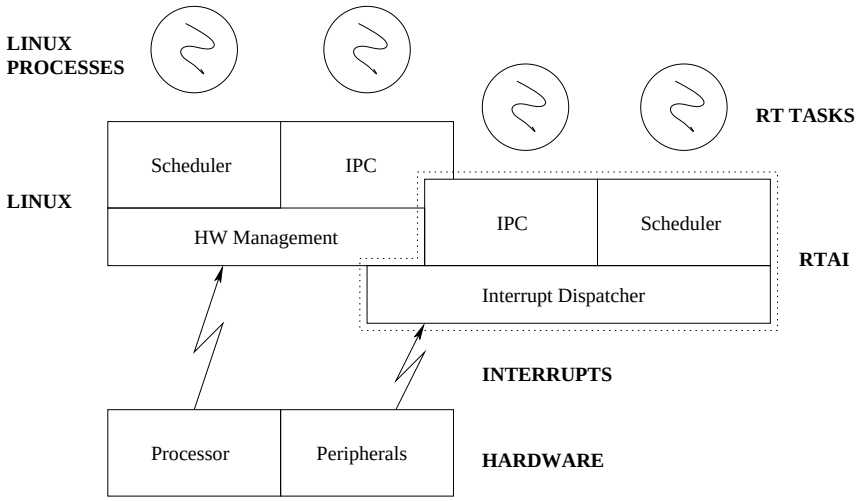

Figure 6.4: The RTAI architecture.

they feel it's important to keep the system as small as possible [3], while the RTAI developers are clearly adding features at a faster pace.

6.2.1 Architecture

Figure 6.4 illustrates the architecture of RTAI. The figure shows a simplified view of how RTAI relates to the Linux kernel. Like RTLinux, RTAI is implemented as Linux kernel loadable modules. The architecture is based on two concepts:

- The Hardware Abstraction Layer (HAL) is basically the modifications that must be done to the Linux kernel in order for it to support RTAI. The HAL provides a framework to extend the kernels capabilities and enables an array of potential OS enhancements. This concept is also known to be used in Windows NT [31]. It is implemented by applying a small kernel patch (about 70 lines of code) to the Linux kernel, and imposes a negligible impact on the kernel's performance [29].

- The HAL provides the necessary support in the Linux kernel to be able to load the three core loadable modules that make up the RTAI. The three core loadable modules are the RTAI module (`rtai`), the scheduler module (`rtai_sched`), and the module which implements Real-Time FIFO (RT-FIFO)s (`rtai_fifos`).

The real-time tasks are also implemented as Linux kernel modules, and can be dynamically loaded into and removed from the kernel, after the RTAI

core modules have been loaded.

The elements of RTAI, as seen in Figure 6.4, are described briefly here. The next section includes a more complete discussion of RTAI scheduling and inter process communication:

- **Interrupt dispatcher**. The interrupt dispatcher is called when hardware interrupts occur. The dispatcher handles the interrupt controller and takes care of possible pending interrupts or service requests. It activates the right handler depending on where the interrupt originates from: RTAI, Linux, or both.
- **Scheduler**. The scheduler is in charge of distributing the CPU(s) to different tasks present in the system, including Linux. Scheduling occurs when certain system calls are made, and on timer handler activation [31].
- **IPC**. IPC is provided by FIFOs that allow Linux processes and RTAI tasks to exchange byte-oriented data streams.

6.2.2 Process Management and Scheduling

RTAI's task scheduler allows hard real-time, fully preemptive scheduling based on a fixed priority scheme. RTAI supports one-shot and periodic timers, although not simultaneously.

- The *periodic* scheduler is called at regular predefined intervals. It looks for ready tasks or tasks with expired period, and elects the one with the highest priority to run. A scheduled task runs until a higher priority task is elected, the tasks terminates, or the task calls a blocking system function.
- The *one-shot* scheduler is similar, except it's decided how long a task should run before it starts executing. A timer is set for that long, and the task runs until that time is used.

RTAI's IPC provides means of data transfer and data sharing between standard Linux processes and real-time processes running within the system. Two general types of IPC:

- **Real-time to non real-time**. Real-time to non real-time communication is done using RT-FIFOs and shared memory.
- **Real-time to real-time**. The mechanisms provided for real-time to real-time include semaphores, mutexes, message queues and Remote Procedure Call (RPC)'s.

Zentropix has contributed with a POSIX overlay for RTAI, which implements the POSIX hard real-time extension [29].

6.3 KURT Linux

KURT is a research project at the University of Kansas, Lawrence. The researchers there argue that a variety of new applications, including multimedia applications have real-time requirements that make them fall somewhere between the two traditional real-time categories of soft real-time and hard-real time (Section 2.2). These applications typically have fine the grain timing requirements typical of hard real-time systems, and the service requirements typically provided in soft real-time systems.

Based on these arguments they introduce a new category, which they call *firm* real-time. The KURT project turns Linux into a system that handles firm real-time and still provides all the services of standard Linux.

6.3.1 Design Model

The KURT Linux design model is based on three key elements:

- **Kernel patch**. Since many firm real-time applications need to track time at resolutions higher than that provided by standard Linux, the kernels temporal granularity had to be refined. KURT Linux accomplishes this by providing a kernel patch which allows for microsecond resolution timers.

- **KURT core**. The KURT core takes care of scheduling real-time events. The KURT core implements a set of system calls in the Linux kernel in order to support firm real-time.

- **Real-time modules**. Real-time modules (RTMods) implement the functionality of a real-time task. These are implemented as Linux loadable kernel modules. Thus, they run in kernel mode and are allowed access devices and other parts of the kernel, in ways not permitted to normal user processes.

The KURT core is responsible for scheduling real-time events and invoking the appropriate RTMods at the appropriate time. The KURT scheduler is an explicit plan scheduler, which means that applications need to specify explicitly the times at which real-time events are to occur.

Since Linux is a time sharing operating system, many of its components are not tuned to perform well under real-time constraints. KURT Linux provides a way to turn off some of these components when real-time tasks are being performed. It is done by providing to different kernel modes:

- **Normal mode**. In normal mode, all processes are allowed to run. Since all processes are allowed to run, some processes may be using kernel services that might block interrupts for an extended period of time.

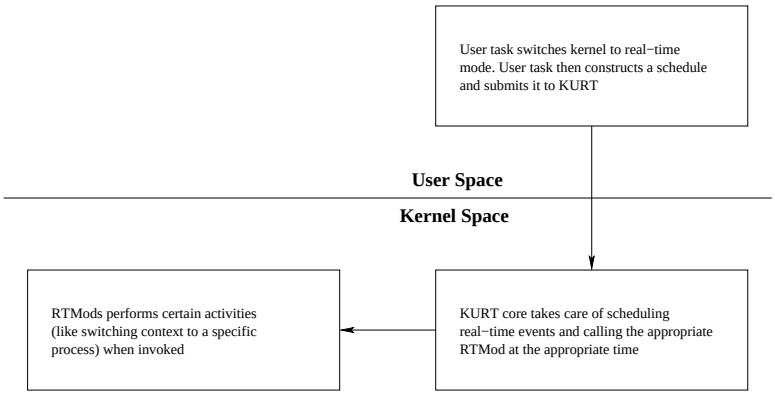

Figure 6.5: Architecture of the KURT system [53].

- **Real-time mode**. To avoid this, the kernel can be switched to real-time mode, in which only processes that are marked as real-time processes are allowed to run.

6.4 RED Linux

RED Linux (Real-time and Embedded Linux) is an ongoing project at the University of California, Irvine. Unlike RTLinux and RTAI, RED Linux does not implement a second real-time kernel. The approach here is to add new real-time capabilities to the Linux kernel, yet keep all existing Linux capability [71]. Furthermore, it is not the intention to re-implement a fully preemptive kernel, but to transform Linux into an OS with *adequate* real-time support [72].

In other words, RED Linux does not qualify as a hard real-time operating systems. Instead the project focuses heavily on providing a flexible scheduling framework that supports a variety of real-time scheduling policies.

The developers claim that one of the main motivations for RED Linux is the popularity and large user base of Linux, and the way the OS is being developed by using the open source method. The current version of RED Linux is implemented for Linux kernel 2.2.14. At the time of writing it is unclear where the project is headed, and whether RED Linux will be implemented for newer Linux kernels.

6.4.1 A General Scheduling Framework

The main focus in RED Linux is to provide real-time scheduling. To this end the developers have implemented a general and unified scheduling frame-

work that allows support of the three most popular real-time scheduling paradigms; priority scheduling, reservation based scheduling, and share allocation scheduling. Section 5.2 has a detailed discussion on these.

In the RED Linux model, the smallest schedulable model is called a job. A job is always executed once. For systems with periodic activities, a job stream defines a periodic task. Four scheduling attributes are defined for each job [71]:

- **Priority**. A job's priority defines the importance of the job relative to the other jobs in the system.
- **Start_time**. The start_time attribute defines when a job can be started. A job cannot be executed before its start time.
- **Finish_time**. The finish_time of a job is its deadline. At a jobs finish_time even if the job is not finished, it must be stopped.
- **Budget**. Budget represents the amount of execution time reserved for the job.

Figure 6.6 shows the model for the general scheduling framework of RED Linux. It is made up of two main components; the scheduling allocator, and the schedule dispatcher:

- **Schedule allocator**. The schedule allocator is responsible for setting up the scheduling attributes for new jobs. It is implemented as a process running in user space, but must usually run as the real-time process with the highest priority. The allocator defines the scheduling policy.
- **Schedule dispatcher**. The schedule dispatcher determines the execution order of the ready jobs, based on their scheduling attributes. It is implemented as a Linux kernel module. The dispatcher provides the scheduling mechanism.

The way in which the allocator sets up the attributes and the dispatcher chooses a job is dependent upon the current scheduling policy.

6.4.2 Kernel Modifications

To avoid the problem of the Linux kernel blocking, RED Linux takes the approach to make the kernel blocking delay short. This is accomplished by introducing a short dispatch time and a high resolution timer, and insert special preemption points in the kernel.

RED Linux adds three new components to Linux: the *micro-timer*, the *integrated scheduler*, and *software interrupt emulation*. This is basically a port of the software interrupt emulation used in RTLinux (Chapter 7).

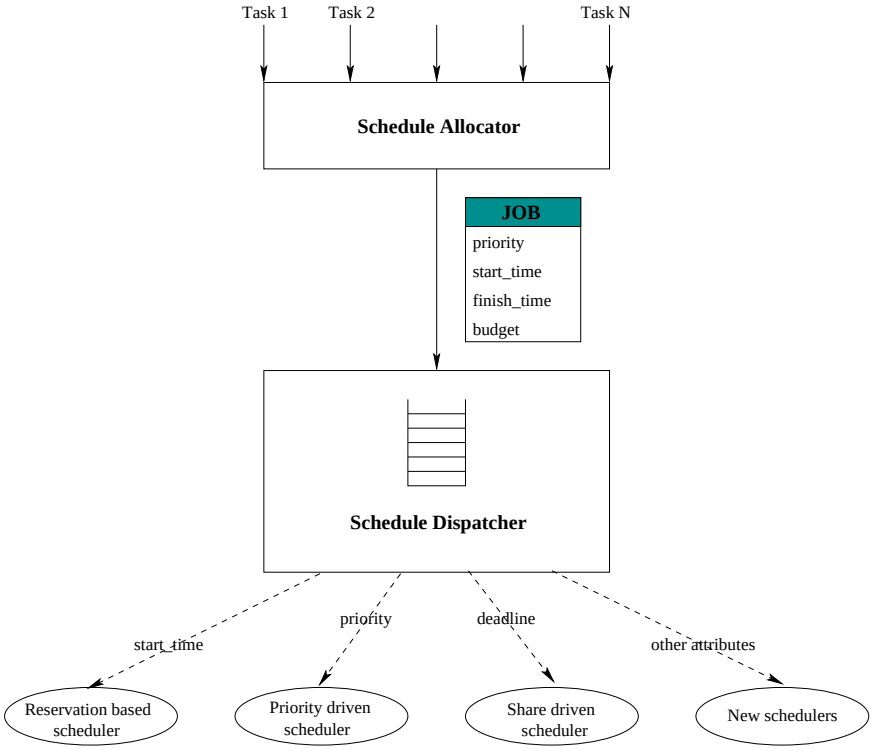

Figure 6.6: The general scheduling framework of RED Linux [71].

Instead of making RED Linux a re-entrant kernel, a small kernel preemption delay is achieved by breaking the execution of some kernel service routines into smaller code blocks. This is basically done by inserting many "preemption points". At each preemption point, the kernel checks whether there are pending real-time jobs, and if so a preemption of the kernel is forced.

6.5 Comparisons

Table 6.2 summarizes some of the more important properties that were taken into consideration when comparing the alternatives based on the material presented in this chapter.

- **Linux version**. What Linux kernel version is it available for? Apparently, this varies a great deal, and it is quite important. Newer versions of the kernel are more stable, supports more features, and has cleaner and better organized source code.

- **Hard or soft real-time**. Does it provide hard or soft real-time? The difference between the two terms are explained in Section 2.2.2.

- **Scheduling**. What kind of scheduling algorithms are available? Section 5.2 discusses scheduling. At the current stage of development, it is difficult to tell what kind of algorithm will be most suitable for scheduling of the Da CaPo threads. It would be useful to perform tests with various algorithms. Thus, the important factor is whether the scheduling module is flexible, so that alternate algorithms might be applied.

- **API**. Is the API based on a standard or is it proprietary? This is important both for portability

- **Simple or feature-rich**. What is the philosophy of the makers? If it is simple, it's normally easier to master and the source code easier to study. If it is feature-rich, the chances that it has high-level features that can be utilized is greater.

- **Type of license**. Commercial, Gnu Public License (GPL), BSD style, etc. It is definitely useful in this thesis to have the source code for the OS available. The less restrictions there are in the license, the better.

- **Documentation**. The quality and amount of documentation available differs greatly. For the purpose of the table, it has been rated good, medium, or poor.

Property	ChorusOS	RTAI	KURT	RED	RTLinux
Linux version	N/A	2.4	2.4	2.2	2.4
Hard or soft	hard	hard	soft(firm)	hard	hard
Scheduler	inflexible	flexible	inflexible	inflexible	flexible
API	POSIX+prop.	proprietary	proprietary	proprietary	POSIX
Type of license	commercial	GPL	GPL	GPL	GPL
Documentation	good	medium	poor	poor	medium
Support	poor	good	medium	poor	good

Table 6.2: Comparison of ChorusOS and Real-Time Linux variants

- **Support**. Support includes any forum where it is possible to get answers to questions regarding use of the RTOS. For the purpose of the table, it has been rated good, medium, or poor.

6.6 Summary and Conclusion

This chapter has given brief overviews of the real-time Linux variants that were candidates for this thesis. An overview was also given of ChorusOS. The real-time Linux variants were compared with each others and with ChorusOS. RTAI Linux and RTLinux are better documented, more mature, are being more actively developed and have larger user bases, than the other two. Thus, KURT Linux and RED Linux was discarded early in the selection.

RTLinux was in the end chosen over RTAI Linux, and the main reasons can be summed up like this:

- RTLinux follows a "limited features, low complexity" philosophy which seem more compatible with the idea of lightweight communication protocols.
- The RTLinux API is very close to being POSIX compliant, while for RTAI Linux, compliance is not a stated goal at all.

Chapter 7 contains a more in-depth examination of RTLinux.

Chapter 7

RTLinux

It has been the goal of many to identify what kind of OS support is required in multimedia systems. Often, it is claimed that legacy OS'es are severely lacking in this regard. As a result, there have been, and are at present, several research projects intent on providing OS support for multimedia. [39] provides an overview of, and sums up the research done in this field.

The following quote from the conclusions in [39] summarizes one of the main points of that study. In addition to being a good formulation of one of the main motivations for this thesis, it also supports, as this chapter will show, the choice of RTLinux as OS for this thesis.

> "New OS abstraction need to be developed to support a mix of applications with real-time and best effort requirements and to provide the necessary performance."

RTLinux [76] *combines* standard Linux, which has proven to be an excellent *best-effort* OS, with a tiny executive environment that provides *hard real-time* properties. By developing a version of Da CaPo that can take advantage of the hard real-time capabilities offered by RTLinux we hope to gain valuable information about running lightweight protocols in a real-time environment.

7.1 Introduction

RTLinux has been designed as a variant of Linux that provides *hard real-time* capabilities. It's being developed under the leadership of Victor Yodaiken of VYJ Associates, Limited Liability Company (LLC). The core mechanism in RTLinux has been patented, but RTLinux is released under GPL and can be freely used, modified, and redistributed under the terms of that license.

> "RTLinux is a small, deterministic, real-time operating system that is somewhat like a single POSIX process sitting on a bare

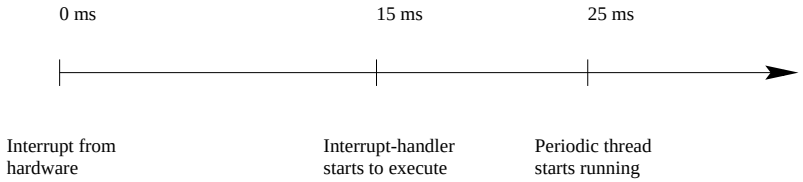

Figure 7.1: Maximum latency in RTLinux

> machine. Hard real-time applications are threads and signal handlers in this process. Linux runs as the lowest priority thread of the RTLinux kernel and it is made always preemptible. The RTLinux programming model is that anything that has strict timing requirements should be written as a thread or signal handler (interrupt handler) and whatever does not need hard real-time should go into Linux. This allows us to keep the RT side small, deterministic and as fast as the hardware will permit, while still drawing on Linux for sophisticated services and applications."
> –from the official RTLinux Frequently Asked Questions (FAQ)

7.2 Performance

Yodaiken [77] claims that the worst case time between the moment a hardware interrupt is detected by the processor and the moment an interrupt handler starts to execute is under 15 microseconds on RTLinux running on a generic Intel x86. An RTLinux periodic thread runs within 25 microseconds of its scheduled time, on the same hardware. This is illustrated in Figure 7.1.

In other words, RTLinux provides very strict guarantees for predictability of the system. Linux, even when using the built in POSIX threads and real-time functions, cannot *guarantee* anything. This is because the Linux kernel does open ended computing (Section 2.2.1), which may, theoretically, block the system indefinitely.

7.3 Adding Real-time Properties to a Desktop OS

Traditionally, RTOS'es have been very different from desktop OS'es in both design and functionality.

An RTOS is designed to be predictable, provide timing guarantees and provide low latency [56] (Section 2.2). To be able to do this, the kernel code is usually kept lean, and complex functionality is avoided. A desktop OS, on the other hand, is expected to be full-featured, and provide all the

Real-time OS	Full featured OS
Optimize worst case	Optimize average case
Predictable scheduler	Efficient scheduler
Simple executive	Wide range of services
Minimize latency	Maximize throughput

Table 7.1: Incompatible properties

services expected of a standard POSIX system: a graphical user interface, networking, file-systems, compilers, web servers, and so on.

These properties have traditionally been considered incompatible. That's why RTOS'es and desktop OS'es are separate. But the authors of RTLinux have managed to combine the properties of RTOS'es and desktop OS'es in RTLinux (Figure 7.2). Table 7.1 shows some of the "incompatible" properties.

RTLinux is a minimal real-time executive that provides a run-time environment for applications with real-time demands. Linux runs as the lowest priority RTLinux process. Furthermore, RTLinux provides ways for real-time applications to communicate with non-real-time applications. Thus, RTLinux is able to provide RTOS services and, at the same time provide all the services of a full-featured OS, on the same system.

7.4 Concept

The basic idea is to make Linux run under the control of a real-time kernel [77]. When there is real-time work to be done, the RTLinux kernel runs one of it's real-time threads. When there is no real-time work to be done, the RTLinux kernel lets the Linux kernel run. When Linux is scheduled to run, it runs as it normally would, and schedules Linux kernel and user threads as usual. Linux is the lowest priority thread of the RTLinux kernel.

The RTLinux kernel puts itself "between" the Linux kernel and the interrupt generating hardware, as shown in Figure 7.3. This is essential, because the Linux kernel disables interrupts in critical parts of the kernel code. If an interrupt for the Linux kernel or a thread running under Linux arrives, it is simply passed on to the Linux kernel, to be handled in a regular fashion. But if there's an interrupt for a thread running under RTLinux, it is intercepted and handled by that thread. Any running thread with lower priority running under RTLinux, including the Linux kernel, will at this point be preempted.

The real-time kernel itself is non-preemptable, but it's routines are very small and fast, so this does not cause big delays. For example, on a Pentium 120, the maximum scheduling delay would be less than 20 microseconds [4].

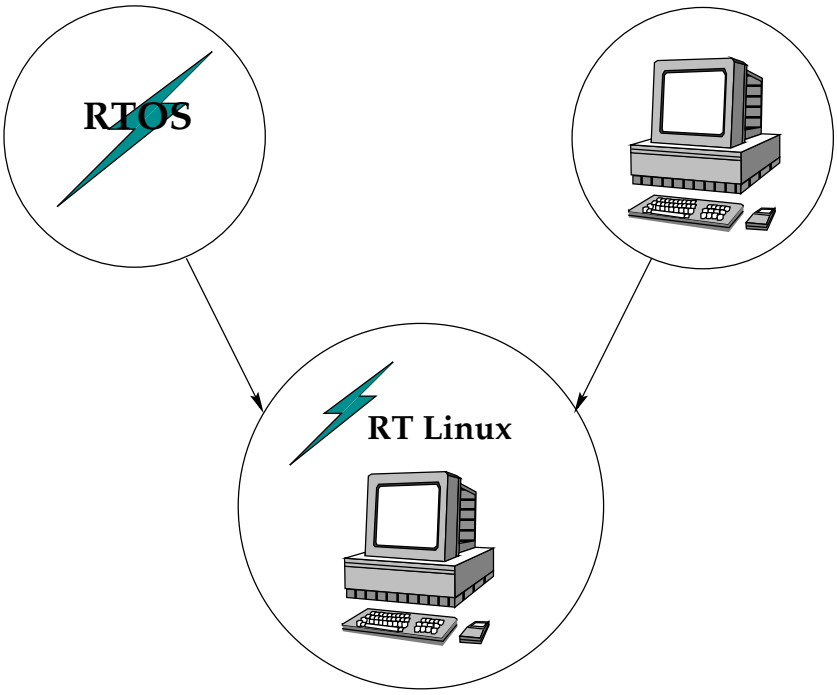

Figure 7.2: Adding real-time properties to a desktop OS.

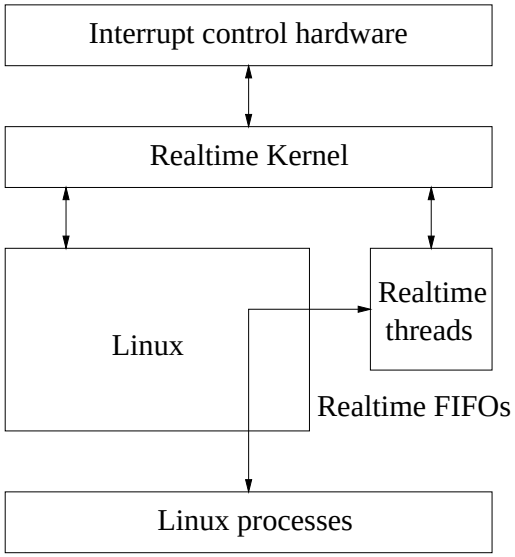

Figure 7.3: Block level design of RTLinux

Module	Short description
`rtl`	Core module
`rtl_time`	Architecture-independent clock support
`rtl_fifo`	RT-FIFOs for IPC with non real-time processes
`rtl_sched`	Default scheduler
`rtl_posixio`	Provides POSIX style IO
`rtl_debug`	Enables debugging with GDB
`mbuff`	Shared memory buffer driver
`psc`	User-level real-time module
`rt_com`	Serial port driver

Table 7.2: The standard RTLinux components

7.5 Architecture

RTLinux is implemented as Linux *loadable kernel modules*. In order to explain this concept, the following overview of the architecture of RTLinux is preceded by a description of the Linux kernel feature called *loadable kernel modules*.

7.5.1 Linux Loadable Kernel Modules

This Linux kernel feature enables dynamical modification of the Linux kernel. That is, parts of the kernel may be added or removed while the kernel is running. When a Linux kernel module is loaded there is nothing that makes it different from the rest the static parts of the kernel except that it may be removed from the kernel.

For example, if a cdrom player in a computer is used only occasionally, the driver for cdrom players could normally not be part of the kernel, but be loaded into the kernel when it was needed.

RTLinux is structured as a small core component and a set of optional components. The components are loaded into the Linux kernel as standard Linux kernel modules. The modules that come with the RTLinux are listed in Table 7.2. Of all these modules, only the `rtl` (core) module is mandatory. Although, for most situations one will also need to load the `rtl_time` (timing support) and `rtl_fifo` (RT-FIFO) modules. A scheduler module must be loaded, but it does not have to be the included `rtl_sched`. The other modules provide optional features. In addition there are other modules contributed by people outside the RTLinux development team.

7.5.2 RTLinux Components

At the time of writing this, the RTLinux documentation lack good descriptions of the internal working of the various modules. Most of the information

in this section is based on studies of the source code for RTLinux.

- `rtl` is the core module for RTLinux.
- `rtl_time` controls the processor clocks and exports an abstract interface for connecting handlers to clocks.
- `rtl_fifo` implements a FIFO mechanism for IPC, through a device layer, between RTLinux threads and Linux processes. (more on this in Section 7.7.1.
- `rtl_sched` is the default scheduler for RTLinux. It is a priority driven scheduler, which is intentionally kept small and simple. See Section 7.6 for more on CPU scheduling in RTLinux.
- `rtl_posixio` provides a file system like interface to drivers. The basic operations for drivers are to register and un-register devices. The module also provides `open`, `close`, `read`, `write`, `ioctl` and `mmap` calls. The operations are typically used in device drivers, like the FIFO driver and the shared memory driver. This module is not of interest in this thesis. But it will be of interest at a later stage, when writing device drivers for Da CaPo.
- `rtl_debug` enables source level debugging of RTLinux using Gnome Debugger (GDB). The module supports all normal GDB functionality, including threads support. It also functions as crash protection. It catches and stops any bugs that would normally crash the system.
- `mbuff` implements a shared memory buffer. The buffer can be shared between both RTLinux threads and Linux processes (Section 7.7.2).
- `psc` is a module that supposedly implements a way to run real-time threads from user-level applications. This could have been very useful in this thesis. But at the time of writing it seems to be incomplete and/or immature, and was thus discarded as an option.
- `rt_com` is an included module that is a driver for 8250 and 16550 families of Universal Asynchronous Receiver/Transmitter (UART)s, which are the microchips commonly used in PCs to control serial ports. It is not of interest in this thesis.

Section 9.1.1 has some related information on how RTLinux is installed on a Linux system.

Mode	Running threads
rt	RTLinux real-time threads
kernel	Linux kernel threads
user	Linux non-kernel threads

Table 7.3: Modes of execution in RTLinux.

7.6 CPU Scheduling

In RTLinux a scheduler and all the threads it schedules are considered to be one process. There can only be one scheduler per processor. Each scheduler can schedule multiple of threads. In other words, there can only be one real-time process per CPU, but that process can own an unlimited number of threads, all using the same scheduler.

7.6.1 Modes of Execution

There are three execution modes defined in RTLinux. They are, as shown in Figure 7.3, *user*, *kernel* and *rt*. User and kernel modes are the standard Linux user and kernel modes. When the Linux thread is running, the processor must be in one of these modes. When a real-time thread or interrupt handler is running, the processor is in rt mode.

7.6.2 Threads

RTLinux threads are POSIX style threads, but with some limitations not usually found in POSIX systems. RTLinux follows what the POSIX standard identifies as a "Minimal Real-time System Profile" [49]. Section 7.9 discusses the full implications of this. As far as threads go, this definition allows that a hard real-time system like RTLinux does not need to support multiple processes (as opposed to multiple threads).

In practice, RTLinux supports an unlimited number of threads. The RTLinux components (Section 7.5.2) and all real-time threads, including the Linux kernel thread, belong to one single process. That is, one cannot have more than one real-time process per CPU. It is possible to run several real-time applications (user created real-time modules), simultaneously. But, RTLinux will see all threads in all real-time applications as belonging to the same real-time process. Table 7.4 summarizes the limitations on the number of threads and processes under RTLinux.

Another important issue is that real-time threads cannot be created in real-time. Threads and their properties must be statically defined. This is due to the way RTLinux and it's threads are created - as Linux kernel modules. When Linux kernel modules are loaded, the kernel must do some

Number of	Limited to
threads	unlimited
processes	limited to one per CPU
user-created modules	unlimited, all threads belong to one process

Table 7.4: Summary of limitations on threads and processes in RTLinux.

operations to make space in memory for them and link them with the kernel. This implies some overhead that is inherently non-real-time.

7.6.3 Modular Scheduler

RTLinux does not mandate the use of one particular scheduler. It is quite simple to replace the default scheduler with an alternative one. RTLinux was designed with this kind of modularity in mind. Schedulers are implemented as Linux loadable kernel modules. Users with special scheduling requirements are encouraged to write their own [77].

The scheduling module in RTLinux treats a scheduler and a collection of real-time threads as a single POSIX process, with threads corresponding to POSIX threads [77]. Figure 7.4 illustrates how RTLinux threads corresponds to Linux, and Linux processes and threads.

7.6.4 Priority Driven Scheduler

The default scheduler is purely priority driven and preemptive. At any time it simply picks the highest priority runable thread and runs it. Whenever a thread with a higher priority than the one running becomes runable, the running thread is preempted and the higher priority thread gets dispatched. The Linux kernel itself is handled as a thread with the lowest priority. Thus, whenever there are real-time threads ready to run, the Linux kernel will have to wait for those threads to finish.

The code for the default scheduler is kept short and simple. The sentiment in the user community seems to be that this scheduler is extremely efficient and suitable for most needs[1]. It's also a well suited as a starting point for modification or implementation of schedulers using other algorithms.

7.6.5 Other Schedulers

Both the EDF and the RM scheduling algorithms (Section 5.2) have been implemented for RTLinux. However, they were implemented for an early version of RTLinux, and uses the old style API. Unfortunately, it has not

[1] This assumption was made after following numerous discussions in various mailing lists

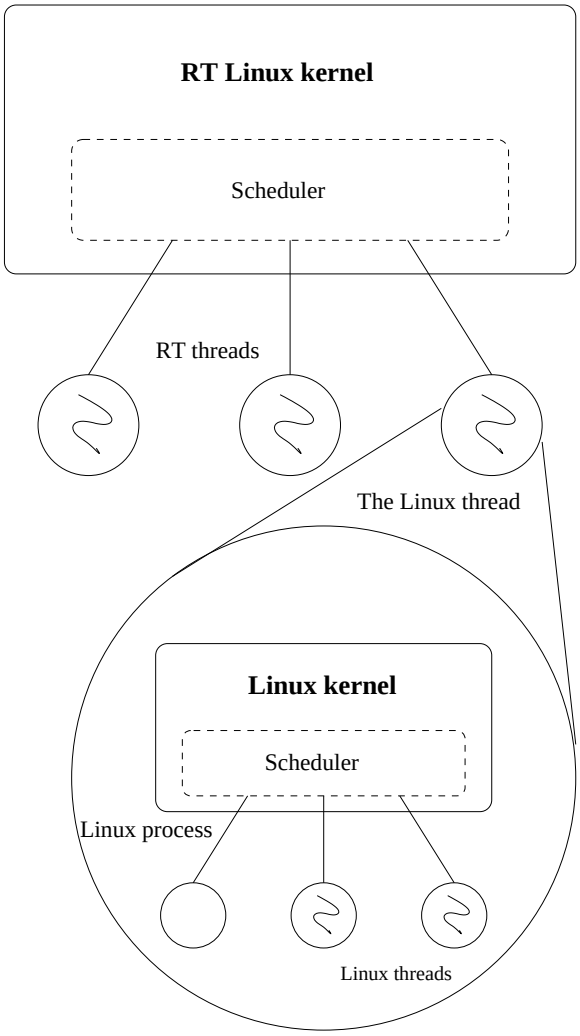

Figure 7.4: Linux scheduled under RTLinux. The RTLinux kernel runs on top of everything. It schedules all the real-time threads. The Linux system, kernel and all processes included, is considered to be one single such real-time thread.

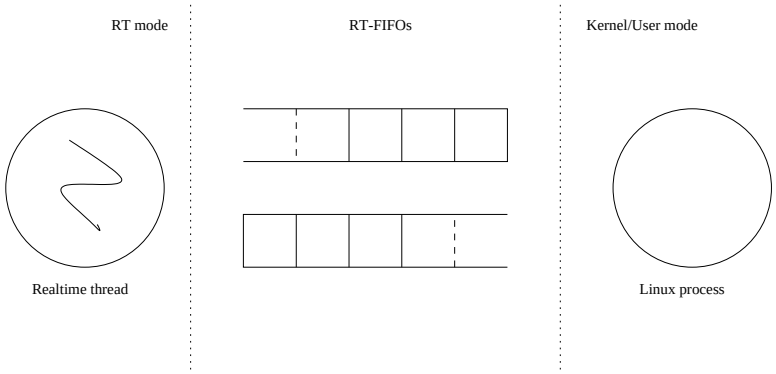

Figure 7.5: RT-FIFOs

been possible to get the source code for any of these alternative schedulers. Thus, it has not been possible to study them in any detail at all. However, in the RTLinux user community, it is understood that they will have to be re-implemented to be useful with the newer versions of RTLinux. At least for the RM scheduler it is likely that this will be done in the near future, as the author has stated so recently in an email to the user community.

7.7 Inter Process Communication

A special FIFO concept have since the first release of RTLinux been the primary means of IPC in RTLinux. A shared memory module was later contributed. The shared memory module has matured rapidly, and is now included in the RTLinux distribution. Both RT-FIFO's and shared memory can be used for communication between both real-time and non-real-time threads.

Some forms of POSIX style IPC is expected to appear in later versions of RTLinux. POSIX style signals have been partly implemented in RTLinux version 3.

7.7.1 RT-FIFO

The module `rtl_fifo` implements a simple FIFO mechanism for communication between real-time threads and Linux processes. Figure 7.5 illustrates the concept.

The real-time threads interface to RT-FIFOs include creation, destruction, reading and writing functions. Reads and writes are atomic and do not block. Non-blocking avoids the priority inversion problem [2]. Linux processes see RT-FIFOs as ordinary character devices. They are reachable

Interface function	Short description
`rtf_create`	create RT-FIFO
`rtf_create_handler`	install rt-fifo handler
`rtf_destroy`	destroy RT-FIFO
`rtf_get`	read from RT-FIFO
`rtf_put`	write to RT-FIFO
`rtf_flush`	empty RT-FIFO
`rtf_make_user_pair`	make bidirectional FIFO
`rtf_link_user_ioctl`	install `ioctl`[2] handler for RT-FIFO

Table 7.5: RT-FIFO functions in RTLinux 3.0

through */dev/rtfx*, where x is a number. The real-time threads access the RT-FIFOs via a POSIXIO interface.

Like real-time threads, and for the same reasons, RT-FIFO's cannot be allocated in real-time. Although not as seriously limiting as with threads, this is an issue that must be taken into consideration if RT-FIFO's are to be used in the design.

In addition, the maximum number of RT-FIFO's must be statically declared when compiling the Linux kernel. The standard setting for this number is so high that it should be sufficient for most needs, so this is not considered to be a problem.

7.7.2 Shared Memory

The `mbuff` module and */dev/mbuff* is intended to be used as a shared memory device. It is especially well suited to be shared between real-time threads and Linux processes. Memory is allocated in the kernel. It is possible to map from user space. And it is reachable from real-time threads via a POSIXIO interface. This memory can not be swapped out, and as such is well suited for real-time applications.

7.8 Application Programming Interface

Documents describing the API has not been easily available. A study of various sources (source code, man-pages, and other documents) was necessary in order to get a good overview of all the available functions. This section and the included tables present the results of that study.

Since shortly after the release of the first version of RTLinux, it has been a goal for the developers to make the API conform to the POSIX standard [76]. RTLinux 3.0 provides a subset of the POSIX threads interface. It follows the POSIX 1003.13 "Single Process/Minimal Real-time System" (PE51) model, which is described in [49]. RTLinux also has some support

for POSIX semaphores (ref), and POSIX mutexes (ref). And in addition, RTLinux also has a set of non-portable functions.

All functions work as described in POSIX and the Single UNIX Specification. Except `pthread_create`, which can only be called by the Linux kernel (Section 7.9).

- The list of RTLinux specific functions is shown in Table 7.7. The source for this list is the documentation and man-pages that comes with the RTLinux distribution.
- Table 7.6 lists the available functions from the POSIX interface, except the condition variable functions and the semaphore functions. The source for this list is the documentation that comes with the RTLinux distribution and [34].
- The interface to the RT-FIFO module is listed in 7.5. The source is the documentation and man-pages that comes with the RTLinux distribution.
- POSIX style condition variables are supported via the functions listed in Table 7.8. The source for this list is also the documentation that comes with the RTLinux distribution and [34].
- POSIX style semaphores are also supported. Table 7.9 lists the POSIX interface functions available. Again the source the list is the documentation that comes with the RTLinux distribution and [34].
- Some functions are available in RTLinux for backwards compatibility with the first version of RTLinux. All these functions have been replaced by newer functions that conforms to the POSIX standards. These functions are not of interest in this thesis.

As work progressed on this thesis, the documentation describing the API for RTLinux has improved a lot. This new documentation has been used to confirm and update the tables.

The RTLinux API is full-featured. By informal evaluation it was decided that it is sufficient for an implementation of Da CaPo.

7.9 Problems and Limitations

The study of RTLinux, which this chapter is a result of, has uncovered a series of problems and limitations inherent in the OS. This section summarizes the relevant issues that must be taken into consideration in the work on the design.

Interface function	Short description
clock_gettime	get current value for a clock
clock_settime	sets a clock
time	get time in seconds since Epoch
usleep	suspend calling thread
nanosleep	high resolution sleep
clock_nanosleep	high resolution sleep
sched_get_priority_max	get max priority limit
sched_get_priority_min	get min priority limit
pthread_self	get ID of calling thread
pthread_attr_init	initialize thread attribute
pthread_attr_setstacksize	set stacksize attribute
pthread_attr_getstacksize	get stacksize attribute
pthread_attr_setschedparam	set schedparam attribute
pthread_attr_getschedparam	get schedparam attribute
pthread_attr_setdetachstate	set detachstate attribute
pthread_attr_getdetachstate	get detachstate attribute
sched_yield	yield processor
pthread_setschedparam	set a threads scheduling parameters
pthread_getschedparam	get a threads scheduling parameters
pthread_create	create thread
pthread_exit	clean termination of thread
pthread_cancel	cancel execution of a thread
pthread_setcancelstate	set and get cancel-ability state
pthread_setcanceltype	set and get cancel-ability type
pthread_join	wait for thread termination
pthread_kill	send a signal to a thread
pthread_mutexattr_setpshared	set process-shared attribute
pthread_mutexattr_getpshared	get process-shared attribute
pthread_mutexattr_init	initialize mutex attributes
pthread_mutexattr_destroy	destroy mutex attributes
pthread_mutexattr_settype	set a mutex type
pthread_mutexattr_gettype	get a mutex type
pthread_mutexattr_setprotocol	set protocol attribute
pthread_mutexattr_getprotocol	get protocol attribute
pthread_mutexattr_setprioceiling	set prioceiling attribute
pthread_mutexattr_getprioceiling	get prioceiling attribute
pthread_mutex_init	initialize a mutex
pthread_mutex_destroy	destroy a mutex
pthread_mutex_lock	lock a mutex
pthread_mutex_trylock	lock mutex, return if already locked
pthread_mutex_unlock	unlock a mutex
sysconf	get configurable system variables
uname	get name of current system

Table 7.6: POSIX functions in RTLinux 3.0

Interface function	Short description
sigaction	POSIX[3] signal handling functions
clock_gethrtime	get high resolution time
gethrtime	get high resolution time
pthread_attr_setcpu_np	set the CPU pthread attribute
pthread_attr_getcpu_np	get the CPU pthread attribute
pthread_wait_np	suspend thread until next period
pthread_delete_np	delete thread
pthread_setfp_np	allow floating-point operations in thread
pthread_make_periodic_np	mark thread periodic
pthread_suspend_np	suspend thread
pthread_wakeup_np	wake up suspended thread
rtl_request_irq	install interrupt handlers
rtl_free_irq	remove interrupt handler
rtl_free_soft_irq	remove software interrupts handler
rtl_get_soft_irq	install software interrupts handler
rtl_global_pend_irq	schedule Linux interrupt
rtl_hard_disable_irq	disable interrupts
rtl_hard_enable_irq	enable interrupts
rtl_allow_interrupts	allow CPU interrupts
rtl_no_interrupts	save and disable interrupts
rtl_restore_interrupts	restore interrupts
rtl_stop_interrupts	disable interrupts
rtl_printf	print formatted output
rtl_getschedclock	get scheduler clock
rtl_setclockmode	set clock mode
rtl_getcpuid	get processor id

Table 7.7: Non-POSIX functions in RTLinux 3.0

Interface function	Short description
pthread_condattr_setpshared	set the process-shared attributes
pthread_condattr_getpshared	get the process-shared attributes
pthread_condattr_init	initialize condition variable attributes
pthread_condattr_destroy	destroy condition variable attributes
pthread_cond_init	initialize condition variable
pthread_cond_destroy	destroy condition variable
pthread_cond_wait	wait on a condition
pthread_cond_timedwait	wait on a condition
pthread_cond_broadcast	broadcast a condition
pthread_cond_signal	signal a condition

Table 7.8: POSIX condition variable functions in RTLinux 3.0

Interface function	Short description
`sem_init`	initialize semaphore
`sem_destroy`	destroy semaphore
`sem_getvalue`	get semaphore value
`sem_wait`	lock semaphore
`sem_trywait`	lock semaphore
`sem_post`	unlock semaphore
`sem_timedwait`	lock semaphore with time-out

Table 7.9: POSIX semaphore functions in RTLinux 3.0

- **Threads cannot be created in real-time**. This is perhaps the most serious limitation in RTLinux. It is very relevant in this thesis, since the principle of dynamically configurable protocols suggests that run-time creation of threads will be needed. This design problem is treated in Section 8.3.1.

- **RT-FIFO's cannot be allocated in real-time**. RT-FIFO's are used in the resulting design in this thesis. The problem is solved in a way similar to the solution for threads, in Section 8.3.4.

- **Memory cannot be allocated in real-time**. Memory blocks must also be statically allocated. This can be solved by allocating a sufficient size memory block at start time, and then provide a memory allocation system that allocates and frees memory from that memory block. This solution has been designed, as seen in Section 8.3.2, and implemented.

- **Incomplete POSIX real-time API conformance**. The real-time POSIX API in RTLinux is quite rich, and is rapidly evolving further. Even so, it's unlikely that it'll ever conform completely to the POSIX 1003.13 real-time standard.

  It is useful to conform to this standard because it provides the only widely accepted and used API for real-time, and because it facilitates connection and software between real-time and non-real-time POSIX systems.

  At the same time it is practically impossible for a hard real-time system to implement all the features described in the standard. A straight-forward implementation of, for example, a POSIX file-system imposes open ended computing and resource commitments that a hard real-time system simply cannot provide.

  The problem is that it will be difficult to do an implementation which is easily portable. The only solution is to stick to standardized API calls as much as possible.

- **One real-time process per processor**. As described in section 7.6.2, there can only be one scheduler/process per CPU. This is not often a problem when writing hard real-time applications. Such applications are usually short pieces of code to control hardware in some fashion. But for more complex programs, like multimedia applications, this is a limitation.

  However, in the Da CaPo design described in Section 8.1, this problem has been solved by making all components threads of the same process.

7.10 Summary

This chapter contains information about RTLinux gathered from various sources. The documentation available for RTLinux, although constantly improving, is still incomplete and not always up to date. The documentation alone is insufficient in order to effectively use RTLinux. The open source code more than compensates for this. Especially since the authors of RTLinux strive to keep the code as short and simple as possible. And, when, as in the case of this thesis, one wants to experiment with OS features, having the source code available is invaluable. Open source combined with excellent support from a thriving user community is far superior to many commercial alternatives.

The conclusion so far is that using RTLinux as a base for the thesis should be both possible and yield interesting results. The contents gathered in this chapter is both necessary and sufficient to start work on the design and then the implementation of a multi-threaded version of Da CaPo for RTLinux.

Chapter 8

Design

This chapter gives a "top-down" presentation of the design process. It starts by giving an an overview of the main components in the design. It describes their general functionality and how they interface with the components that are outside the scope of this thesis. Later in the chapter, there are presentations of the more detailed low-level design considerations, and discussions on why the specific design decisions were made.

8.1 Overall Design

When developing the design, the main goal was to make its organization logical, and as simple as possible. A decision was made that can be summed up as a fundamental design rule:

> The modules are the active components. Administrative tasks are implemented as supporting functions, which are called by specially designed modules with administrative responsibilities.

This simply means moving all the active administration components into modules. Mostly into the modules in the configuration manager.

8.1.1 Components

Figure 8.1 shows the the main components and basic relationships, traffic flow and control flow, between them. There are three horizontal lines separating parts of the figure. The two top lines separate, from top-down, user-space, kernel space, and real-time space. The "spaces" represent the various modes of execution in RTLinux (Section 7.6.1). This separation is very important in the design, because both implementation and behavior of programs is different in the various modes. It also shows which of the components will be running in real-time.

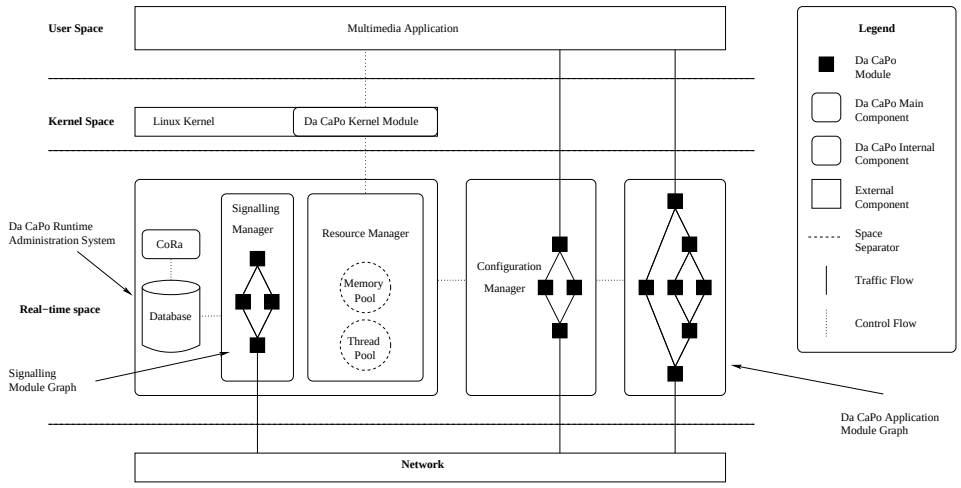

Figure 8.1: Component overview and basic relationships in Da CaPo 2 for RTLinux.

- **Multimedia application**. An application that is to use Da CaPo and take advantage of QoS features need to be adapted for this. One way to do this is to create a library of functions. The development of such a library is beyond the scope of this thesis. For the testing of the code from this thesis, a simple simulation of the application will suffice.

- **Da CaPo kernel module**. This component is optional. It is a possible solution to have a direct control flow between the application and the Da CaPo run-time administration system, instead of via a Da CaPo kernel module. Another option is to implement a kernel module that alters the network stack in the Linux kernel so that Da CaPo is used in it's place. This component is outside the scope of the thesis.

- **CoRA and database**. These components are outside the scope of this thesis. Their functionality is not relevant in the design of the components that are in the scope of the thesis.

- **Signaling manager**. Also outside the scope of this thesis. Ingvild Kalleberg has written a thesis [23] that discusses signaling in Da CaPo.

- **Resource manager**. This component has been added in order to accommodate for some of the restrictions imposed by the use of RT-Linux. In RTLinux, memory blocks and threads cannot be allocated in real-time (Section 7.9). This problem has been solved by reusing threads and memory. The solution is described in Sections 8.3.1 and 8.3.2.

- **Configuration manager**. The configuration manager consists of a static module graph composed of modules that are specially made to enable the applications and Da CaPo to negotiate QoS. The functionality is located in these modules. It is not a goal in this thesis to create modules for this. However, the design must accommodate for the fact that some modules need to communicate with the other components of Da CaPo. There can be multiple configuration managers running simultaneously, for example one for each connection.

- **Da CaPo application module graphs**. The problems and solutions are discussed in Sections 8.2.3 and 8.3.4. There can be multiple module graphs simultaneously, one or more per configuration manager. In other words, one or more per connection.

The configuration manager and the application module graph is set up as needed when a Da CaPo session is started. All the other components must be started before a session can be initiated. This can be done at three times:

1. At system start up (when booting the OS).

2. Manually, at any time after system start up and before session is initiated.

3. When a session is initiated. This solution requires some mechanism for detecting a session initiation when the Da CaPo system is not running. It also causes a lag in the session initiation.

Regardless of when it is started, the process is the same. Figure 8.2 illustrates and describes the process. Figure 8.3 shows the chain of events that happen when a session is initiated. Figure 8.4 illustrates how threads and data queues are allocated from the pools as a module graph is build as part of session initiation. The design is explained in detail in Sections 8.3.1 and 8.3.2.

8.1.2 Data and Control Flows

Figure 8.3 shows how the connection management and data protocols are setup when a Da CaPo session is initiated by an application at one of two hosts. Figure 8.5 shows the control and data flows after that initial setup step.

The control flows:

- **Between application and connection manager**. The connection manager module graph has an A-module that transmits and receives control information to and from the application. Chapter 8.3.4 describes the design of the A-module. The connection manager presents

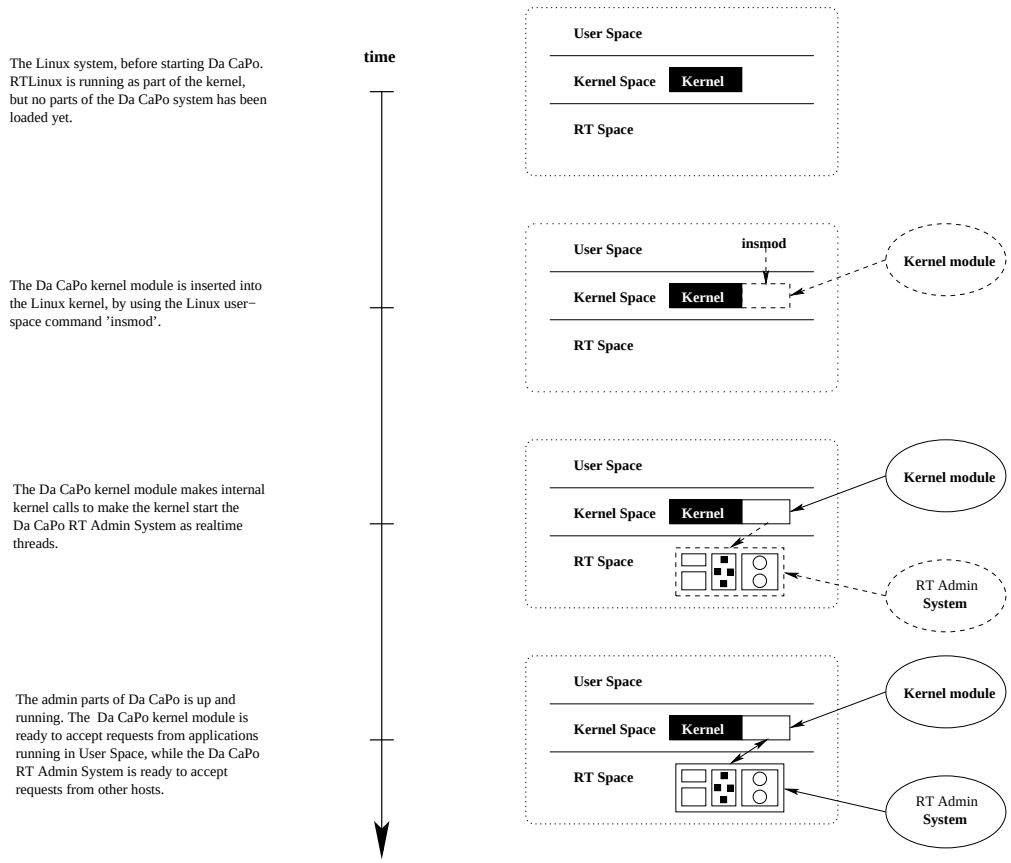

Figure 8.2: Starting Da CaPo for RTLinux.

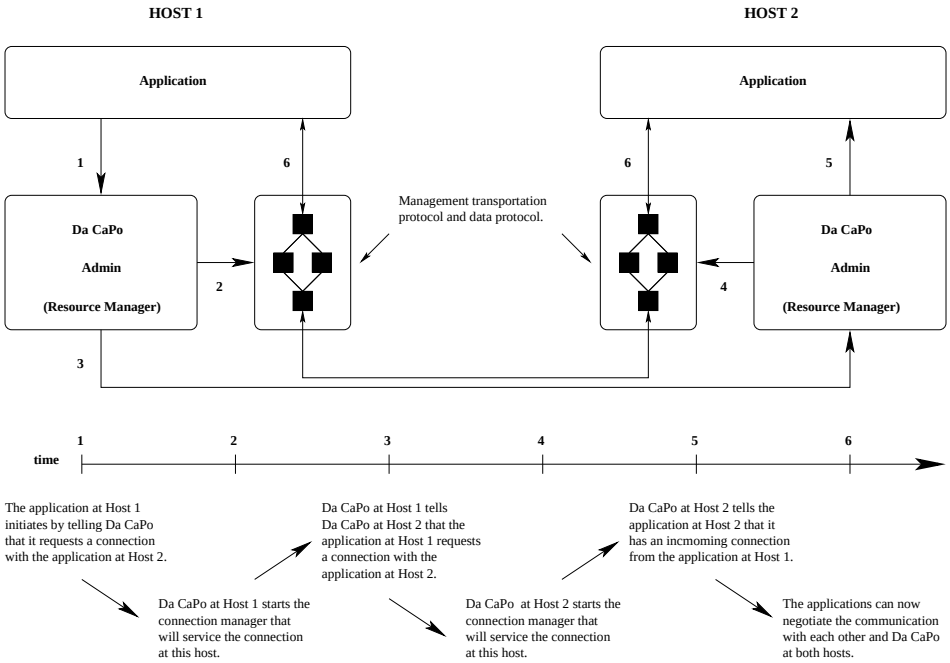

Figure 8.3: Time-line for establishing connection in a Da CaPo session.

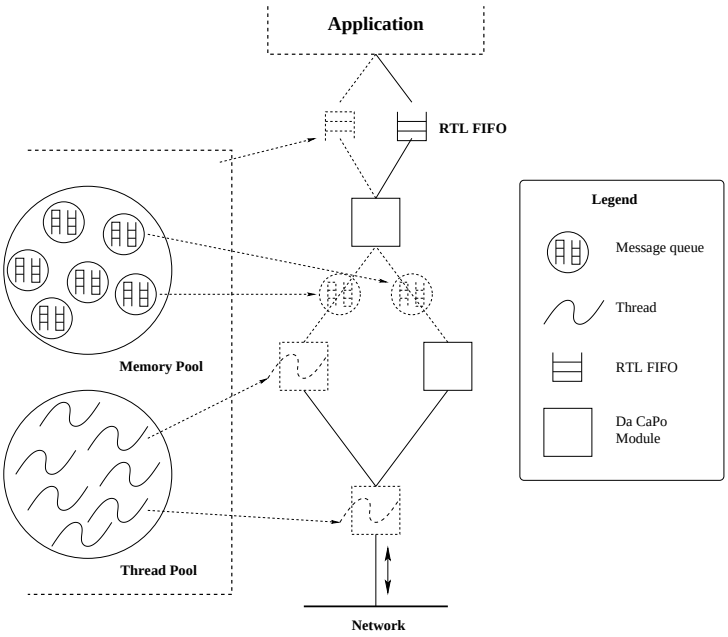

Figure 8.4: Starting a Da CaPo session. Threads and memory blocks are allocated from pools.

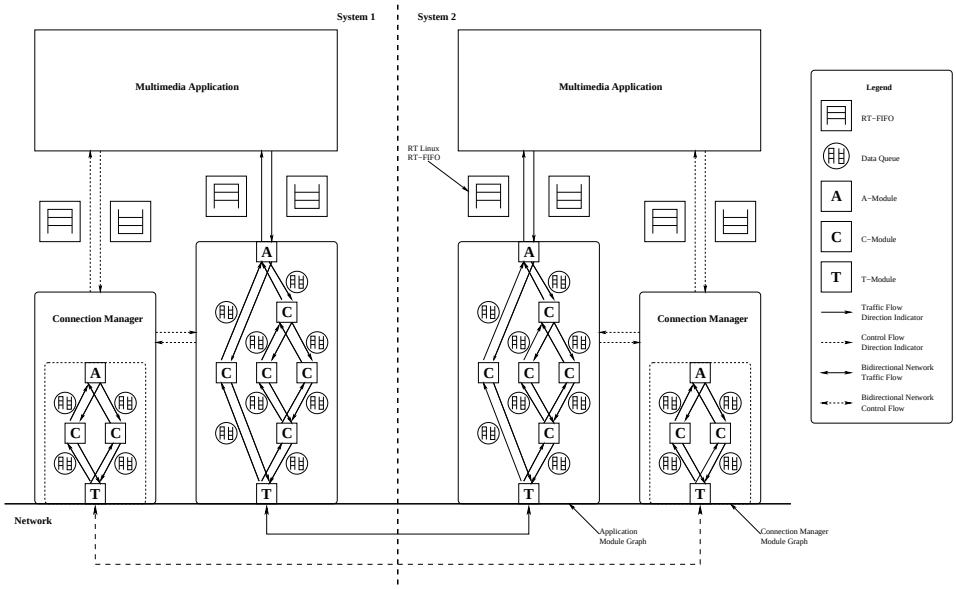

Figure 8.5: Data and control flow in Da CaPo between two multimedia applications on different hosts.

the application with a unified interface for the purpose of QoS negotiation. The only time the application communicates control information with other parts of Da CaPo is when it requests a session, before a connection manager has been set up.

- **Between connection manager and application module graph(s)**. The connection manager is responsible for setting up, re-configuring, and bringing down its application module graph(s). Modules can be created with the capability to return monitoring information, hence the control flow from application module graph to connection manager. The application module graph is a logical entity, in reality, the connection manager has direct control over each module in it, and via the modules, each data queue in it. The modules are designed with interfaces that allows this (Section 8.3.4).

- **Between connection managers on different hosts**. This is were the data for the QoS negotiation between the hosts is transmitted over the network. A standard T-module in the connection manager module graph facilitates the transmission.

- **Between connection manager and Da CaPo run-time administration system** (shown in Figure 8.1). This flow includes initiating call from the signaling manager and all calls to facilities provided by the

Da CaPo run-time administration system to the connection manager.

The data flows:

- **Between application and its module graph(s)**. The actual data the applications transmits and receives.
- **Between module graphs on different hosts**. The data in transmitted after it propagated through the modules in the application module graph. It may now look significantly different, but when it has been propagated through the module graph on the receiving end, the receiving application should receive the data as intended by the sending application.

8.2 Da CaPo 2

This section presents some of the design issues specifically related to Da CaPo. Each problem that is discussed is introduced by a description of the problem. Next, a solution, or possibly several alternative solutions is presented. When there are alternative solutions, they are examined and choice made based on the findings.

8.2.1 Threads

The current version of Da CaPo (Section 4.3) is not multi-threaded. Thus, when designing a multi-threaded version, a major design consideration is how to organize the threads. That is, what parts of Da CaPo should be implemented with independent threads of execution. The choice of design should result in an organization of threads that:

1. Provides a high level of control in module graph. This is an important point because the design aims to exploit the resource management capabilities of RTLinux.
2. Provides feedback for monitoring purposes.
3. Is logical in the sense that it constitutes an organization which is conceptually easy to understand.
4. Is simple to implement for RTLinux.
5. Enables a design without the "lift" control components. Chapter 8.2.2 explains why this is desirable.

Viable alternatives have been thoroughly examined by Bjørn Volden in [69]. Most of this section is a recapitulation of the findings made in his thesis.

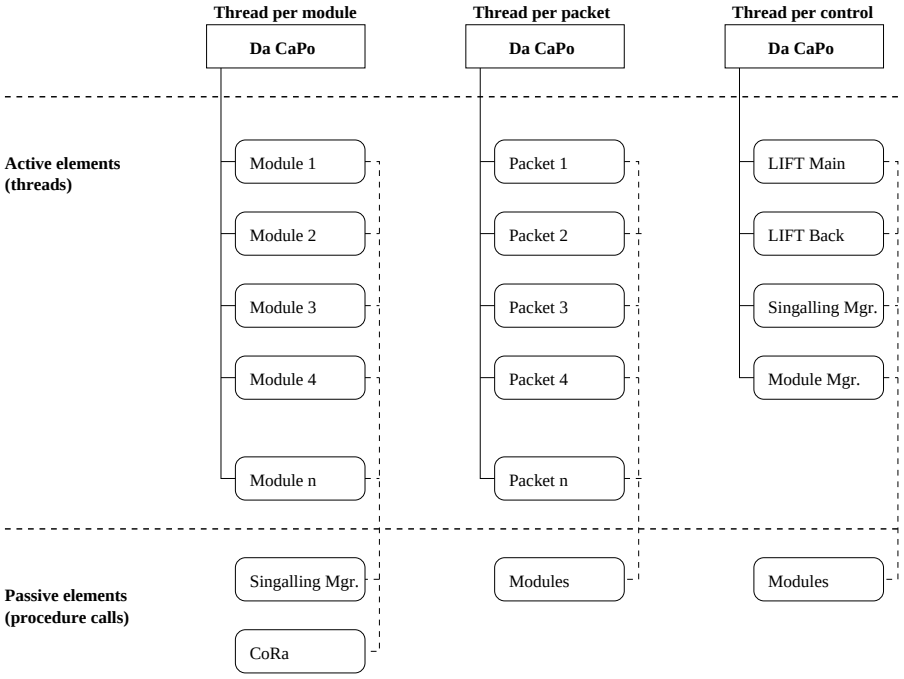

Figure 8.6: Summary of the alternative organization of threads.

Alternatives

The alternatives discussed in [69] are summarized in Figure 8.6. It is a simplified figure, but it explains the concepts.

- **Threads per module**. In this model, the modules themselves are the active components. They take over the work of propagating packets through the module graph. The other Da CaPo components are implemented as passive elements, procedures that are called from the modules. The arguments for and against this model are listed in Table 8.1.
- **Threads per packet**. A thread is assigned to each packet. That thread is responsible for propagating the packet through the module graph. The modules and control components are passive, and called as procedure calls by the packet threads. The arguments for and against this model are listed in Table 8.2.
- **Threads per control**. In this model, the control components are active threads. It is the model that which is matches the current implementation closest. A special control component (lift algorithm) is responsible for propagating packets through the module graph. The arguments for and against this model are listed in Table 8.3.

PROS	Logical design. Easy to implement. Feedback for monitoring can be implemented in modules. High level of control of module graph. Low complexity in control components. Low complexity in parallel modules. Lift components not needed.
CONS	High complexity in modules. More overhead in modules

Table 8.1: Arguments for and against the threads per module model.

PROS	Lift components not needed.
CONS	Abstract design. Difficult to implement. High complexity in parallel modules.

Table 8.2: Arguments for and against the threads per packet model.

PROS	Logical design Simple to implement. Low complexity in modules.
CONS	Lift components must be present. Low level of control of module graph. High complexity in control components.

Table 8.3: Arguments for and against the threads per control model.

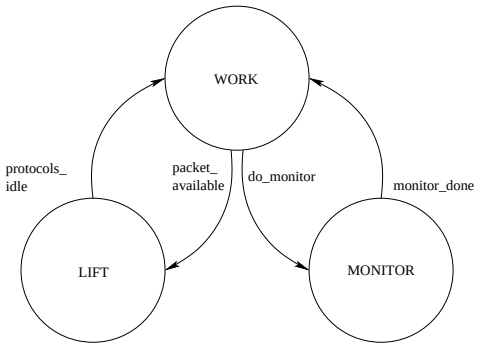

Figure 8.7: Example showing how data cells are shared between parallel modules.

Conclusion

As in [69], the **threads per module** solution is considered to be the most likely to satisfy the requirements listed. It is the most elegant solution in terms of implementation for RTLinux. The "lift" component is not needed. The fact that each module has its own thread enables a high level of control of the module graph, and possibilities for detailed monitoring information.

8.2.2 Lift Algorithm

"Lift - the packet transport algorithm" is a feature of the current implementation of Da CaPo. It is a result of the fact that that implementation was for an OS that did not support threads, hence entire Da CaPo is implemented as one single process. This description of it is a quote from [16]:

> "Packet forwarding within protocols is done by a special data transport algorithm. On protocol setup, every path of a protocol is divided into *execution stage*, which form atomic units of the transport algorithm, i.e. every time the algorithm activates a protocol path it executes one stage."

Da CaPo operates in three states, *WORK*,*LIFT* and *MONITOR*. In the WORK state, all tasks that are unrelated to moving packets and monitoring are executed. When a packet is ready to be moved, or a new packet becomes available, Da CaPo enters the LIFT state. The lift-algorithm is called and it moves packets as necessary and returns to the WORK when it is done. The MONITOR state is entered at regular interval to do tasks that collects statistics about the state of modules etc. Figure 8.7, which is taken from [25], illustrates the three states.

Conclusion

In Dynamic Configuration of Protocols 2 (Da CaPo 2), the functionality of the lift-algorithm will be distributed between all the modules. Each module will be responsible for fetching and delivering packets to and from themselves. A module will not move a packet to or from the next module(s) in the graph. When packets are between modules they will be in data queues. Thus, modules will move packets to and from data queues.

Modules will not *block* in the same sense as in the old design. Because modules are threads, they will be able to put themselves to sleep when there is no more work to be done. Simple mechanisms that are inherent in RTLinux enable function calls to wake sleeping modules whenever there is new data to work on. There is no need for system states, as the modules will perform the tasks of all three states, as needed. The concept of data queues is explained in Section 8.3.3.

8.2.3 Parallel Modules

The possibility of having parallel modules in a graph is desirable because it allows parallel processing of packets. The problems when trying to find a good design for parallel modules was that in a straightforward design, with or without data queues between the modules, copies of the packets would have to be made for each of the parallel modules, and, how should the copies be combined again after the parallel modules. This problem was also identified in [25].

Conclusion

The solution is based on a design where there are queues, between the modules in the graph, that keep track of data cells which are waiting to be processed by a module. The fact that each module has it's own thread of execution enables a design with passive data queues.

The modules themselves do the work of propagating the data cells through the graph. When a module is ready to receive a data cell, it fetches one from its in-queue(s). And when it is finished it places the data cell in its out-queue(s).

Figure 8.8 shows how data cells are shared between parallel modules. The arrows from data queues and Da CaPo modules indicate that the objects in the data queues and are in fact pointers to the real data cell objects. Thus, parallel modules can do computations on the contents of the same data cell simultaneously.

A module may have an arbitrary number of queues in both directions. Internal mechanisms that are common in all modules (part of the module shell) keep track of the number of queues to and from each module. If there are more than one queue in the in-direction, then the module is in a level

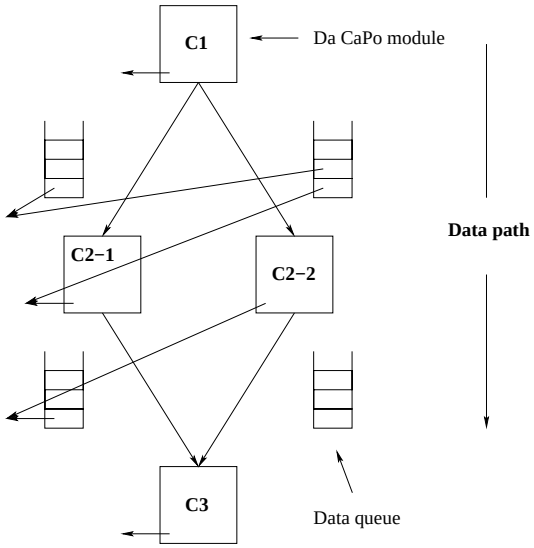

Figure 8.8: Example showing how data cells are shared between parallel modules.

below parallel modules. If there are more than one queue in the out-direction, then the module is in parallel with one or more modules. How many modules a module is in parallel with is irrelevant to that module. There can be nested levels of parallel modules, as illustrated in Figure 8.9.

When a module is finished doing computations on the contents of a data cell, the module puts the data cell in the queue for the next level in the module graph. When there are parallel modules in the next level, the data cell is put in the in-queue for each of these.

A module which is in the level after parallel modules will have several in-queues, one for each of the parallel modules. Before such a modules begins computations on the contents of a data cell, that data cell must be present in all it's in-queues. All modules have simple mechanisms for synchronizing their in-queues, if there is more than one.

8.3 RTLinux

This section presents some of the design issues specifically related to RT-Linux. Each problem that is discussed is introduced by a description of the problem. Next, a solution, or possibly several alternative solutions is presented. When there are alternative solutions, they are examined and choice made based on the findings.

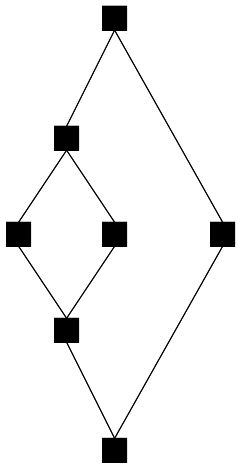

Figure 8.9: Nested levels of parallel modules.

8.3.1 Threads

An RTLinux thread and all it's resources are statically allocated. Calls to `pthread_create` can only be done in kernel mode. As explained in Section 7.6.1, real-time threads cannot run in kernel mode. This means that new threads cannot be created in real-time.

Alternative solutions

- One alternative is to make real-time threads call a non-real-time process and make it create a new real-time thread. The solution enables dynamic creation of threads from within running threads, but creation of threads will not be in real-time. Figure 8.10 illustrates a possible way to realize the concept:

  1. An old real-time thread wants to create a new real-time thread.
  2. The old real-time thread passes instructions to start a new real-time thread, to a non-real-time process running under Linux.
  3. The non-real-time process tells the Linux kernel to create a new real-time thread. This is done using standard Linux library functions for loading kernel modules.
  4. The Linux kernel loads the indicated kernel module, and thus creates a new real-time thread.

  Table 8.4 summarizes the arguments for and against this design.

- Another solution is to design a pool for threads. In this alternative, a large amount of threads for modules are crated when Da CaPo is

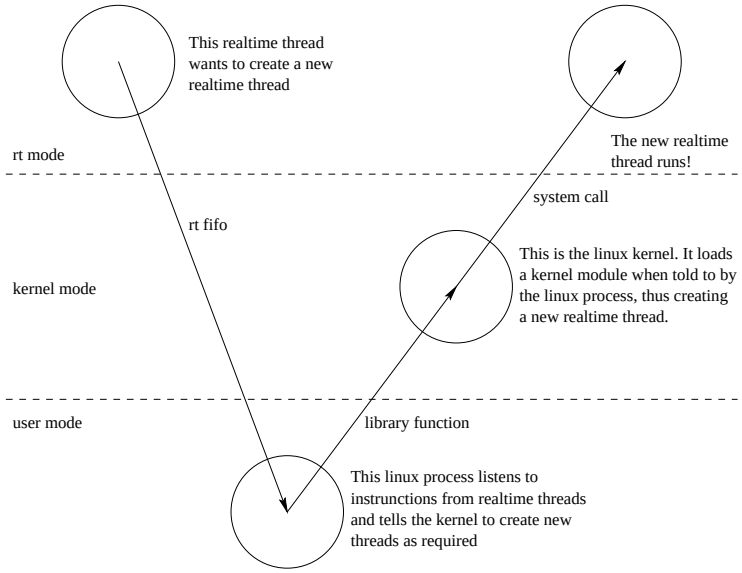

Figure 8.10: An example of how to do dynamic creation of threads

PROS	Easy to implement.
CONS	Thread creation not in real-time.

Table 8.4: Arguments for and against dynamic non-real-time creation of threads.

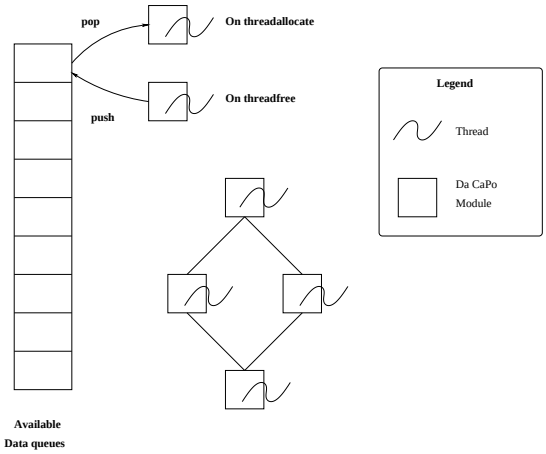

Figure 8.11: Threads are recycled when a module is no longer needed.

PROS	Application maintains real-time properties.
CONS	More difficult to implement.

Table 8.5: Arguments for and against a stack based pool for threads.

initialized. When a module thread is not used in a graph, it is put into the thread pool.

A design for a thread pool that can be implemented under RTLinux is illustrated in Figure 8.11. The pool is a stack of references to the module threads in the pool. Calls to create and destroy threads are replaced by calls to pop and push threads from the stack. When a module graph is set up, or a module inserted into an existing graph, module threads are taken from the stack. When a module graph is dismantled, or a module is removed from it, the module threads are returned to the stack.

Table 8.5 summarizes the arguments for and against this design.

Conclusion

Choosing a design where thread creation is the only non-real-time function call would cause the application to be unpredictable. This was considered to be so undesirable that it was made a high priority to come up with a design that solved this problem without imposing non-real-time functions into the real-time application. The resulting design is a satisfactory solution, even though it is more complex to implement. It leads to one of the fundamental rules in the overall design:

> The threads in the thread-pool are "module-shells". While in the pool, they are identical in that they contain common module functionality, but no Da CaPo module functionality. The appropriate Da CaPo module functionality is added to a module when used in a module graph.

In other words, all modules have some common functionality, while the "protocol functionality" of a module is dynamically replaceable. Da CaPo 2 should include a selection of functions that implement the desired protocol functions. All modules can be instructed to reference any one of these protocol functions.

A positive effect of using pool solutions is that allocation and deallocation of pooled resources can be more efficient than kernel calls for allocation/deallocation. This also applies for the pool solutions in the next sections.

8.3.2 Memory Allocation

The memory allocation problem (Section 7.9) is very similar to the problem with threads. Memory must be statically declared, and allocated when the application starts. But, unlike with threads, it is not a viable alternative to have the application make non-real-time calls at run-time to allocate memory. This would be a very complex solution.

Conclusion

Thus, a design for a memory pool was chosen. The solution is similar to the solution in the old design. A large memory block is declared statically and allocated at initialization of Da CaPo. The memory block is separated into chunks of a preset size. Each chunk is allocated to a data cell. A data cell is an object with a pointer to the chunk of memory, and some additional variables. These data cells will contain the data packets that are sent through the module graph. Also at start up, a stack is created, containing all the data cells. Whenever a packet arrives, the first module in the graph requests data cells from the stack and copies the packet into the memory chunk of those data cells. Then the data cells are propagated through the module graph. The data itself is never moved from its location as long as the data cell is being processed through a module graph. Rather, a pointer to the data is propagated through modules and data queues. When the data cells reach the final module in the graph, the data is copied from the data cell, and the data cell is returned to the stack.

Figure 8.12 illustrates the concept. In the figure, the direction of the module graph is from application to network, but the design is the same for the reverse direction.

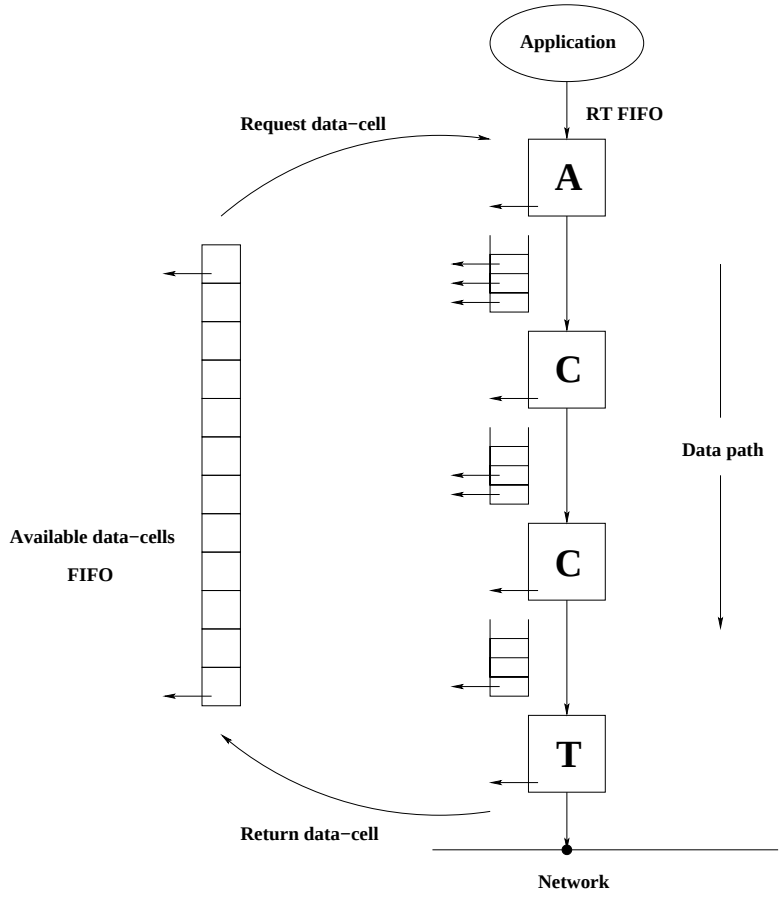

Figure 8.12: Pointers to data cells propagate through the graph.

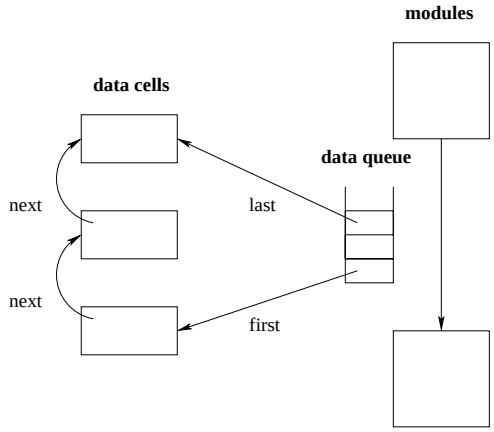

Figure 8.13: The data queue is a linked list that works like a FIFO queue.

8.3.3 Data Queues

Data queues are queues for pointers to data cells that are between modules in a graph. They are necessary for two reasons:

- They enable a module to start working on a new data cell as soon as it is finished with one. That is, a module does not have to wait for the receiving module(s) to be ready to receive the data cell. It simply places the data cell in the queue(s) for the next module(s), where the next module(s) can fetch it when ready.

- They are necessary in order to enable the design for parallel modules (Section 8.2.3).

When creating the design for data queues, the most important consideration was to avoid copying of data, because copying of data is an expensive operation in terms of execution time. The design must also fit with the other parts of the overall design, and be possible to implement for RTLinux.

The resulting design is illustrated in Figure 8.13. The elements stored in the queues are data cells. The queue is implemented as a linked list of data cells. When a module fetches a data cell, it unlinks it from the queue so that the queue functions as a FIFO queue. Similarly, when a module is finished processing the data in a data cell, it links the data cell in as the last member in the linked lists that are its out-queues. Each module is linked to one or more in-queues and one or more out-queues. This can be seen in Figure 8.14.

In addition, the design uses the same model for recycling data queues as with threads and data cells. Unused data queues are kept on a stack (Figure 8.14).

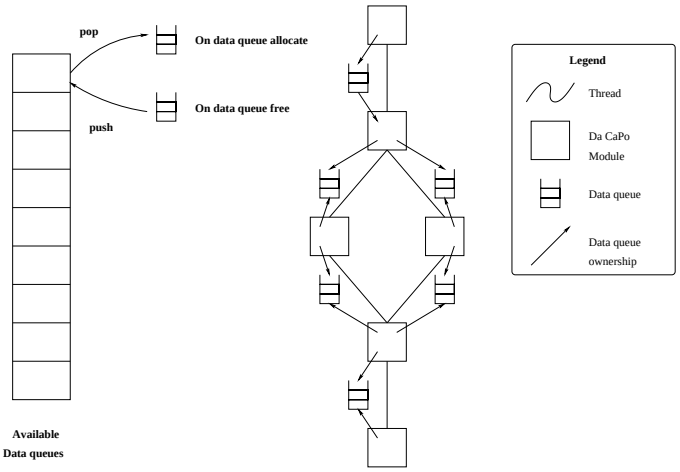

Figure 8.14: Data queues are recycled when no longer needed in a graph.

In addition, the design uses the same model for recycling data queues as with threads and data cells. Unused data queues are kept on a stack (Figure 8.14).

8.3.4 Modules

This chapter is concluded with a brief summary of how modules work. After a module graph has been assembled, each module in it runs as an independent thread in the same process. A module loop through an algorithm as long as there is data to be processed. When data is ready, it is processed by the module. When a module is done processing some data, it passes the data on to the next module(s), and starts processing the next piece of data. If there is no data available, the module puts itself to sleep. As soon as data becomes available, the module is activated again.

There is no need for the module states of the old Da CaPo design. The modules are, to the kernel, ordinary threads. As such they can be in the ordinary thread states, executing, ready to execute or asleep. Modules that are not currently assigned to a module graph are in the module stack, and as long as they are in the stack they are sleeping threads. They still sleep when they are allocated for a module graph, and is first wakened when there is data waiting to be processes in its in-queues.

- **A-module**. An a-module gets data packets from an application, via its designated RT-FIFOs. Next, it gets available data-cells from the data-cells stack, and copies the data packets into the data-cells. Finally, it places the data-cells in it's out-queues. Figure 8.15 illustrates the a-module.

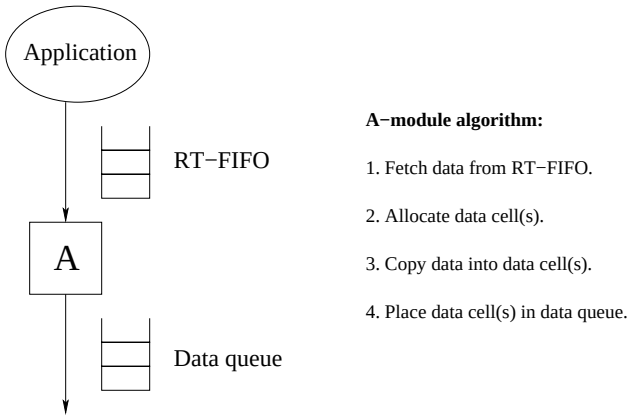

Figure 8.15: A-module behavior.

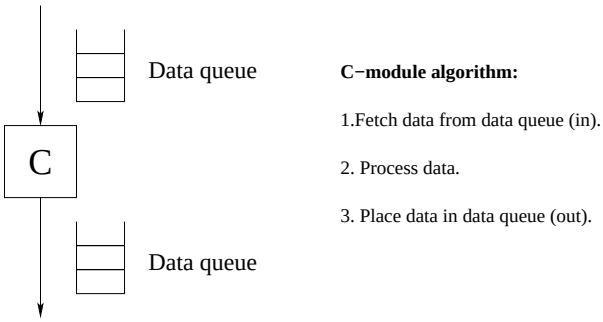

Figure 8.16: C-module behavior.

- **C-module**. A c-module gets a data-cell from its in-queues. Next, it processes the data in the data cell. When finished, it places the data-cell in its out-queues. Figure 8.16 illustrates the c-module. This behavior is according to the specification in [16], page 14.

- **T-module**. A t-module gets a data-cell from its in-queues. Next, it sends the data out on the network. Finally, it returns the data-cell back to the data-cell stack. Figure 8.17 illustrates the t-module. The t-module can be implemented to communicate directly with the network hardware, ie. it is a driver, or the driver can be a separate thread, outside of Da CaPo, that can serve several t-modules.

Because they are threads, the modules are scheduled according to whatever scheduling policy the kernel is using. In RTLinux, the scheduler is modular (Section 7.6), so different scheduling algorithms can be used. Because of the rich possibilities for monitoring of modules/threads in the thread per mod-

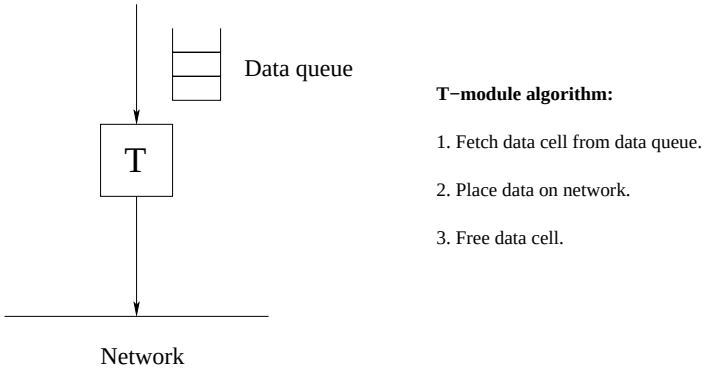

Figure 8.17: T-module behavior.

ule design, it should be possible to very good result with scheduling policies based on dynamic and preemptive scheduling. This is the province of future study (Section 10.3).

8.4 Summary

This chapter has presented the overall design that was created as part of this thesis. The design problems that has been studied in in the work on this thesis was explained, possible solutions discussed, and design choices was made based on these discussions. The next chapter presents the implementation work that has been accomplished based on the designs in this chapter.

Chapter 9

Implementation

This chapter describes the implementation and the implementation process. The design process was more time-consuming than anticipated, and that reduced the amount of time available for work on implementing the design. Thus, the implementation is far from complete. But the important data structures for both modules, data cells and data queues have been implemented, and many of the functions related to the data structures.

9.1 Related Issues

A step-by-step guide for how to setup a system under which the implementation compiles and runs under is included so that the research conditions can be replicated. The next section explains why C was the chosen programming language.

9.1.1 Installation of RTLinux

The process of setting up the system used in this thesis is described here for two reasons; to simplify the process of recreating the conditions the implementation runs under, and also to make it clear to the reader how RTLinux is organized.

The developers of RTLinux have emphasized that they want it to be as simple as possible. As a result the method has changed somewhat over the years. And the user now has a choice of three slightly different methods. I will here outline the steps involved when using the method I prefer when installing RTLinux version 3.0, which is patching a standard Linux kernel source tree. This is not intended as an installation manual[1], and assumes that the reader is familiar with the process of compiling a standard Linux kernel.

[1] An installation manual is included with the RTLinux distribution.

Hardware
CPU: AMD Duron 650MHz RAM: 256MB
Software
Linux distribution: RedHat Linux 7.1 Kernel version: 2.4.1

Table 9.1: The RTLinux system used for this thesis.

System requirements

This information is gathered from the official RTLinux FAQ[3], and the documentation included with the RTLinux version 3.0 distribution. RTLinux version 3.0 runs on Intel and compatible x86 processors, from 486 and higher, Power PC processors, and Alpha processors. As little as 4 MB of Random Access Memory (RAM) is required. There are no other hardware requirements. Version 3.0 supports Linux kernel versions 2.2 and 2.4. Any one of the Linux software distribution can be used.

The system used for this thesis had, as shown in Table 9.1, an Advanced Micro Devices, Inc (AMD) Duron 650 MHz processor and 256 MB RAM. The Linux kernel was version 2.4.1. Apart from the kernel, the software setup was standard RedHat Linux 7.1.

Method

The steps describes the process of installing RTLinux by patching a standard Linux kernel. There are in fact three slightly different installation methods which are supported.

- Patching a standard Linux kernel.
- Installing a pre-patched Linux kernel.
- Installing using RedHat Packet Manager (RPM) packages.

Both the latter methods involves fewer steps and is meant to be simpler. Personally, I prefer to patch a standard kernel, because this gives me better overall control of the process.

The Five Steps

1. **Getting the software**. The software is available for free download from the official website for RTLinux[2]. The main file that needs to

[2] http://www.rtlinux.org

be downloaded is `rtlinux-3.0.tar.gz`. It contains the entire RT-Linux distribution, including source tree, kernel patch, documentation and programming examples. In addition I recommend getting the file `rtldoc-3.0.tar.gz`, which contains additional documentation, mainly man page.

2. **Unpacking the software**. The files are packaged as `tar.gz` files, which is a common format for packaging source files on Unix systems. Log in as `root` and unpack the files, using the `tar` tool, to where source code is normally kept.

```
$ su -
$ cd /usr/src
$ tar zxvf rtlinux-3.0.tar.gz
```

This places the RTLinux distribution in `/usr/src/rtlinux-3.0`. Then,

```
$ cd /
$ tar zxvf rtldoc-3.0.tar.gz
```

places the additional documentation in `/usr/rtlinux-3.0`.

3. **Patching the kernel**. Assuming that the kernel source tree is located in `/usr/src/linux`:

```
$ cd /usr/src/linux
$ patch -p1 < /usr/src/rtlinux-3.0/kernel_patch-2.4.1-x86
```

4. **Compiling the kernel**. Configure the patched kernel with the settings you need, compile the kernel, install it, and re-boot using the new kernel.

5. **Compiling and installing RTLinux**. The final step is to compile and install RTLinux kernel modules that will work with the new and ready-patched kernel, and prepare the device files needed for the RT-FIFO's.

```
$ cd /usr/src/rtlinux-3.0
$ make menuconfig
$ make dep
$ make
$ make devices
$ make install
```

PROS	Native language of RTLinux. Better portability. Simpler.
CONS	Not used in other theses.

Table 9.2: Arguments for and against choosing C.

PROS	Used in other theses. More feature rich. Object oriented.
CONS	Support is immature in RTLinux.

Table 9.3: Arguments for and against choosing C++.

9.1.2 Programming Language

There were good arguments for choosing either C or C++ as the programming language for the implementation..C is a natural choice, since that is the native language for RTLinux. C++ is the preferred language in at least two other papers on developing the next version of Da CaPo ([25] and [23]).

- **C**. RTLinux is implemented in C. It is the only officially supported language at this time. C is a simple language to use, and its features are sufficient in order to implement the design from the previous chapter. C is one of the most widely available programming languages, so mortality between architectures is good. Arguments for and against are summarized in Table 9.2.

- **C++**. The programming language is more feature rich. Among the things supported is object oriented programming. Some of theses features are extensively taken advantages of in the designs in both [25] and [23]. That reason alone makes it more desirable to use C++ also in this thesis. However, even though it is possible to use C++ when programming for RTLinux, it is not fully supported yet. There seem to be a multitude of problems related to using C++ at this time. Arguments for and against are summarized in Table 9.3.

Conclusion

Based on the reasons mentioned above, it was perceived that it would have been preferable to implement in C++. Yet, C has been used. The sole reason for this is that it was the impression that C++ support for RTLinux was still too immature. The potential for problems related to this was to

big, and because of the time-frame for the thesis, it was decided to avoid this.

A consideration that made the choice simpler was that he C++ support for RTLinux seem to be maturing rapidly, and porting the C code to C++ at a later stage should be relatively easy.

9.2 Organization of the Code

The implemented code is organized in different files that logically separates functions and variables:

- `rtl_dacapo.c`. The main file. It is organized according to specifications in [47] so that it will be recognized as a Linux kernel module. It makes all the necessary calls in order to initialize Da CaPo 2. At the current stage of the development, it also contains calls to start a module graph for testing purposes. When the implementation is complete, establishing module graphs will be done when requested by an application (see Section 8.1).

- `rtl_dacapo.h`. The common header file. All the basic data structures are declared here. Chapter 9.3 discusses the data structures.

- `data_cell.c`. The file defines the data cell objects and the data cell stack. Furthermore it contains all the functions related to the data cell stack. This is further explained in Section 9.3.2.

- `data_queue.c`. The file defines the data queue objects and the data queue stack. Furthermore it contains all the functions related to the data queue stack. This is further explained in Section 9.3.3.

- `module.c`. This file defines the statically declared number of modules, and the module stack. It contains all the functions related to the module stack. It also contains the generic code for the modules themselves. I is further explained in Section 9.3.1.

- `module_functions.c`. This is a file that contains dummy functions to be run by the modules as the non-generic code. This file is only present for testing purposes. The decision on where and how to store the Da CaPo module functions has not been made yet.

- `management_toolkit.c`. The file contains the administrative functions that is directly related to only one of the data structures. They are explained in Section 9.4.

9.3 Data Structures

This section presents the essential data structures for modules, data cells and data queues, and the functions related to these, which are the implementations of the designs in Chapter 8. In addition it was necessary to add a data structure to store the module graph structure in.

Due to the RTLinux limitation that makes it impossible to allocate memory in real-time (Section 7.9), some values are being set at compile time. The names for the variables holding these values are in uppercase letters. Declaring the values like this a temporary shortcut. In the final implementation they should be arguments to the Da CaPo kernel module.

9.3.1 Modules

Implementation of the modules, or rather the "module-shells" can be summed up in two concrete tasks:

- Implement a way for the modules Da CaPo module function to be dynamic. That is, the function that the module is to perform on the data-cells must be replaceable.
- Implement a way for the modules to be linked with the appropriate data queues.

The data structure for modules is implemented as follows. Because the name "module" is already in use in the Linux kernel, the name "node" is used in the program code.

```
struct node {
  int type;
  pthread_t task;
  int fifo;
  void (*function)(void *);
  int up_queues;
  int down_queues;
  struct data_queue *up_queue[PARALLELLITY];
  struct data_queue *down_queue[PARALLELLITY];
};
```

Figure 9.1 illustrates how the first task is solved by letting each module have a function pointer, `function`, that references the module function that is at any time assigned to the module. The second task is solved by having an array of pointers to the data queues for each direction, `up_queue` and `down_queue`. The number of slots in the array is determined by the statically declared value in `PARALLELLITY`, which then sets the maximum limit for number modules in parallel.

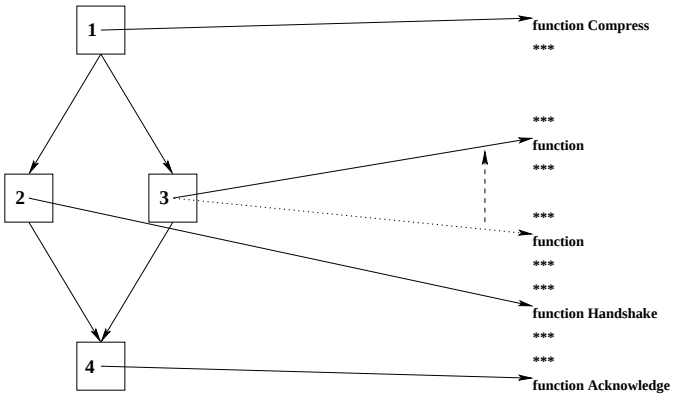

Figure 9.1: Modules are dynamically linked to the appropriate module functions. Module no.4 is in the process of having it's module function changed.

The following functions that manipulate modules have been implemented:

```
void init_nodes(void);
void cleanup_nodes(void);
void init_nodes_stack(void);
void node_free(struct node *);
struct node * node_alloc(void);
```

The `init_nodes` and `cleanup_nodes` are the functions that are used respectively to initialize the modules at system start-up and clean up at system exit. The cleanup releases allocated threads as required by RTLinux. The `node_free` and `node_alloc` functions respectively implement the push and pop functionality on the module stack, as discussed in Section 8.3.1.

9.3.2 Data Cells

The `data_cell` struct implements the design from Section 8.3.2. The data cells are passive elements with two required properties; they must have a memory segment where packets are to be kept, and they must be linkable in order to implement the linked list design for data queues, from Section 8.3.3.

```
struct data_cell {
  char cell[DATACELLSIZE];
  int used_bytes;
  struct data_cell *next;
};
```

The statically defined `DATACELLSIZE` sets the size in bytes for the data cell memory blocks. The `next` variable is a pointer to the next data cell in the linked list when the data cell is in data queue.

```
void init_data_cells_stack(void);
void data_cell_free(struct data_cell *);
struct data_cell *data_cell_alloc(void);
```

The `init_data_cells_stack` function allocates the data cells at start up, and sets up the data cell stack. The number of data cells that are allocated is statically defined. At start, all data cells are pushed on the stack. The `data_cell_free` and the `data_cell_alloc` functions respectively implements the push and pop functionality of the design in Section 8.3.2.

9.3.3 Data Queues

The `data_queue` struct, together with the `data_cell` struct implements the design from Section 8.3.3. The data cells a data queue are linked together and the `data_queue` contains pointers to the first and last element of the list.

```
struct data_queue {
  int status;
  struct data_cell *first;
  struct data_cell *last;
};
```

`first` and `last` points to the first and last elements in a data queue.

```
void init_data_queue_stack(void);
void data_queue_free(struct data_queue *);
struct data_queue *data_queue_alloc(void);
```

The `init_data_queue_stack` function allocates the data queues at start up, and sets up the data queue stack. The `data_queue_free` and the `data_queue_alloc` functions respectively implements the push and pop functionality of the design in Section 8.3.3. The functionality of moving a data cell from a data queue and into a module is implemented in the modules.

9.3.4 Graphs

The main problem with the implementation was that it was necessary to be able to access a graph as an object. This is necessary for example when assembling a module graph based on a given pattern, and when removing, replacing or inserting a module. A module graph, when it is set up, interconnects all the modules, data queues and data cells in the graph, but the structure of the graph is not preserved, so there is no easy way to access directly one specific component in the graph. The `graph` struct was created to be able to do this.

Each module graph will at all times have related graph struct. Figure 9.2 illustrates how a graph object is used to access elements in the module

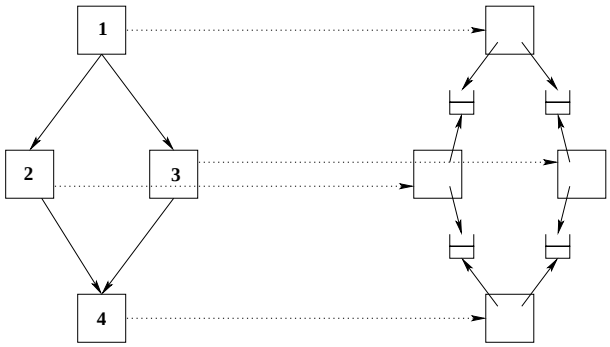

Figure 9.2: The module graph elements, on the right, are accessed via the graph object on the left.

graph. The graph struct, on the left, stores the structure of the module graph by having pointers to all modules, and storing all the inter-module relationships. In the module graph, on the right, modules store pointers to the data queues they each connect with. Data cells in the module graph are not shown, as they are always either linked to a module or a data queue. Because the modules themselves are ignorant of which module is at the other end of a which data queue, that information is also stored in the graph struct.

```
struct edge {
  int up;
  int down;
};

struct graph {
  int direction;
  int nodesingraph;
  int edgesingraph;
  struct node *nodes[MAXNODES];
  struct edge edges[MAXEDGES];
};
```

As the code listing shows there is a struct called `edge` in addition to the one called `graph`. An edge is in this context the relationship between two interconnected modules in a graph. The variable `MAXEDGES` contains a statically defined value that sets the maximum limit for how many edges there may be in a graph. The `*nodes` array contains pointers to all the modules in the graph. An `edge` struct contains references, in the form of `*nodes` index values, to the two modules the edge connects. Thus, the structure of the graph is stored in the `edges` array.

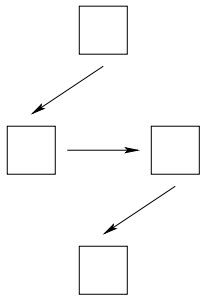

Figure 9.3: A module graph being assembled.

9.4 Supporting Functions

The only supporting function has been implemented so far;

```
int AssembleGraph(struct graph *);
```

The format of the "blueprints" for possible graph compositions has not been decided upon yet. For now, the `AssembleGraph` function takes a reference to a `graph` struct as an argument. It is assumed that the graph composition that is to be build is already stored in that graph, and the function assembles the module graph based on it. The `graph` struct is traversed and modules allocated in the pattern shown in figure 9.3. Next, data queues are set up as required for each module.

9.5 Summary

This chapter has presented the elements of the implementation that has been done so far. The data structures for module, data cells and data queues have been implemented. An additional data structure has been created to enable management of module graphs and their elements. Functions for setting up a graph and for dismantling it have been implemented. More supporting functions still need to be implemented, for example functions for replacing, inserting or removing modules in a running graph.

Testing of the implementation so far have been done by using dummy graphs and modules. The implementation compiles, and builds and activates module graphs. The implementation is incomplete, but some of the most important elements have been implemented. The work has come far enough that it is possible to claim that the complete implementation is at least possible.

Chapter 10

Conclusion

10.1 Summary of Thesis

This thesis is the result of a study aimed at taking advantage of real-time scheduling support in Da CaPo. The problem definition arises because distributed multimedia have requirements that current communication protocols were not designed to meet. Those protocols have no or little support for QoS, and were mostly designed for networks with low bandwidth capacity. Da CaPo is a framework for flexible protocols that aims to solve many of these problems. This is why the continued development of Da CaPo is the motivation for this thesis.

In order to make correct decisions on choice of OS and what kind of functionality to include in the resulting design, background research was done on real-time and multimedia in general, and the concept of QoS. These studies are the basis in which the choice of RTLinux as OS was made.

Many considerations had to be made in the design process, to accommodate for desired features, and limitations imposed with the use of RTLinux. The resulting design is a complete overall design, and has detailed designs for modules and message queues.

The design had been partly implemented. It compiles and runs with test values. It strongly indicates that the design is valid, and implementation is possible. Unfortunately, it was not possible to complete the implementation process, because of to the time limited time-frame.

10.2 Goal Achievement

A summary of results is given in this section. The part problems have set the direction of the thesis work. A look at how they have been solved is a good measure how well the goals of the thesis have been reached.

> **Problem 1:** "What is required of a real-time operating system for it to be suitable as a base for a multi-threaded Da CaPo?".

Due to the practical rather than theoretical angle that was chosen for the thesis, this problem was largely interpreted as "Which real-time operating system should be chosen as a base for a multi-threaded Da CaPo?". The recent advent of real-time enabled variants of Linux presented an alternative to using a standard micro-kernel RTOS. The Linux option provides real-time properties, like real-time scheduling, and is, at the same time, a fully featured mainstream desktop OS. This blend was perceived as very interesting in the context of Da CaPo, since in most (all?) distributed multimedia applications, at least one of the hosts is a desktop system.

In order to be able to choose between the real-time Linux variants, some study was made of what features to specifically look for in the context of this thesis. The most important issues were scheduling features and portability, but for these non-commercial OSes, things like documentation and support was also considered to be of great importance.

RTLinux was finally chosen over RTAI, because of its "keep it simple" philosophy and its POSIX compliance. Since that choice was made, it has proven to be a good one. Of the other alternatives, one is very slow in development, while the other seems to have become a dormant project. Both RTAI and RTLinux have been in heavy development, both have improved a lot, and user bases have grown large. All the reasons for choosing RTLinux remains valid. It is considered to have been a good choice.

> **Problem 2:** "What should the design of a multi-threaded Da CaPo look like?".

Again, due to the practical angle in the thesis, and based on the choice of RTLinux as OS, this problem was interpreted as: "Create a design of a multi-threaded Da CaPo for RTLinux!".

It was never the goal of this thesis to create a design that covers every detail of Da CaPo. The system is too complex for that to be done in one master thesis. Several theses deal with aspects of Da CaPo. This thesis has been concerned with creating a overall design that is able to take advantage of the real-time properties of RTLinux.

Some design problems are inherent in RTOSes, such as how to organize threads. Other problems were introduced by limitations in RTLinux, mainly related to the fact that many resources can't be allocated in real-time. This was solved by creating designs for buffers or "pools" of threads and memory constructs.

The resulting design has solutions to all of these problems. Furthermore, it is complete in the sense that the "black boxes", parts of the design that was not studied in detail, have been accounted for, and can be inserted into the model. The RTLinux developers has recently announced that they are developing a thread buffer, similar in characteristics to the one that has been presented here.

Problem 3:"Is it possible to implement the new design on the chosen OS?".

The amount of problems encountered when working on the design exceeded what had been anticipated. Thus, the design process turned out to be quite time consuming, at the expense of the implementation phase.

The design has been partially implemented on RTLinux. The detailed designs that solves the RTLinux specific problems have been implemented. The code compiles and runs with test values. The implementation so far indicates that the overall design is valid, and implementable.

10.3 Future Work

During the work on this thesis, it became clear that some tasks would be more time-consuming than had at first been anticipated. As a result, the goals and problem definitions were adjusted along the way. There is some future work that is directly related to this thesis, and would have been included, if not for the time limitations.

- **Complete the implementation**. The implementation work has achieved its goal of showing that the complete implementation of the design is possible. The work of completing what remains of the implementation is probably not very time-consuming or difficult.

- **Design and implement remaining components**. Again due to the time-limitations, it was necessary to limit the design elements. Possible future task involves design and implementation of the Da CaPo components that were left out of the scope of this thesis.

- **Perform scheduling tests**. When a full, or simulated full implementation of Da CaPo for RTLinux is available, it is possible future work to experiment with and adjust the scheduling facilities of the OS.

It is the opinion in this thesis that RTLinux is a viable platform for Da CaPo and presents many interesting possibilities related to scheduling of modules and improved QoS support. As such, it is naturally hoped that the future work listed above will be done.

Regardless, it is hopeful that some of the work contained within can be of use in other Da CaPo related projects. The "pool" designs, the data queue, the data cells and the "parallel modules" design are all designs with properties that can be useful in other implementations of Da CaPo.

10.4 A Personal Note

I do not wish to use much space on lamentation, but I do feel that it is fitting to say that the work on my thesis did not at all go as smoothly as I had hoped for. I never wavered in my decision to complete the work, but I must admit that at times my motivation has been low. The work has not been as rewarding, or given me the pleasure that I feel research should.

Much of this, I feel, is due to the unfortunate decision to first use ChorusOS as the base for the thesis. At that time, ChorusOS seemed a very good choice. It was a commercial RTOS with UNIX compatibility, the kernel source code was available, and it had already been used in some Da CaPo related work.

I attempted for a while to do a thesis where ChorusOS was a central element. I gradually discovered that there was practically no documentation on kernel internals available, there are no active user communities, there was no official support available, and the kernel source code is extremely poorly commented and is confusingly organized. After struggling with these problems for a while, ChorusOS was acquired by Sun Microsystems. Sun proceeded to change ChorusOS in a manner that removed much of my motivation for using it in the first place.

I decided to use RTLinux instead, and have not regretted that decision. But a lot of work and time was wasted on the failed attempts with ChorusOS.

Bibliography

[1] M. Barabanov and V. Yodaiken. Introducing real-time unix. *Linux Journal*, 34, Feb 1997.

[2] Michael Barabanov. A Linux-based real-time operating system. Masters thesis, New Mexico Institute of Mining and Technology, Jun 1997.

[3] Michael Barabanov and Victor Yodaiken. The rtlinux faq. From http://www.rtlinux.org.

[4] Michael Barabanov and Victor Yodaiken. Real-Time Linux. *Linux Journal*, Mar 1996.

[5] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dir Verworner. *Linux Kernel Internals*. Addison-Wesley, 1998.

[6] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[7] G. Coulson, G. Blair, P. Robin, and D. Shepherd. Extending the chorus micro-kernel to support continuous media applications. In *Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 49–60, Nov 1993.

[8] G. Coulson, G. Blair, P. Robin, and D. Shepherd. Supporting continuous media applications in a micro-kernel environment. *Architecture and Protocols for High-Speed Networks*, 1994. Kluwer Academic Publishers.

[9] Robert A. Day. *How to Write & Publish a Scientific Paper*. Oryx Press, fifth edition, 1998.

[10] Peter J. Denning, Douglas E. Comer, David Gries, Michael C. Mulder, A. Joe Turner Allen Tucker, and Paul R. Young. Computing as a dicipline. *Communications of the ACM*, 32(1):9–23, Jan 1989.

[11] Willibald A. Doeringer, Doug Dykeman, Matthias Kaiserswerth, Bernd Werner Meister, Harry Rudin, and Robin Williamson. A survey of light-weight transport protocols for high-speed networks. *IEEE Transactions on Communications*, 38(11):2025–2038, Nov 1990.

[12] Frank Eliassen, Tom Kristensen, Thomas Plageman, and Hans Ole Rafaelsen. Multe-orb: Adaptive qos aware binding. In *Workshop on Reflective Middleware (RM 2000)*, New York, USA, apr 2000. in conjunction with IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000).

[13] Jerry Epplin. Linux as an Embedded Operating System. *Embedded Systems Programming*, Oct 1997.

[14] Bill O. Gallmeister. *POSIX.4*. O'Reilly & Associates, Inc., first edition, 1995.

[15] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley & Sons, Inc., second edition, 1987.

[16] A. Gotti. *The Da CaPo Communication System, Technical Report*. Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Jun 1994.

[17] The Open Group. Corporate overview. Online.

[18] The Open Group. The Open Group and IEEE to develop joint revision to POSIX and UNIX standards. Press release, Jul 1999.

[19] F. Halsall. *Data Communications, Computer Networks and Open System*. Addison-Wesley, 1994.

[20] Robert Hill, Balaji Srinivasan, Shyam Pather, and Douglas Niehaus. Temporal resolution and real-time extensions to linux. Technical Report ITTC-FY98-TR-11510-03, Informatiom and Telecommunication Technology Center, Department of Electrical Engineering and Computer Sciences, University of Kansas, Jun 1998.

[21] Moses Joseph. Is POSIX Appropriate for Embedded Systems? *Embedded Systems Programming*, page 90, Jul 1995.

[22] Andrew Josey. *Go Solo 2*. Prentice Hall, 1997.

[23] Ingvild Berlin Kalleberg. Integrasjon av fleksibel signalering i da capo. Masters thesis, University of Oslo, Department of Informatics, Nov 2000. In Norwegian. Translated title: Integration of flexible signalling in Da CaPo.

[24] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1990. Norwegian edition.

[25] Tom Kristensen. Utvidelse av objektmegleren COOL med fleksibel støtte for tjenestekvalitet. Masters thesis, University of Oslo, Department of Informatics, Aug 1999. In Norwegian. Translated title: Extension of COOL with flexible support for quality of service.

[26] Tom Kristensen, Ingvild Berlin Kalleberg, and Thomas Plagemann. Implementing configurable signalling in the multe-orb. In *Proceedings of Fourth IEEE International Conference on Open Architectures and Network Programming (OPENARCH 01)*, Anchorage, Alaska, USA, apr 2001. to appear.

[27] Leslie Lamport. *LaTeX User's Guide and Reference Manual.* Addison-Wesley, second edition, 1994.

[28] Bil Lewis and Daniel J. Berg. How to program with threads. *SunWorld*, Feb 1996.

[29] P. Mantegazzi, E. Bianchi, L. Dozio, S. Hughes S. Papacharalambous, and D. Beal. *Introducing the Real Time Application Interface (RTAI) for Linux.* Dipartimento de Ingegneria Aerospaziale, Politecnico di Milano and Zentropic Computing, LLC, 1999.

[30] Yves-Olivier Metais. Conception et implementation d'un ordannanceur a echeance au sein du noyau Chorus. Memoire présenté en vue d'obtenir le dimplôme d'ingenieur C.N.A.M, Conservatoire National des Arts et Metiers, Paris, Mar 1994. In French. Translated title: Design and implementation of an execution scheduler in the Chorus kernel.

[31] Patrick Mourot. RTAI internal presentation. Alcatel, France, 1999.

[32] Sape J. Mullender, Ian M. Leslie, and Derek McAuley. Operating-system support for distributed multimedia. In *1994 Summer Usenix Conference in Boston, Ma.*, Jun 1994.

[33] D. Niehaus, William Dinkel, and Sean B. House. Effective real-time system implementation with KURT Linux. Information and Telecommunication Technology Center, Electrical Engineering and Computer Science Department, University of Kansas.

[34] The Open Group. *The Single Unix Specifiction*, second edition, 1997.

[35] Thomas Plagemann. Protocol configuration - a flexible and efficient approach for qos provision. In *IFIP IWQoS'96*, 1996. Position statement.

[36] Thomas Plagemann. Analysis of multithreading and real-time scheduling for the flexible communication sub-system da capo on chorus. In *Proceedings of Multimedia Systems and Application Conference, SPIE's Symposium on Voice, Video, and Data Communication*, pages 339–351, Boston, USA, Nov 1998.

[37] Thomas Plagemann. A framework for dynamic protocol configuration. *European Transactions on Telecommunications (ETT)*, Special Isssue on Architectures, Protocols and Quality of Service for the Internet of the Future, 1999.

[38] Thomas Plagemann, Frank Eliassen, Brita Hafskjold, Tom Kristensen, Robert H. Macdonald, and Hans Ole Rafaelsen. Flexible and extensible qos management for adaptable middleware. In *Proceedings of International Workshop on Protocols for Multimedia Systems (PROMS 2000)*, Cracow, Poland, oct 2000.

[39] Thomas Plagemann, Vera Goebel, Pål Halvorsen, and Otto Anshus. Operating system support for multimedia systems. *Computer Communications Journal*, Special Issue on Interactive Distributed Multimedia Systems and Telecommunications Services 1998 (IDMS'98), Winter 1999.

[40] Thomas Plagemann, Vera Goebel, and Morten Tollefsen. Depend: Distance education for people with different needs. In *Proceedings of Second IASTED ISMM International Conference Distributed Multimedia Systems and Applications, Stanford, California*, pages 159–162, Aug 1995.

[41] Thomas Plagemann, Andreas Gotti, and Bernhard Plattner. CoRa - a heuristic for protocol configuration and resource allocation. In *IFIP Workshop on Protocols for High-Speed Networks*, Vancouver, Aug 1994.

[42] Thomas Plagemann and Bernhard Plattner. Evaluating crucial performance issues onf protocol configuration in Da CaPo. In *First Workshop on High Performance Protocol Architecture HIPPARCH'94*, Sophia Antipolis, France, Dec 1994.

[43] Thomas Plagemann, Bernhard Plattner, Martin Vogt, and Thomas Walter. A model for dynamic configuration of light-weight protocols. In *Proceedings IEEE Third Workshop on Future Trends of Distributed Systems*, Apr 1992.

[44] Thomas Plagemann, Bernhard Plattner, Martin Vogt, and Thomas Walter. Modules as building blocks for protocol configuration. In *Proceedings International Conference on Network Protcols, ICNP'93*, pages 106–115, San Fransisco, California, Oct 1993.

[45] Thomas Plagemann, Knut A. Sæthre, and Vera Goebel. Application requirements and qos negotiation in multimedia systems. *Second Workshop on Protocols for Multimedia Systems, PROMS'95*, Oct 1995.

[46] Thomas Plagemann, Janusz Waclawczyk, and Bernhard Plattner. Management of configurable protocols for multimedia applications. ETH Zurich, Computer and Networks Laboratory (TIK), ETH Zentrum, CH-8092 Zurich, Switzerland.

[47] Ori Pomerantz. Linux kernel module programming guide. Linux Documentation Project, 1999.

[48] Portable Applications Standards Committee of the IEEE Computer Society, New York, USA. *Information technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]*, second edition, Jul 1996. ANSI/IEEE Std 1003.1.

[49] Portable Applications Standards Committee of the IEEE Computer Society, New York, USA. *IEEE Standard for Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP)*, Sep 1999. IEEE Std 1003.13-1998.

[50] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems. In *Proceedings of the IEEE*, volume 82, pages 55–67, Jan 1994.

[51] Ismael Ripoll. Real-Time Linux (RT-Linux). *Linux Focus*, 1998.

[52] Douglas C. Schmidt. Wrapper facade. *C++ Report Magazine*, Feb 1999.

[53] Balaji Srinivasan. A firm real-time system implementation using commercial off-the-shelf hardware and free software. Masters thesis, Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas, Lawrence, Kansas, 1998.

[54] Balaji Srinivasan, Shyamalan Pather, Robert Hill, Furquan Ansari, and Douglas Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Proceedings of the Real-Time Technology and Applications Symposium*, Denver, USA, Jun 1998.

[55] John A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, pages 10–19, Oct 1988.

[56] John A. Stankovic and Krithi Ramamritham, editors. *Advances in Real-Time Systems*. IEEE Computer Society Press, Los Alamitos, California, 1993.

[57] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? Editorial, Jul 1993.

[58] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A dynamic proportion allocator for real-rate scheduling. Department of Computer Science and Engineering, Oregon Graduate Institute.

[59] Ralf Steinmetz and Klara Nahrstedt. *Multimedia: Computing, Communications and Applications*. Prentice Hall, 1995.

[60] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[61] Umar Syyid. *The Adaptive Communication Environment: ACE*. Hughes Network Systems, 1998. A tutorial.

[62] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[63] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[64] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, third edition, 1996.

[65] Uresh Vahalia. *Unix Internals*. Prentice Hall, 1996.

[66] S. Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2), Feb 1997.

[67] Andreas Vogel, Brigitte Kerhervé, Gregor von Bochmann, and Jan Gecsei. Distributed multimedia and qos: A survey. *IEEE Multimedia*, Vol. 2(No. 2):10–19, Summer 1995.

[68] Martin Vogt, Bernhard Plattner, Thomas Plagemann, and Thomas Walter. A run-time environment for Da CaPo. In *Proceedings INET'93*, 1993.

[69] Bjørn Volden. Realtime operating system support for multimedia communication. Cand.scient thesis, University of Oslo, Department of Informatics, May 1997.

[70] Yu-Chung Wang and Kwei-Jay Lin. Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. Department of Electrical and Computer Engineering, University of California.

[71] Yu-Chung Wang and Kwei-Jay Lin. Providing real-time support in the linux kernel. Department of Electrical and Computer Engineering, University of California.

[72] Yu-Chung Wang and Kwey-Jay Lin. Enhancing the real-time capability of the linux kernel. In *Proceedings of 5th RTSCA'98*, Hiroshima, Japan, Oct 1998.

[73] David K.Y. Yau and Simon S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *ACM Multimedia*, Nov 1996.

[74] Victor Yodaiken. Cheap operating systems research and teaching with Linux. Departmenet of Computer Science, New Mexico Tech.

[75] Victor Yodaiken. The RT-Linux approach to hard real-time. A short position paper. Departmenet of Computer Science, New Mexico Institute of Technology.

[76] Victor Yodaiken. The rtlinux manifesto. Description of RTLinux as of Version 1.

[77] Victor Yodaiken and Michael Barabanov. RTLinux Version Two. Part of the RT Linux documentation, 1999.

Appendix A

Acronyms

ACM Association for Computing Machinery. ACM, founded in 1947, is the largest and oldest international scientific and industrial computer society, and fosters research and communication in a broad range of computing areas.

AMD Advanced Micro Devices, Inc.

ANSI American National Standards Institute.

API Application Program Interface. An API is the specific method prescribed by a computer operating system or by an application program by which a programmer writing an application program can make requests of the operating system or another application.

ARC Adaptive Rate Controlled process scheduling.

ATM Asynchronous Transfer Mode. ATM is a dedicated-connection switching technology.

CoRA Configuration and Resource Allocation.

CPU Central Processing Unit. CPU is an older term for processor and microprocessor, the central unit in a computer containing the logic circuitry that performs the instructions of a computer's programs.

CSCW Computer-Supported Cooperative Work.

Da CaPo Dynamic Configuration of Protocols. Da CaPo is a design for dynamically configurable lightweight communication protocols.

Da CaPo 2 Dynamic Configuration of Protocols 2. The next generation Da CaPo.

DEC Digital Equipment Corporation.

DCE Distributed Computing Environment.

DIAPM Department of Aerospace Engineering, Politecnico de Milano.

DVD Digital Versatile Disk. An optical disk technology that is expected to rapidly replace the CD-ROM disk (as well as the audio compact disc) over the next few years.

EDF Earliest Deadline First process scheduling.

FAQ Frequently Asked Questions. The FAQ or list of "frequently-asked questions" (and answers) has become a feature of the Internet. The FAQ seems to have originated in many of the Usenet groups as a way to acquaint new users with the rules. Today, there are thousands of FAQs on the World Wide Web.

FIFO First-In, First-Out.FIFO is an approach to handling program work requests from queue or stack so that the oldest request is handled next

FPU Floating Point Unit. An FPU, also known as a numeric co-processor, is a microprocessor or special circuitry in a more general microprocessor that manipulates numbers more quickly than the basic microprocessor in computers.

FTP File Transfer Protocol. FTP, a standard Internet protocol, is the simplest way to exchange files between computers on the Internet

GDB Gnome Debugger.

GPL Gnu Public License.

HAL Hardware Abstraction Layer. The Linux kernel modifications done by RTAI.

HCI Human Computer Interaction. The study of how people interact with computers and to what extent computers are or are not developed for successful interaction with human beings.

HDTV High Definition Television. Television display technology that provides picture quality similar to 35 mm. movies with sound quality similar to that of today's compact disc.

HTTP Hyper-text Transfer Protocol. HTTP is the set protocol for exchanging files (text, graphic images, sound, video, and other multimedia files) on the WWW.

I/O Input/Output. I/O describes any operation, program, or device that transfers data to or from a computer.

IBM International Business Machines Corporation.

IEEE Institute of Electrical and Electronics Engineers. The IEEE fosters the development of standards that often become national and international standards

IMRAD Introduction, Methods, Results, and Discussion. Method for organizing a scientific paper.

INRIA Institut National de Recherche en Informatique et en Automatique.

IP Internet Protocol. IP is the protocol by which data is sent from one computer to another on the Internet

IPC Inter-process Communication. IPC is a set of programming interface that allow a programmer to create and manage individual program process that can run concurrently in an operating system.

ISO The International Organization for Standardization. ISO, founded in 1947, is a worldwide federation of national standards bodies from some 100 countries, one from each country.

KURT Kansas University Real-Time Linux.

LLC Limited Liability Company.

LWP Lightweight Processes.

MAC Media Access Control. MAC is a sub-layer of the Data-Link Layer layer.

MPEG Moving Picture Experts Group. MPEG develops standards for digital video and digital audio compression. It operates under the auspices of ISO.

MTP Management Transfer Protocol.

MULTE Multimedia Middle-ware for Low-Latency High-Throughput Environments.

OS Operating System. An OS is the program that, after being initially loaded into the computer by a boot program, manages all the other programs in a computer. Application programs make use of the operating system by making requests for services through a defined API.

OSF Open Software Foundation.

OSI Open Systems Interconnection.

OSI-RM Open Systems Interconnection Reference Model.

PASC Portable Application Standards Committee.

POSIX Portable Operating System Interface. (API)

QoS Quality of Service. The view in this thesis fits the definition proposed by Vogel [67]:

> "Quality of Service represents the set of those quantitative and qualitative characteristics of a distributed multimedia system necessary to achieve the required functionality of an application."

RAM Random Access Memory. RAM is the place in a computer where the operating system, application programs, and data in current use are kept so that they can be quickly reached by the computer's processor. RAM is much faster to read from and write to than the other kinds of storage in a computer, the hard disk, floppy disk, and CD-ROM. However, the data in RAM stays there only as long as your computer is running.

RM Rate Monotonic process scheduling.

RR Round Robin process scheduling. A method of having different program processes take turns using the resources of the computer by limiting each process to a certain short time period, then suspending that process to give another process a turn (or "time-slice").

RT-FIFO Real-Time FIFO.

RTAI Real-Time Application Interface.

RTLinux Real-Time Linux.

RTOS Real-Time Operating System. An operating system that guarantees a certain capability within a specified time constraint.

RPC Remote Procedure Call. RPC is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details. RPC uses the client/server model. The requesting program is a client and the service-providing program is the server

RPM RedHat Packet Manager. RPM is an open software package managements system available for anyone to use. It is used in several different Linux distributions.

SVID System V Interface Definition.

SMP Symmetric Multiprocessing.

SMTP Simple Mail Transfer Protocol. SMTP is a TCP/IP protocol used in sending and receiving e-mail.

TCP Transport Control Protocol. A transport layer protocol used in the TPC/IP stack.

TCP/IP Transport Control Protocol/Internet Protocol. Commonly used to reference the protocol stack of the Internet. Named after the most commonly used transport layer protocol and the network layer protocol of that stack.

UART Universal Asynchronous Receiver/Transmitter. UART is the microchip with programming that controls a computer's interface to its attached serial devices.

UDP User Data-gram Protocol. UDP is a communications protocol that offers a limited amount of service when messages are exchanged between computers in a network that uses IP.

UI Unix International.

UNIK The Center for Technology at Kjeller. The acronym is for the norwegian name, which is Universitetsstudiene på Kjeller.

WWW World Wide Web.

Appendix B

Source code

B.1 rtl_dacapo.h

```
// The following statements are values that must be specified
// statically, before compilation.
#define MAXNODES 10 // maximum number of modules
#define MAXEDGES 20 // maximum number of inter-
                    // modular connections
#define MAXDATAQUEUES 10 // maximum number of data-queues
#define DATACELLSIZE 10 // the size (in bytes) of each data-cell
#define MAXDATACELLS 40 // number of available data cells
#define MAXSTREAMS 1 // max number of one-way streams
#define PARALLELLITY 2 // max number of modules in parallell
#define MAXNODESINGRAPH 10 // max number of modules in a graph
#define OUT 0 // data directions
#define IN 1
#define DEBUG

// the module data structure
struct node { // it's called 'node' because of a naming conflict
  int type; // AMODULE,CMODULE, or TMODULE
  // a thread is allocated for each module at start-up
  // this reference is set to that thread at that time,
  // and is never changed
  pthread_t task; // thread pointer
  int fifo;
  // when a module is allocated for a graph,
  // it's module function is also determined and set
  // the function might be replaced while the module
  // is in an active graph (dynamic reconfiguration)
  // when a module is deallocated,
```

```
  // it's module function pointer is reset
  void (*function)(void *); // module function pointer
  // these to ints stores how many queues the module
  // are connected with in each direction
  int up_queues; // number of queues before this node
  int down_queues; // number of queues after this node
  // and these are arrays that contain the actual
  // pointers to the data_queue structs
  struct data_queue *up_queue[PARALLELLITY];
  struct data_queue *down_queue[PARALLELLITY];
};

// the edge truct is used by the graph datastructure
// an edge represent an intermodular connection in
// a graph.
// up and down references the connected module via
// index values for the nodes array in the graph struct
struct edge {
  int up;    // index of the nodes
  int down;  // in the nodes array
};

// the graph datastructure
struct graph {
  int direction; // IN or OUT
  int nodesingraph; // the number of modules in the graph
  int edgesingraph; // the number of intermodular connections
  struct node *nodes[MAXNODES]; // references to all the
                                // modules in the graph
  struct edge edges[MAXEDGES]; // references all the edges
                               // in the graph
};

// the data queue datastructure
// the data queue is a linked list with FIFO behaviour
// the struct references the first and last members of
// the queue, plus the number of data cells currently
// in the queue
// the list itself is linked with the next pointer in
// the data_cell structs
struct data_queue {
  int status; // number of data cells currently in queue
  struct data_cell *first;
  struct data_cell *last;
```

```
};

// the data cell datastructure
struct data_cell {
  char cell[DATACELLSIZE];
  int used_bytes; // holds the number of bytes in the cell
                  // currently used to store data
  struct data_cell *next; // the next data_cell in a
                          // data queue
};

// module related functions
void init_nodes(void);
void cleanup_nodes(void);
void init_nodes_stack(void);
void node_free(struct node *);
struct node * node_alloc(void);

// data cell related functions
void init_data_cells_stack(void); // initialize LIFO list
void data_cell_free(struct data_cell *); // returns a data_cell
                                         // to the stack
struct data_cell *data_cell_alloc(void); // gets an available
                                         // data_cell from stack

// data queue related functions
void init_data_queue_stack(void); // initialize LIFO stack
void data_queue_free(struct data_queue *); // puts a data_queue
                                           // on the stack
struct data_queue *data_queue_alloc(void); // gets a data_queue
                                           // from the stack

// supporting functions
int AssembleGraph(struct graph *);
int DemolishGraph(struct graph *);

// module functions (will get separate header file)
void dummy(struct data_cell *); // test module function
```

B.2 rtl_dacapo.c

```
#include <rtl.h>
#include <pthread.h>
```

```
#include "rtl_dacapo.h"

// when the Linux kernel module is loaded:
int init_module(void) {

  // populate a sample graph for testing purposes

  struct graph testgraph;

  testgraph.direction=OUT;
  testgraph.nodesingraph=6;
  testgraph.edgesingraph=6;

  testgraph.edges[0].up=0;
  testgraph.edges[0].down=1;

  testgraph.edges[1].up=1;
  testgraph.edges[1].down=2;

  testgraph.edges[2].up=1;
  testgraph.edges[2].down=3;

  testgraph.edges[3].up=2;
  testgraph.edges[3].down=4;

  testgraph.edges[4].up=3;
  testgraph.edges[4].down=4;

  testgraph.edges[5].up=4;
  testgraph.edges[5].down=5;

#ifdef DEBUG
  printk("Loading Da CaPo for RTLinux.\n");
#endif
  init_nodes(); // allocate modules
  init_nodes_stack(); // set up module stack
  init_data_cells_stack(); // allocate data cells and
                           // setup data cell stack
  init_data_queue_stack(); // allocate data queues and
                           // setup data queue stack

  AssembleGraph(&testgraph); // build the testgraph

  return 0; // kernel modules must have return value=0
```

```
}

// when the Linux kernel module is removed
void cleanup_module(void) {
  cleanup_nodes(); // cleanup all threads
#ifdef DEBUG
  printk("Da CaPo for RTLinux unloaded.\n");
#endif
}
```

B.3 module.c

```
/*
 * This file should include all code that is logically
 * related to modules (nodes).
 *
 */

#include <rtl.h>
#include <pthread.h>
#include <time.h>
#include "rtl_dacapo.h"

#define __NO_VERSION__

static void * node_start_routine(void *);

/* --- */

static struct node nodes[MAXNODES]; // module struct objects
static int node_sp; // stack pointer !!MUTEX!!
static struct node *available_nodes[MAXNODES]; // the stack !!MUTEX!!

// module shell
// the function that is the start routine for module threads
static void * node_start_routine(void *mod_p) {

  struct node *i = (struct node*) mod_p;
  struct data_cell *data; // references the data cell
                          // currently being processed

// set scheduling parameters
```

```
  struct sched_param p;

  p.sched_priority = 1;
  pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
  pthread_make_periodic_np (pthread_self(), gethrtime(), 500000000);

  i->function=dummy; // for test purposes
  data=NULL; // for test purposes

  i->function(data);

  return 0;

  // Module loops on:
  //
  // - sleep until a data cell is ready in the in queues
  // - fetch that data cell
  // - perform module function on that data cell
  // - put data cell in out queue
}

// the function that initializes the modules by creating
// a module thread for each module
void init_nodes(void) {
  int i;

#ifdef DEBUG
  printk("Commencing module thread creation.\n");
#endif
  for (i=0;i<MAXNODES;i++) { // for each module module
    pthread_create(&nodes[i].task,NULL,node_start_routine,(void *)&nodes[i]);
  }
  return;
}

// release threads for clean exit from RTLinux
void cleanup_nodes(void) {
  int i;

  for (i=0;i<MAXNODES;i++) { // for each module
    pthread_delete_np(nodes[i].task);
  }
  return;
}
```

```
void init_nodes_stack(void) { // initialize LIFO stack
  int i;

  // none of the modules are in use at this time, so we simply
  // put all the nodes on the stack, one by one
  for (i=0;i<MAXNODES;i++) {
    available_nodes[i]=&nodes[i];
  }
  node_sp=MAXNODES-1;
#ifdef DEBUG
  printk("Module stack loaded.\n");
#endif
  return;
}

// module is no longer in use, register it as available
void node_free(struct node *noderef) {
  int i;

  // reset module
  noderef->function=NULL; // reset module
  for (i=0;i<PARALLELLITY;i++) {
      noderef->up_queue[i]=NULL;
      noderef->down_queue[i]=NULL;
  }
  available_nodes[++node_sp]=noderef; // put it on the stack
  return;
}

// allocate an available module
struct node * node_alloc(void) {
  return available_nodes[node_sp--];
}
```

B.4 data_cell.c

```
/* the data cells (memory space to hold packets during processing)
 *
 * Synopsis:
 * The modules pass the data to each other through data queues.
 * Each data-queue is comprised of a number of pointers to data-cells.
 * When a data-cell is in a data-queue, it is between processing,
```

```
 * waiting to be processed by the next module in the graph.
 *
 * Each data-cell can hold one data-packet. The packet is kept in the
 * same data-cell throughout the processing, to avoid expensive
 * copy operations. Each modules performs its function on a packet
 * in its data-cell, and when done puts a reference to its data-cell
 * in the data queue for the next module in the graph.
 *
 * When a data-cell is in a data-queue, its packet is between processing,
 * waiting to be processed by the next module in the graph.
 *
 * Implementation:
 * RTLinux requires that memory allocation be statically declared
 * and performed at init. In order to meet this requirement, we
 * statically declare a large number of data-cells. These data-cells
 * are recycled.
 *
 */

/* the available data_cells LIFO list
 *
 * Synopsis:
 * a LIFO list of data_cells that are currently not in use
 * All available (not-in-use) data-cells are put on a stack. When a
 * data-cell is requested for a module-graph, a pop is performed on
 * the stack. When a data-cell is no longer needed it is pushed back
 * on the stack.
 *
 * Implementation:
 * a linked list of data_cells. we only need pointers to the first and
 * last item in the list, thus we can use a data_queue struct and the
 * data_cells own next pointer to maintain the list together with new
 * pop and push functions that implements a LIFO list
 */


#include <rtl.h>
#include <pthread.h>
#include "rtl_dacapo.h"

#define __NO_VERSION__

// initialize data cells
static struct data_cell cell_table[MAXDATACELLS];
```

```
// the stack itself is really a linked list that is
// referenced using a data_queue struct
static struct data_queue data_cell_stack; // !!MUTEX!!

// initialize data cell stack
void init_data_cells_stack(void) {
  // no data cells are in use at this time,
  // so we put them all on the stack
  int i;
#ifdef DEBUG
  printk("Initalizing data cell stack.\n");
#endif
  for (i=0;i<MAXDATACELLS;i++) {
    data_cell_free(&cell_table[i]);
  }
#ifdef DEBUG
  printk("Data cell stack initialized.\n");
#endif
  return;
}

// return data cell to stack
void data_cell_free(struct data_cell *freed) {
  // add the data cell as the last one in the data cell stack
  // (which is really a list)

  // reset the freed data cell
  freed->used_bytes=0;
  if (data_cell_stack.last!=NULL) {
    data_cell_stack.last->next=freed;
  }
  data_cell_stack.last=freed;
  return;
}

// allocate data cell
struct data_cell *data_cell_alloc(void) {
  struct data_cell *allocated;
  allocated=data_cell_stack.first;
  data_cell_stack.first=allocated->next;
  allocated->next=NULL; // reset the data cell's next pointer
  return allocated;
}
```

B.5 data_queue.c

```
/* the data_queue struct
 *
 * Synopsis:
 * between each inter-modular relationship in a graph is
 * a data_queue
 * the data_queue holds all data_cells until the next module
 * in the graph is ready to process them
 *
 * Implementation:
 * the data_queue is implemented as a FIFO linked list
 * the data_cell objects contain a next pointer, thus the
 * data_queue object need only contain pointers to the first
 * and last element in the list
 */

/* the data_queue stack (LIFO stack)
 *
 * Synopsis:
 * it is a virtual stack in that we stack references to the
 * the data_queues, not the actual data_queues.
 *
 * Implementation:
 * as reference to a data_queue we use its index in the
 * data_queues array
 */

#include <rtl.h>
#include <pthread.h>
#include "rtl_dacapo.h"

#define __NO_VERSION__

// allocate the data queues
static struct data_queue data_queues[MAXDATAQUEUES];
static int data_queue_sp; // stack pointer !!MUTEX!!
static struct data_queue *data_queue_stack[MAXDATAQUEUES]; // the stack !!MUTEX!!

// initialize data queue stack
void init_data_queue_stack(void) {
  int i;
#ifdef DEBUG
  printk("Initializing data queue stack.\n");
```

```
#endif
  // none of the data queues are in use at this time,
  // so we put all of them on the stack, one by one
  for (i=1;i<MAXDATAQUEUES;i++) {
    data_queue_stack[i]=&data_queues[i];
  }
  data_queue_sp=MAXDATAQUEUES-1;
#ifdef DEBUG
  printk("Data queue stack loaded.\n");
#endif
  return;
}

// return data qeueue to stack
void data_queue_free(struct data_queue *freed) {
  freed->status=0; // reset freed data queue
  freed->first=NULL;
  freed->last=NULL;
  data_queue_stack[++data_queue_sp]=freed;
  return;
}

// allocate a data queue
struct data_queue *data_queue_alloc(void) {
  return data_queue_stack[data_queue_sp--];
}
```

B.6 management_toolkit.c

```
// supporting functions

#include <rtl.h>
#include <pthread.h>
#include <time.h>
#include "rtl_dacapo.h"

#define __NO_VERSION__

// build a module graph
// the function is used when setting up a module graph
// the module graph can be altered during its lifetime
int AssembleGraph(struct graph *graphref) {
  // allocate modules and data queues, and interconnect them
```

```
  // based on the structure stored in the graph struct
  int nodes=graphref->nodesingraph; // number of modules in the graph
  int edges=graphref->edgesingraph; // number of edges in the graph
  int i;

  // allocate nodes for the graph
  for (i=0;i<nodes;i++) {
    graphref->nodes[i]=node_alloc();
  }

  // allocate data queues for all edges and connect them with nodes
  for (i=0;i<edges;i++) {
    struct node *upnode, *downnode;
    struct data_queue *queueref;
    queueref=data_queue_alloc();
    upnode=graphref->nodes[graphref->edges[i]->up];
    downnode=graphref->nodes[graphref->edges[i]->down];
    upnode->down_queue[upnode->down_queues++]=queueref;
    downnode->up_queue[downnode->up_queues++]=queueref;
  }
#ifdef DEBUG
  printk("the number of nodes in graph is %d\n",nodes);
#endif
  return 1;
  }

// not implemented yet
//int DemolishGraph(struct graph *graphref) {
//}
```

B.7 module_functions.c

```
/* functions for the modules */

#include <rtl.h>
#include <pthread.h>
#include "rtl_dacapo.h"

#define __NO_VERSION__


// test function
void dummy(struct data_cell *data) {
```

```
  rtl_printf("Module %d: running\n",0);
  while (1) {
    pthread_wait_np ();
  }
  return;
}
```