A Middleware for Cooperating Mobile Embedded Systems

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von: Diplom-Informatiker Stefan Schemmer
geb. am 8. Oktober 1973 in Koblenz

Gutachter:
Prof. Dr. Edgar Nett
Prof. Dr. Jörg Kaiser
Prof. Dr. Andrea Bondavalli

Ort und Datum des Promotionskolloquiums Magdeburg, 12. Oktober 2004

Zusammenfassung

Die Kooperation mobiler eingebetteter Systeme eröffnet ein Spektrum neuer, vielversprechender Anwendungen in Gebieten wie der industriellen Automatisierung, Logistik, Telematik, und Team Robotik. Solche Applikationen unterliegen aufgrund der physikalischen Interaktionen zwischen den mobilen Systemen und ihrer Umwelt Echtzeitanforderungen. Eine zeitlich vorhersagbare Kooperation ist allerdings nicht ohne weiteres zu erreichen. Zum einen hängen die Ausführungszeiten lokaler Aufgaben und die Zuverlässigkeit des Kommunikationsmediums von der dynamisch sich ändernden Umwelt ab und können daher kaum vorhergesagt werden. Zum anderen besteht ein inhärenter Tradeoff zwischen einer effizienten Kooperation einerseits und der Autonomie der kooperierenden Systeme andererseits. Die Abhängigkeiten unter den Aktionen der mobilen Systeme können zu komplexen Interaktionen zur Laufzeit und damit zu einem schwer analysier- und vorhersagbaren Verhalten führen.

In dieser Arbeit wird eine Middleware vorgestellt, die Anwendungsentwickler bei der Lösung der oben genannten Probleme unterstützt. Auf den beiden unteren Schichten bietet sie Dienste zur zeitlich vorhersagbaren Multicast-Kommunikation und Ausführung lokaler Aufgaben. Beide Dienste greifen dabei nicht auf Worst-Case-Annahmen zurück. Um den inhärenten Tradeoff zwischen Kooperation und Autonomie aufzulösen und ein zeitlich vorhersagbares Verhalten auch auf der Ebene der kooperativen Anwendung zu erreichen, stellen die beiden oberen Schichten der Middleware koordinierte, gemeinsame Sichten für die Applikation zur Verfügung. Auf Grundlage dieser gemeinsamen Sichten entscheiden die mobilen Systeme lokal über ihr Verhalten, so dass auf der Applikationsebene eine Koordinierung zur Laufzeit nicht erforderlich ist. Die gemeinsamen Sichten beziehen sich auf applikationsunabhängige Aspekte des Kontrollsystems, wie etwa die Gruppenmitgliedschaft, und eher applikationsspezifische Aspekte des kontrollierten Systems, wie etwa die Position und Geschwindigkeit der mobilen Systeme. Zwei Applikationenszenarien dienten als Leitfaden des Designs und wurden prototypisch realisiert. Im ersten Szenario koordiniert eine Gruppe mobiler Systeme ihre Geschwindigkeiten, um einen kollisionsfreien und effizienten Zugriff auf eine räumliche Ressource zu realisieren, im zweiten fusionieren die Systeme ihre Sensordaten, um eine vollständigere und genauere Wahrnehmung ihrer Umgebung zu erreichen. Während der erste Prototyp demonstriert wie sich basierend auf gemeinsamen Sichten ein koordiniertes Verhalten auch mit lokalen Entscheidungen auf der Applikationsebene erreichen lässt, verdeutlicht der zweite Prototyp wie ein zeitlich vorhersagbares Verhalten auch unter hohen und variable Lasten erreicht werden kann.

Abstract

The cooperation of mobile embedded systems gives rise to new and promising applications in fields such as industrial automation, logistics, telematics, and team robotics. Such applications are subject to real-time constraints due to the physical interactions between the mobile systems and their environment. Achieving a timely predictable cooperation, however, is a challenging task. The execution times of the tasks and the reliability of the communication links depend on the dynamically changing environment and are hence hard to predict. Moreover, there is an inherent tradeoff between an efficient cooperation and the autonomy of the mobile systems. Managing the interdependencies between the actions of the mobile systems may result in complex runtime interactions, thus rendering the timing behavior of the application hard to analyze and predict, even if the underling services are timely.

In this thesis, we present a middleware supporting the application designers in overcoming the above-mentioned challenges. On the two lower layers, it provides timely predictable multicast communication and task execution. Both service are not based on worst-case assumptions. To resolve the cooperation / autonomy tradeoff and achieve a timely predictable behavior on the level of the cooperative application too, the two upper layers of the middleware provide common views to the application. Based on these common views, the mobile systems decide about their actions locally so that runtime coordination in the application can be avoided. Common views are available for application-independent aspects of the control system, like group membership, as well as for more the application-specific aspects of the controlled system, such as position and speed of the mobile systems. Two application scenarios guided the design of the middleware and have been implemented as prototypes. In the first scenario, a group of mobile systems coordinate their velocities to achieve collision-free and efficient access to a shared spatial resource; in the second, the systems fuse their local sensor data to accomplish a more complete and accurate perception of their environment. While the first prototype shows how a coordinated behavior can be achieved with local decisions on the application level basing on the common views the middleware provides, the second demonstrates how a timely predictable behavior can be achieved even under a high and variable processing load.

Acknowledgements

I am pleased to have the opportunity to say thank you to at least part of the people who contributed so much the completion of this thesis. First of all, I would like to thank Prof. Dr. Edgar Nett. In many fruitful and inspiring discussions, he not only significantly influenced the work presented herein, but also my way of working and thinking. I would like to thank the people in my working group “Real-Time Systems and Communication” for the important and the humorous talks, for proofreading this thesis, and most of all for the friendly atmosphere in our working group. A thankyou also to the students who did a great job in implementing part of the middleware and the prototypes. I owe thanks to my parents. Without their continuous support from my very first steps, this work would not even have been started. Most of all, I owe thanks to my wife, Kerstin, and Mira, my little daughter. There are no words for their patience and their continuous encouriging and inspiring support. They gave me the greatest gift I ever could wish for and the one thing that really counts in my life – their love.

Part of the work presented herein was conducted in the project “Middleware für kooperierende mobile Roboter” (Middleware for Cooperating Mobile Robots), which was supported by the Deutsche Forschungsgemeinschaft[1] (DFG) under grant NE 837/2-1.

[1] German Research Foundation

Table of Contents

1 Introduction

The cooperation of mobile embedded systems gives rise to a lot of interesting applications in many different fields, such as industrial automation, logistics, telematics, and team robotics. However, as is often the case with the most promising objectives, some challenging problems have to be overcome, before the envisaged applications will be reality. Amongst them, the following two are particularly pressing: How to achieve a reliable and timely predictable cooperation while the mobile systems are moving in a dynamically changing environment using lossy wireless links to communicate; and how to achieve a coordinated behavior of the cooperating systems while keeping them as autonomous as possible? The objective of this thesis is to develop a middleware for cooperating mobile embedded system that contributes to the solution of both of these problems.

1.1 Motivation

There is a sustained trend to embed computer systems in all kinds of intelligent products, such as photocopiers, cameras, telephones, cars, planes, and even razors. This trend is fuelled, on the one hand, by the miniaturization and cost-reduction of computing hardware and by the ever increasing demand for new and easier-to-use functionality on the other hand. Computing devices are becoming ubiquitous and pervasive to our every day live. Within this general trend there is a development to enhance the functionality of such systems beyond the provision of easy-of-use and comfort to more safety-critical tasks where they exert direct control over the intelligent product. This development is particularly interesting and clear for mobile embedded systems, such as robots and cars, in which the computer systems partially or completely control the motion of the controlled system. The increasing deployment of mobile embedded systems as well as the rising expectations on their functionalities, will sooner or later require the cooperation of such systems, e.g. for the resolution of spatial resource conflicts.

Coincidently with the long ranging trend outlined above, we witness the fast emerging and rapid deployment of wireless communication technology. The starting signal being given in 1999 when the IEEE approved its Standard 802.11, vendors soon started shipping interoperable products. Owing to its manifold appealing advantages — amongst which are the reduction of wiring costs, the mobility of users, the flexibility of networks, and the access to information resources anywhere at anytime — the technology was readily adopted so that nowadays wireless communication is a large and still fast growing market segment. One of its most interesting and promising advantages is its permitting the networking of mobile embedded systems. Like the networking of general purpose computers, in intranets and via the Internet, gave rise to a plenty of new applications, so will the networking of mobile embedded systems. Coming back to what we said above, it is this very possibility that renders the cooperation of mobile embedded system a true and realistic perspective for the future.

Today, there are cooperative applications for mobile embedded systems that are subject to industrial research and development already and others, being more ambitious, are envisaged for the future. Industrial automation and logistics represent two fields of application in which mobile embedded systems – like automated guided vehicles (AGVs) or destination coded vehicles (DCVs) – are already widely deployed these days. The cooperation of such systems gains increasing attention because it allows the mobile systems to coordinate their access to shared spatial resources, like crossings, thus achieving a better utilization of the resources and reduced waiting times for the mobile systems. In fact, coordinating the access the shared resources will become inevitable with an increasing number of such mobile systems being deployed. A concrete example being in use today already is a baggage transport system for airports consisting of a large number of rail-bound DCVs, each carrying a peace of baggage, that achieve collision avoidance cooperatively using wireless communication links. A more visionary example, with a very similar idea, however, is the cooperation of cars at hot spots, such as crossings or merging roads, to coordinate their access to the shared resource. Again, the cooperation promises improved utilization of the spatial resource and less queuing in front of it.

Another promising field of application is team robotics. That is why part of this work was supported by DFG within the project "Cooperating Teams of Mobile Robots in Dynamic Environments". Teams of robots are expected to fulfill tasks in such areas as factory automation, fire fighting, de-mining, contaminated areas, etc. The RoboCup, a soccer championship for robots, has been devised as a testbed for such applications. Every year, a lot of scientists gather during the contests showing the progress of their work. It turned out that cooperation of the robots using wireless communication is crucial to be successful in this arena.

1.2 Problem Exposition

In this thesis, we consider the cooperation of mobile embedded systems. We focus on groups of mobile systems operating in a common, local environment, using a single wireless medium for communication. The locomotion of the mobile systems and their physical interactions with the environment impose real-time constraints on the cooperation. In particular, when operating in a common environment, the mobile systems must coordinate their movements. Furthermore, fulfilling cooperative tasks such as cooperative sensing (e.g. sensor fusion) or acting (e.g. object transportation or manipulation) in a dynamic environment requires a timely predictable cooperation.

Local groups are a natural starting point for research on the cooperation of mobile embedded systems. For one thing, cooperative tasks with tight real-time constraints are typically performed in local groups, since on the one hand, real-time requirements frequently stem from the physical interactions of the cooperating system, and on the other hand, stronger real-time services can be provided to systems connected via a single wireless LAN. Moreover, solutions developed for the cooperation in local groups can serve as building blocks when considering cooperation in larger-scale networks. In particular, we consider larger-scale networks to consist of loosely coupled local groups, called cells in infrastructure networks and clusters in ad-hoc networks. Scaling the solutions presented herein to larger networks is subject to ongoing efforts in our working group (more details will be given in Chapter 7).

We found that the cooperative applications for mobile embedded systems share at least the following two challenges, which have to be tackled besides the genuine difficulties of the specific applications.

First, the mobility of the systems imposes real-time and reliability requirements — referred to under the common notion *Quality of Service (QoS)* in the following — that have to be met in a dynamically changing environment. For cooperative applications this means that both the execution of the local tasks as well as the communication must exhibit a timely predictable and reliable behavior. But, unpredictable, dynamically changing conditions, namely varying execution times of the tasks, and a varying number of message losses on the communication links, render this a hard task. It means that the resources required to completely execute a task or transfer a message to all its recipients are hardly predictable. As a consequence, the middleware cannot base the provision of QoS guarantees on known and tight worst-case bounds, as is done in conventional approaches. Doing so would be extremely inefficient, if possible at all. To free application designers from concerning these problems, we require the middleware to provide task execution and communication services that exhibit a reliable and timely predictable behavior in spite of the above-mentioned dynamically changing conditions.

Second, the inherent tradeoff between cooperation and autonomy has to be resolved. When mobile systems cooperate, their actions become mutually dependent. These dependencies have to be managed to achieve effective cooperation; that is, the actions of the mobile systems have to be coordinated (Malone and Crowston 1994). This means that the mobile systems no longer decide about their actions autonomously, but that they have to agree on a mutually consistent set of actions under time constraints. (Mock 2003) presented a formal framework to express these consistency and timing requirements. Without support from the middleware coordination must be achieved at the application level. This gives rise to complex runtime interactions of the mobile systems, which render the overall timing behavior hard to analyze and predict, even if the underlying communication and task execution services are timely predictable. The behavior of a mobile system can no longer be analyzed separately, but may be the outcome of some sophisticated and hardly observable interactions. It is up to the application designer to ensure that a mutually consistent behavior of the mobile systems is achieved in bounded time. Application designers (from fields such as logistics, industrial automation, automotive industry, etc.), however, are typically no communication or distributed systems professionals. So, whereas autonomy is desirable to keep system development and analysis simple, effective cooperation cannot be achieved as long as each system makes its own local decisions based on its own local worldview. Therefore, we require our middleware to support application designers in achieving a coordinated behavior while keeping the mobile systems as autonomously as possible on the application level.

Notwithstanding their above-mentioned commonalties, there is a wide spectrum of cooperative mobile applications with quite different demands. Demands differ, for example, w.r.t the deadlines of messages and tasks, the reliability required for communication and task execution, and the tightness of coordination that must be achieved. To accommodate different applications with different requirements, the middleware must be adaptable to their needs. This means that the set of services the middleware provides should be configurable to fit the needs of the application. Furthermore, it should allow tuning the QoS of these services to the actual demands of the application. This prevents producing overhead for such services or QoS that the application does not really need.

The objective of the work presented herein is to develop a modular middleware for cooperative mobile applications. The middleware is situated above a standard and off-the-shelf hardware and operating system. It provides services going beyond those of the underlying off-the-shelf components regarding support for the specific needs of cooperative mobile applications. In particular, it provides services supporting the application developer in solving the above-mentioned issues, thus allowing them to concentrate on the application-specific aspects of their tasks. Nevertheless, the middleware remains generally applicable for the indented spectrum of applications. To support analyzing the application requirements, guide the design process, and allow for evaluations in a real application context, the middleware is considered in the context of two application scenarios and their prototypical implementations.

Nowadays, object-oriented middleware, like CORBA and DCOM, is already widely deployed to build distributed applications (Microsoft Corporation 1996,Object Management Group 2002). This kind of middleware, however, addresses the heterogeneity of implementations and platforms and provides a location-transparent and object-oriented programming model. Both these topics are not in the focus of this thesis. On the other hand, this kind of middleware does not address the problems we address.

1.3 Approach

To address the three problems described above — the provision of QoS in dynamic environments, the autonomy vs. cooperation tradeoff, and the adaptation of the middleware to a variety of application demands — we adopt the following three approaches:

First, provision of QoS guarantees in dynamic environments: Facing varying execution times of the tasks and a varying number of message losses on the network, our approach is to avoid using worst-case bounds and work with more realistic estimates instead. This implies that there may be tasks instances with insufficient resources to be completed or messages with insufficient resources to be transferred to all intended recipients. We term this situation a resource fault. If not detected and handled adequately, a resource fault may lead to a task being completed or a message being delivered after its deadline. This not only results in an unpredictable timing behavior of the system, but also means allocating resources to tasks or messages that missed their deadline and will bring about no value (at best). Additionally, a faulty task or message instance may consume resources allocated to others, which means that the resource fault possibly propagates. Our approach is to guarantee a predictable timing behavior in case of resource faults; that is, the middleware ensures that no task is executed and no message transferred after its deadline. Furthermore, the middleware prevents propagation of resource faults so that they do not affect tasks or messages other than the faulty one. With this approach, there may be task instances not being completed and messages not received by all intended recipients. To address this problem, the middleware provides the means to maintain consistency of the system in case of message or task abortions. We believe, that with timeliness and consistency being guaranteed, many applications are able to tolerate aborted task or message instances to a certain degree. This can be achieved by exploiting several kinds of application-inherent redundancy, which we found to be present in cooperative embedded applications. In fact, knowing that completing all tasks and message transmissions may not be possible, the designer can provide these kinds of redundancy explicitly at design time.

Second, resolving the autonomy vs. cooperation tradeoff: Coordination is the key to effective cooperation, but poses some intricate problems to the application designer. Our approach is to achieve coordination in the middleware. Rather than coordinating the actions of the mobile systems on the application layer, we let the middleware coordinate their worldviews. To this end, the middleware provides agreed-upon common views to the application at the mobile systems. This includes common views on application-independent aspects of the control system as well as more application-specific common views on aspects of the controlled system, such as position and speed of the mobile systems. Based on these common views, the mobile systems decide about their actions locally using a coordinated set of rules. In this approach, the mobile systems act autonomously on the application layer. Nevertheless, they exhibit a coordinated behavior because the middleware achieves a coordination of their worldviews. Thus, complex runtime interactions at the application-level are avoided. The application designer is enabled to concentrate on the application-specific optimality of the decision rules and needs not be concerned with achieving agreement among the systems. Still, the local rules used by the mobile system must be coordinated. However, this coordination is much simpler to achieve since it is performed statically at design time and does not require any explicit coordination at run-time. The dynamic, run-time coordination is achieved by the middleware.

Third, adaptation of the middleware: Our approach is to design and build the middleware in a modular manner so that it is configurable to the demands of different applications. This can be thought of as a modular assembly concept, where you build your own specific middleware from a set of modular building blocks. This allows coming to a middleware that is tailored quite well to the application without designing a new middleware every time.

1.4 Overview of the Middleware

The middleware is located above a system layer consisting of commercial and off-the-shelf (COTS) hardware and operating system components. For the wireless network, we decided to build on the IEEE 802.11 Standard, which is commonly accepted and already widely deployed. It provides predictable medium access based on polling, which both the original 802.11 Standard as well as the upcoming supplement 802.11e include as an access method.

The middleware consists of four layers. The two bottom layers implement our approach to the provision of QoS in dynamic environments, whereas the two top layers implement our approach to resolve the autonomy-cooperation tradeoff. We designed the middleware in a modular manner so that it can be configured to comprise just those layers and services the application requires. In the following, we present the layers bottom-up.

In the bottom-most layer of the middleware, the scheduling for the CPU and network resources is located. Its main task is achieving timeliness. It comprises a task execution service based on the TAFT concept, which guarantees a timely predictable execution of tasks with unpredictable execution times (Nett and Gergeleit 1997,Gergeleit 2001). The service schedules the resource demands of the communication protocols of the middleware as aperiodic requests and allows for precedence constraints between periodic tasks. Furthermore, this layer comprises a dynamic network scheduling service. This service maintains the polling list according to which the underlying layer grants access to the medium. It allows mobile systems to request to be added to the polling list at runtime and removes system leaving the range of the wireless medium.

The second layer addresses the reliability of message transmission on the wireless network. It comprises a reliable multicast service. When sending a multicast, the user can specify an expected-case bound, called *resiliency*, on the number of message losses. The service guarantees timely transmission of the multicasts in spite of the varying number of message losses on the medium. To this end, it possibly aborts transmitting a multicast, in which case the next layer allows achieving consistency.

The third layer comprises three services providing common views on different application-independent aspects of the control system. The first service provides a common view on the global system time. The second is an atomic multicast service. It resides on top of the reliable multicast service and provides a common view on the multicast messages delivered. It ensures that all stations observe the same sequence of multicast messages in the same order, even if the transmission of messages is aborted in the layer below. Finally, the third service provides a common view on the membership of the group. It ensures that all group members deliver the same membership views in the same order and between the same two atomic multicast messages.

The three layers presented so far constitute the hardcore of the middleware. Based on the QoS and common views the hardcore provides, the Event Service at the highest layer of the architecture provides common views on the global state of the controlled system. The global system state consists of the local states of the mobile systems w.r.t to a common point of time on the global time base. The local states of the mobile systems comprise those state variables that serve as input for the control application; for example, position and speed of the mobile systems. The Event Service is part of a larger concept, called the *Cooperative Application Development Interface (CADI)* (Nett and Schemmer 2004). The CADI will comprise a family of services providing common views on different aspects of the control system's environment.

The cooperative mobile applications reside on top of the middleware. We consider two application scenarios, which we shall explain in the next section.

1.5 Application Scenarios and Prototypes

We present the middleware in the context of two application scenarios, both with a prototypical implementation. As example applications they provide valuable input during the design process (e.g. concrete requirements) and allow evaluations to be conducted in a realistic context. Both scenarios represent instances of common tasks to be performed by cooperating mobile systems so that their prototypical implementations are indeed lab-scale prototypes of interesting future applications. The scenarios were chosen to highlight different parts of the middleware, thus enabling us to present it in two steps. Whereas the first offers a good context for the presentation of our approach to the coordination among mobile systems and the corresponding parts of the middleware, the second features a high and varying CPU load so that it can be used to present our approach to the provision of QoS under unpredictable resource demands.

The first application scenario is a concrete example for a coordination task with tight real-time constraints. The general problem it addresses is coordinating the access to a shared spatial resource amongst a group of mobile systems operating in a common environment. In this application scenario, a group of track-guided mobile systems cooperatively coordinate their velocities at the intersection of two tracks, a so-called hot spot. The term "tracks" is meant here in a very general sense and may represent such things as rails, roads, or traces. The group changes dynamically with new systems approaching and group members leaving the hot spot. Its stringent requirements w.r.t the coordination of the mobiles systems and the timeliness and reliability constraints applying to its achievement render this scenario particularly interesting for our purposes. This application is mostly based on the communication services of the middleware, which provide the common views to the mobile systems and perform the dynamic scheduling of the network. It turned out that the middleware significantly simplifies the design of the application. Using the strong global state semantics of the Event Service, developing the application layer consisted of developing a scheduling function that each mobile system computes locally to determine a schedule of the hot spot from the global state of the group (position and velocity of all systems w.r.t. the same time on the global clock). Determining the global state, transmitting events to trigger the scheduling reliably and timely, and handling the dynamically changing group is accomplished in the middleware transparently for the application. This shows how our approach to the coordination of mobile systems simplifies application design by keeping the mobile systems autonomous on the application-level.

The second application scenario is a concrete example of a distributed fusion of sensor data within a group of mobile robots. We consider a group of mobile systems equipped with laser scanners that fuse their local worldviews to come to a more complete and accurate perception of their environment. The particular interest of this scenario stems from the high and variable processing load it puts onto the systems and the inherent redundancy the application exhibits. Hence, it is used to illustrate and validate our approach to enforcing QoS in dynamically changing environments. The focus is on the CPU scheduling, which is particularly important due to the environment-dependent execution times of the application tasks. The execution service achieves a timely predictable execution of the sensor data processing tasks in spite of their unpredictable resource demands. According to our approach, several levels of application-inherent redundancy are exploited to tolerate abortions of the tasks. The application prototype was used to examine how functional and structural redundancy can be exploited and to illustrate the concept of application-level adaptation under persistent overload.

We built up prototypes for both application scenarios. In the first, a group of trace-guided mobile robots coordinate their speeds at a hot spot where the traces overlap. The behavior of the robots at the hot spot shows that basing on the middleware a tightly coordinated behavior is achieved. The robots managed to avoid collisions in the running demonstration adapting their speeds according to the situation at the hot spot. Measurements show that the middleware provides delays sufficiently small for the intended applications. In the second prototype, a fusion of the output of several laser scanners is performed in a RobotCup-like setting. Measurements show that all instances of the sensor data processing tasks met their deadlines. Furthermore, it shows how task abortions are tolerated by exploiting the application-inherent redundancy.

1.6 Structure of the Thesis

In Chapter 2, we present the architecture of the middleware describing its layered architecture and the services it provides. Chapter 3 then takes a more application-oriented perspective and introduces the two application scenarios we use to present our middleware in an application context. These applications are chosen such that the middleware can be presented in two steps. In Chapter 4, we present the communication part of the middleware, which is used to achieve coordination in the first application scenario, whereas Chapter 5 is concerned with the scheduling of the local CPU resources, which is particularly important in the second scenario where complex sensor data are processed.

Chapter 4 describes the protocols implementing the communication services of the middleware. Before actually presenting the design of the protocols, we have to model the system in which the protocols are expected to run. The first sections of this chapter therefore consider the middleware and its system environment from a modeling perspective. Here, basic concepts and notions are introduced and a brief overview of the IEEE 802.11 Standard is given (Section 4.1). As well, a formal system model capturing the characteristics of groups of mobile systems connected by a wireless network is presented (Section 4.2). Having set the stage in sections 4.1 and 4.2, we explain the communication protocols of the middleware in Section 4.3. After giving an overview of the protocol stack as a whole, we present the protocols bottom-up, starting with a description of the underlying polling protocol and moving up the layers to the topmost layer, the Event Service. Formal descriptions of protocols in the communication hardcore are provided in the appendix. Related work is discussed in Section 4.4.

Chapter 5 is concerned with the scheduling of the local CPU resources. In Section 5.1, we analyze the problem of environment-dependent execution times in the context of the distributed sensor fusion prototype. We then describe the TAFT scheduling concept (Section 5.2), which we adopt to achieve a predictable timing behavior for tasks with environment-dependent execution times. Section 5.3 considers what kinds of application-inherent redundancy can be exploited to tolerate task abortions, using again the distributed sensor fusion scenario as an example to support our findings. To apply TAFT in our middleware, the task model underlying the current implementation has to be extended. To schedule the resource demands of the communication protocols, it must allow for aperiodic requests, and it must allow for precedence constraints between the task pairs to accommodate the requirements of the application tasks. Sections 5.4 and 5.5 successively show how this can be achieved. In Section 5.4, we present a scheduling algorithm, called TAFT-IPE, that schedules task sets consisting of periodic task pairs and aperiodic requests. We present acceptance criteria for both the periodic task pairs and the aperiodic requests. Section 5.5 shows how periodic task pairs with precedence constraints can be scheduled with TAFT-IPE.

In Chapter 6, returning to the application perspective, we describe the application prototypes we built. They corroborate the feasibility of our approaches as well as of the middleware that implements them. Additionally, it presents a modular implementation of the communication hardcore, which shows that a modular yet efficient design and implementation of the communication services can be achieved. Finally, Chapter 7 concludes this thesis and gives an outlook to future work.

2 Architecture of the Middleware

In this chapter, we present the architecture of the middleware, explaining its layers and services and how it can be tailored to the needs of a wide spectrum of applications (Schemmer et al. 2001,Nett and Schemmer 2003b,Nett and Schemmer 2004). Figure 2-1 exhibits the architecture of the middleware. The figure shows the middleware in its context, which consists of the system layer it is based on and the application layer it supports.

The system layer underlying the middleware consists of two sub-layers: the hardware layer and a basic operating system layer. The latter, among other things, provides basic access services for the CPU as well as the network resources. Standard and commercial and off-the-shelf (COTS) components are being used in the system layer. For the wireless network, we decided to build on the IEEE 802.11 Standard, which is commonly accepted and already widely deployed. To support the provision of QoS, it specifies an access method in which a central station, the so-called access point (AP), grants exclusive medium access through polling. This polling mechanism is part of the original 802.11 Standard as well as of the upcoming supplement 802.11e. We chose RTLinux as the operating system because (i) it has support for real-time performance (small, bounded interrupt latencies e.g.); (ii) its sources are open, so we can extend and modify it at the source code level if required; (iii) it is free; (iv) it is based on Linux, which comes with a full fledged development environment.

The middleware is divided into four layers, each with a specific task. The two bottom layers deal with the provision of QoS, implementing our approach to the provision QoS in dynamic environments. The lowest layer achieves timeliness for task execution and unreliable message transmission; the following layer accomplishes reliable multicast transmission while still preserving a timely predictable behavior. The two top layers implement our approach to the coordination of mobile systems. As pointed out above, this approach requires the middleware to achieve dynamic coordination among the mobile systems by providing common views to the application, the latter controlling the local actions of the mobile systems based on these common views. Both layers provide such common views – the third layer on general, control system internal aspects, e.g. what message have been delivered, and the fourth layer on aspects of the controlled system, such as position and speed of the mobile systems, which are more application-specific.

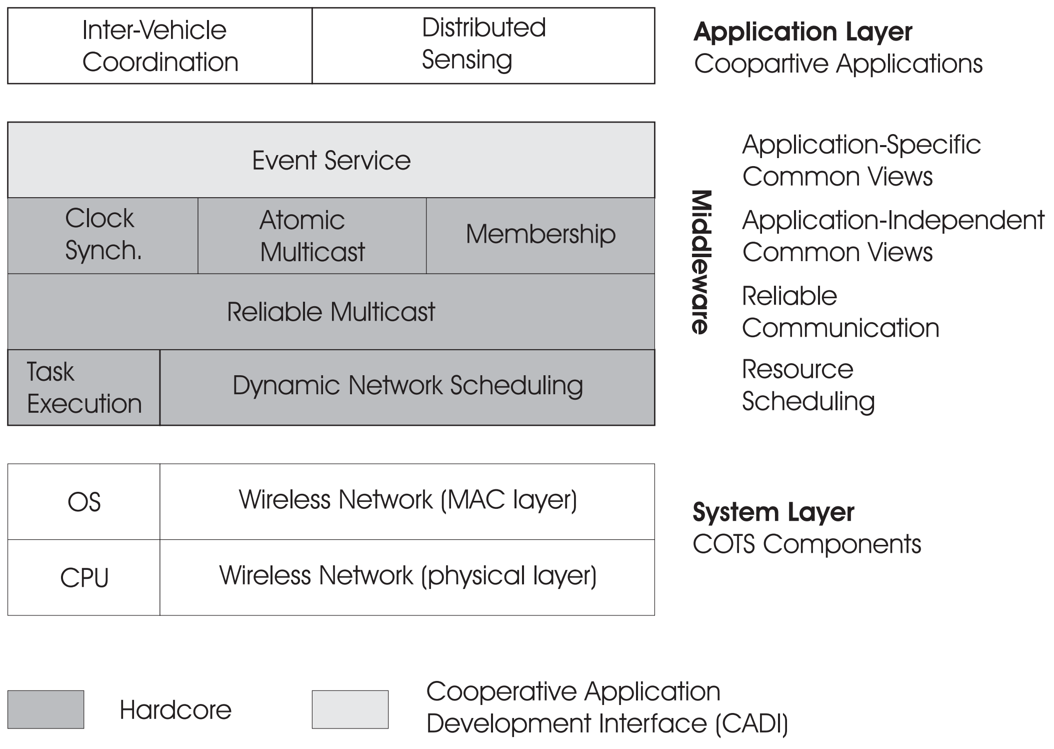

Inter-Vehicle Coordination
Distributed Sensing
Application Layer
Coopartive Applications
Event Service
Clock Synch.
Atomic Multicast
Membership
Reliable Multicast
Task Execution
Dynamic Network Scheduling
Middleware
Application-Specific Common Views
Application-Independent Common Views
Reliable Communication
Resource Scheduling
OS
Wireless Network (MAC layer)
CPU
Wireless Network (physical layer)
System Layer
COTS Components
Hardcore
Cooperative Application Development Interface (CADI)

Figure 2-1. Architecture and context of the middleware

2.1 Resource Scheduling

The first layer of the middleware is the adaptive resource-scheduling layer spanning both the network and processing resources. It comprises a task execution service that guarantees a timely predictable execution of tasks. In doing so, it must cope with their widely varying execution times. Therefore, we apply the TAFT concept to realize this service (Nett and Gergeleit 1997,Gergeleit 2001). TAFT allows using realistic *excepted-case execution times (ECETs)* instead of WCETs, yet still guarantees that no task instance misses its deadline. As long as a task instance does not exceed the specified ECET, it is completed before its deadline; if it exceeds the ECET, it may be necessary to abort it before its deadline. In this case TAFT ensures a timely exception handling, which allows keeping the system in a consistent state. As turned out in the sensor fusion application, we must assume a task model that allows for precedence constraints between periodic tasks and for aperiodic requests in addition to the periodic tasks. Dealing with aperiodic requests is necessary to support the execution of the communication protocols of the middleware, while precedence constraints arise if data are processed in a sequence of pipelined stages. We developed a scheduling algorithm for the execution service that implements the TAFT concept for such a task model. Acceptance criteria are provided for both the periodic tasks as wells as the aperiodic requests. While for the periodic tasks predictability is achieved on a per task basis, it is achieved on a per instance basis for the aperiodic requests.

The dynamic network scheduling handles requests for network resources and maintains the polling list. As the underlying polling mechanism grants medium access according to the polling list, the polling list represents the schedule of the medium. Mobile systems transmit their resource requests to the AP, which decides whether or not they are admitted to the polling list. The layer allows mobile systems to be added to the polling list in bounded time. Thus, it allows providing predictable medium access to a dynamically changing set of systems.

2.2 Reliable Communication

This is the second layer concerned with achieving QoS. It addresses the reliability of message transmission on the wireless network.

It comprises a service for the reliable transmission of multicast messages. The multicast semantics facilitates the cooperation in groups of mobile systems; messages are not addressed to single recipients but to all members of the group. In fact, considering multicast instead of point-to-point messages is already a first step towards providing common views on the following layers. To achieve reliability, it uses a dynamic redundancy approach, where messages are retransmitted when message losses are detected. The service has to cope with a varying number of message losses on the wireless medium, which means that the number of necessary retransmissions is hardly predictable. It does, therefore, not require specifying worst-case bounds on the number of message losses. According to our approach, the user can specify an expected-case bound, called *resiliency*, instead. The service guarantees timeliness of messages transmission in spite of the varying number of message losses. After up to resiliency retransmissions the service aborts the transmission of the message in order to avoid a later delivery of the message and waste of resources. In this case, the next layer, which is presented in the following section, allows achieving consistency. Choosing a resiliency allows the application to decide in the tradeoff between reliability and timeliness. Choosing a small resiliency reduces the resource demand so that a shorter deadline can be met while it also increases the probability of some of the intended recipients not receiving the message.

2.3 Application-Independent Common Views

This layer in conjunction with the next one realizes our approach to the coordination of mobile embedded systems. The key issue this layer addresses is providing common views on application-independent aspects of the distributed control system. It comprises three services providing such common views.

The first service provides a common view on the global system time. An existing protocol, developed in our working group and at Fraunhofer AiS, was integrated into the middleware to synchronize the local clocks of the cooperating systems (Mock et al. 2000b,Mock et al. 2000a). It achieves a high precision and a continuous adjustment of the local clocks. The second service is the atomic multicast service. It resides on top of the reliable multicast service and provides a common view on the multicast messages delivered. It ensures that all stations observe the same sequence of multicast messages in the same order. In particular, it achieves consistency when the reliable multicast service aborts the transmission of messages in the layer below. An aborted message may have been received by some of its intended recipients already. The atomic multicast service ensures that none of the intended recipients delivers the message in such cases. Finally, the third service provides a common view on the membership of the group. The service provides up-to-date views on the current set of correct group members. It ensures that all group members deliver the same membership views in the same order, so share a common view on the current membership. Additionally, it ensures that all members deliver a membership view between the same two atomic multicast messages. So, the members agree on the position of membership changes within the stream of totally ordered multicast message; or, to put it the other round, they agree on the membership view in the context of which a multicast messages is delivered.

The three layers presented so far constitute the hardcore of the middleware. In providing their services, the protocols in the layer above it rely on the QoS and the common views the hardcore provides.

2.4 Application-Specific Common Views

On the highest layer of the middleware resides the *Event Service*. The Event Service provides common views on the global state of the controlled system, that is, the group of cooperating mobile systems. The global system state consists of the local states of the mobile systems w.r.t a common reference time on the global clock. The local state comprises those state variables of the mobile system that serve as input for the control application. For example, consider a group of mobile systems coordinating their behavior at a shared spatial resource. In such a scenario, the local states of the controlled systems are their position and speed (and maybe others). So, a consistent global state would comprise the positions and speeds of all participating mobile systems w.r.t the same point of time on the global time base. Rather than maintaining the global state all of the time, the Event Service determines the global state when it delivers an event to the application. When the application detects an event that requires coordinated actions of the mobile systems, it calls the Event Service to propagate the event to the group. The Event Service associates with each event it delivers a consistent global state. Thus, when an event is delivered at a mobile system, the control application can use the global state as input to decide locally about the action of the mobile system. Yet, the mobile systems in the group will exhibit a coordinated behavior since all local decisions are based on a common view on the global state. Since there are no interactions between the mobile systems on the application level following the delivery of the event, a timely predictable reaction to the event can be achieved basing on the timeliness properties the execution service provides. The semantics of the Event Service are a combination of event and state messages. State information is advantageous in dynamically changing groups, as new members frequently lack the context to understand event information. Global states are a strong abstraction for the application, which actually hides all communication efforts of the communication hardcore from the application.

The Event Service is part of a larger concept, called the *Cooperative Application Development Interface (CADI)*. The CADI is intended to provide common views on the environment of the control system, in contrast to the services below it, which provide common views on control system internal aspects. Further services for the CADI are under development in our working group (cf. Chapter 7).

2.5 Modularity

To be adaptable to the demands of a wide spectrum of applications, we designed the middleware in a modular manner. Applications need not use all the layers and services. The middleware can be configured to comprise just the bottom-most layer, the two bottom-most layer etc., up to any of the four layers. As well, it need not comprise all services of the chosen top layer. For example, if an application only needs timely predictable point-to-point messages with a varying set of station on the wireless medium, only the dynamic network scheduling may be used. It is also possible to omit the dynamic network-scheduling service and still use the communication services on the higher layers if a dynamic access to the medium is not required. Additionally, there is a vertical distinction between the task execution and the communication services. If an application is more communication centric with only little local computation, the task execution service may be omitted from the middleware and the pure OS scheduler be used instead. Likewise, one can imagine using the execution service and omitting the communication services. For the communication protocols in the hardcore, the modular design not only allows adapting the middleware, it also renders the protocols simpler and easier to analyze and permits reusing an existing protocol when realizing new services.

3 Application Scenarios

3.1 Coordinating the Access to Shard Spatial Resources

With several mobile systems operating in a common environment, coordinating the access to shared spatial resources becomes a general problem that must be solved to avoid collisions and achieve a good utilization of the available space. This is particularly the case when considering track-guided vehicles, like robots following traces, transport systems moving on rails, or cars driving on roads. Intersections of the tracks, which we refer to as *hot spots*, are critical shared resources that have to be shared efficiently so as to avoid congestion in front of the hot spot. Semaphore-like approaches (e.g. traffic lights) that force the mobile systems to stop when the hot spot is busy do not achieve this kind of efficient coordination. In our application scenario, the mobile systems approaching a hot spot form a group and coordinate their access to the hot spot cooperatively (Schemmer et al. 2001,Nett and Schemmer 2003b). In his formal framework, (Mock 2003) shows that this scenario requires coordination under tight real-time constraints. Therefore, the scenario is well suited to exemplify our approach to the coordination of mobile systems and to serve as a field of application for the communication part of our middleware. We built up a prototype of the application scenario, which will be presented in Section 6.1.

As an example with a real application background, consider a baggage transport system for airports. In this system, rail-bound vehicles, so-called destination coded vehicles DCVs, transport the baggage from the check-in desks through several stages of security screening and finally to the terminal where they are loaded onto the planes. Each DCV carries a single piece of baggage to a series of destinations. Knowing its route, it autonomously steers through the railroad system. One of the key advantages of this system is the high speed of the DCVs (up to 10m/s), which significantly outperforms traditional conveyer belts. To avoid collisions, each DCV coordinates with its predecessor using wireless communication. The predecessor periodically sends its current position to the successor, which is thus enabled to keep a fixed safety distance. Although a centralized approach is currently being pursued for the junctions, this example can be directly mapped to the scenario at hand if each junction is considered as a hot spot requiring coordination of the approaching DCVs.

According to the approach taken in our application scenario, the DCVs would coordinate their speeds at the junctions cooperatively. Such a cooperative solution would be more scalable and achieve a better utilization of the junctions.

More visionary, intelligent driving assistants in cars will cooperate to coordinate their access to shared road space such as crossroads and merging roads. Handling the always-increasing traffic by building new roads and highways will sooner or later meet its natural limits. Therefore, it is of utmost importance for the economy and the personal mobility that the existing road space will be used more efficiently. The problem of poorly utilized road space partly stems from the brute force methods, e.g. traffic lights and stop signs, which are used to avoid collisions at shared road space, such as crossroads and merging roads. According to the approach taken in our application scenario, the driving assistants in the cars form a group and schedule the shared road space cooperatively. As compared to the semaphore-like approaches mentioned above, scheduling the shared road space does not force the cars to stop while the resource is busy, but allows them to go on with a possibly reduced speed. Thus, utilization and throughput of the resource can be increased.

3.1.1 Description of the Scenario

To model the scenario outlined above, we are considering two one-way tracks that merge and then divide again. The intersection of the tracks is shared by the mobile systems on both tracks. The hot spot is an area of a given length d_m at the beginning of the intersection (depicted dark gray in Figure 3-1). The hot spot represents the shared resource for which mutual exclusion is to be achieved; that is, at any time there has to be at most one mobile system in the hot spot, even if it covers a part of the hot spot only. This ensures that there is a safety distance of at least d_m between a mobile system leaving the hot spot and any following system. The mobile systems approach the hot spot at different speeds and may change tracks while in the hot spot. As can be seen in Figure 3-1, this scenario represents joining as well as crossing tracks. If two mobile systems approach the hot spot on different tracks and both stay on their own track, the situation corresponds to traversing a crossing (cf. Figure 3-1.a and Figure 3-1.b). If one of the mobile systems changes the track, the situation represents two mobile systems driving on merging tracks (cf. Figure 3-1.c and Figure 3-1.d). On both tracks there is an *approaching zone* (the light gray zones in Figure 3-1) starting at a distance d_a in front the hot spot. The maximum speed at which a mobile system may enter the approaching zone and the length of the latter must be chosen such that d_a is greater than the braking distance of the mobile system.

All mobile systems in the approaching zones plus the one in the hot spot cooperatively determine a schedule for the hot spot. A schedule for the hot spot contains for each of these systems a tuple $(t_e,t_l) \in \mathbb{T}^2$ denoting the times at which the system enters and leaves the hot spot respectively, where $\mathbb{T} := \mathbb{R}_0^+$ denotes the set of points of time. Although the system in the hot spot is not subject to rescheduling, there is a tuple for it in the schedule to express the time at which it will leave the hot spot. The mobile systems must agree on this schedule; that is, the enter and leave time of each mobile system can be considered as part of the same global schedule. The main consistency constraint that this schedule must fulfill is mutual exclusion, which means that the intervals $[t_e,t_l]$ of any two stations do not intersect.

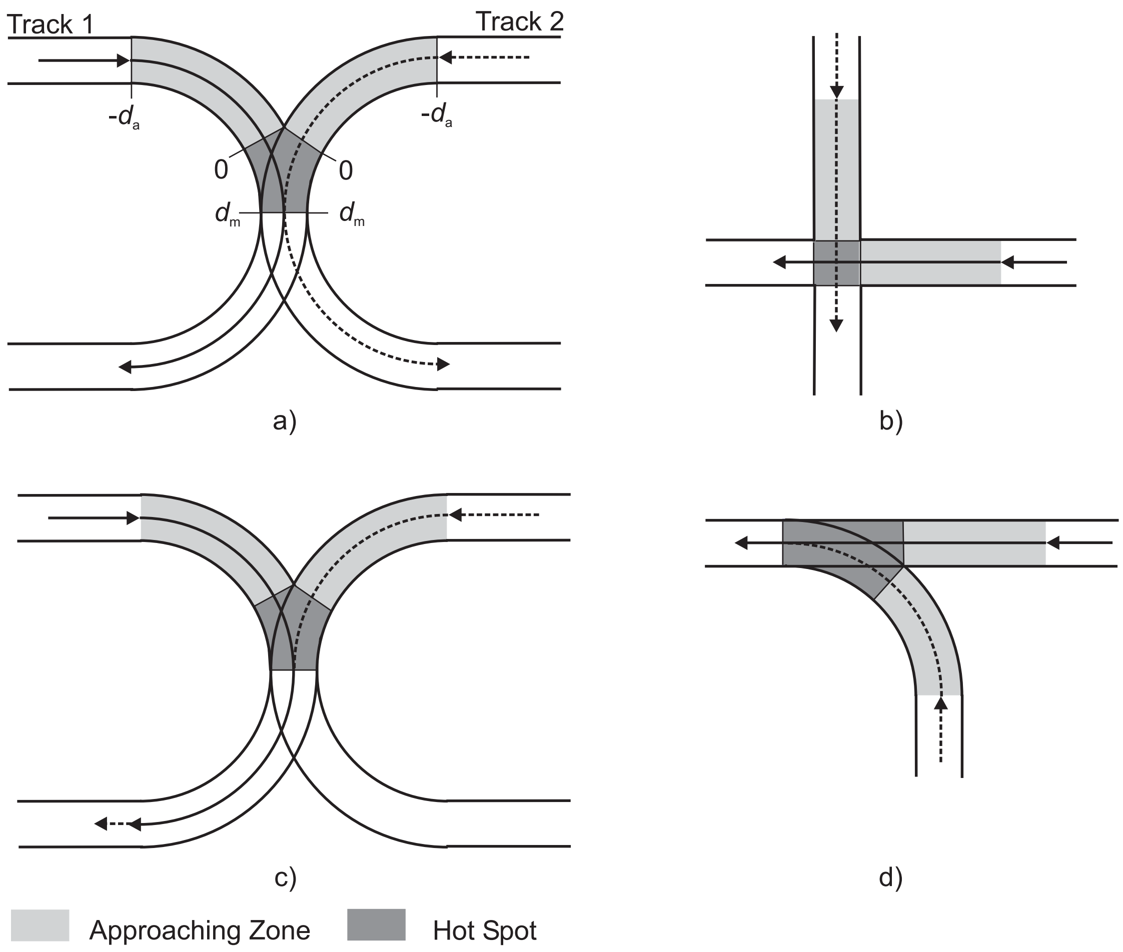


Figure 3-1. The scenario and corresponding real life situations

Apart from the consistency constraints, the schedule must be implementable — it must comply with the physical limitations of the mobile systems, such as bounded acceleration and speed, — and it must meet constraints and objectives the application imposes. As the mobile systems are approaching the hot spot while determining the schedule, agreement on a consistent schedule must be achieved in bounded time. In particular, whenever a further system enters an approaching zone, the last computed schedule is no longer complete since it does not specify enter and leave times for the new system. Hence, a new schedule must be computed. The new system must learn its scheduled enter and leave times sufficiently early so that it is able to implement them.

3.1.2 Application Architecture

Figure 3-2 depicts the architecture of the application. It reflects our approach to the coordination of mobile systems. All processing on the application level is executed locally on the mobile systems with the Event Service achieving coordination by providing a common view on the system wide global state.

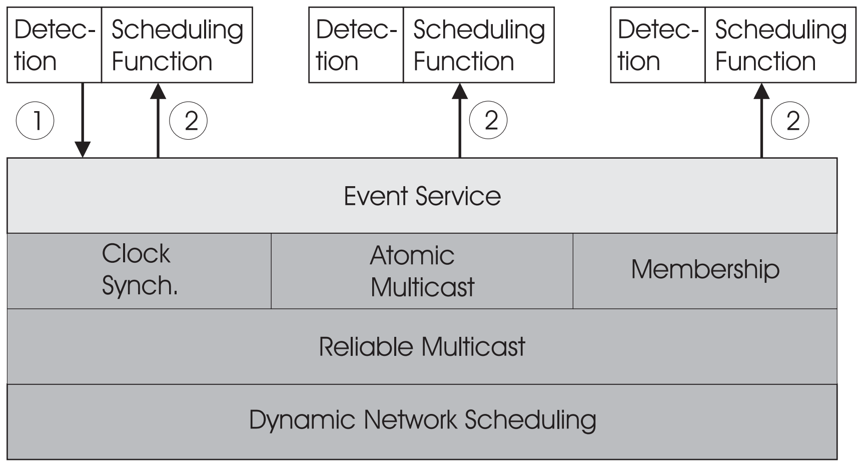


Figure 3-2. Architecture of the shared spatial resource application (system layer omitted)

In this architecture coordinating the access to the hot spot works as follows. First, the mobile systems must detect that they are approaching a hot spot. This can be accomplished using a global positioning system or landmarks for example. Detecting that it is entering an approaching zone, the mobile system knows it has to coordinate with the other approaching systems. It calls the Event Service to propagate this event and trigger the necessary global rescheduling (Figure 3-2, arrow 1). The Event Service delivers the event to all mobile system in the approaching zones, including the one currently arriving (Figure 3-2, arrow 2). Together with the event it delivers a consistent global state to the application. Delivering the trigger event, all mobile systems compute a schedule for the hot spot based on the global state associated with the event.

To define in more detail what the global state of the controlled system is in this scenario, we introduce the following notions. Let $\mathcal{S} := \{s_i \mid i \in \mathbb{N}\}$ be the set of mobile systems. For each mobile system s_i we denote its position and velocity at time $t \in \mathbb{T}$ in the following way. System s_i's position $p_i(t)$ is denoted by a tuple $p_i(t) := (d_i(t), r_i(t)) \in \mathbb{R} \times \{1,2\}$, $|d_i(t)|$ is the distance of s_i to the start of the hot spot, with $d_i(t) < 0$ in front of the hot spot and $d_i(t) > 0$ within and after the hot spot. $r_i(t)$ is the identifier of the track the system is driving on (cf. Figure 3-1.a). With $d_i(t)$ defined as above, system s_i has a non-negative velocity $v_i(t) = d_i'(t)$ when it is driving towards the hot spot. We assume that s_i's velocity is bounded by $v_{\max}$. The local state $z_i(t) := (p_i(t), v_i(t))$ of s_i at time t consists of position and speed of s_i at time t. The group $\mathbf{g}(t) := (s_{i_1}, \ldots, s_{i_n})$ contains exactly those mobile systems s_i for which $d_i(t) \in [-d_\mathrm{a}, d_\mathrm{m}]$; that is, those mobile systems that are in an approaching zone or in the hot spot. The global state $\mathbf{z}(t) := (z_{i_1}(t), \ldots, z_{i_n}(t))$ of this group is the vector of the local states of the group members.

Based on this global state, each mobile systems determines a schedule for the shared resource by a locally computed function, the so-called scheduling function. The input of the scheduling function is the global state and its output is a schedule comprising enter and leave times for all mobile systems that are part of the global state. The scheduling function is completely local and incurs no runtime interactions between the coordinating systems.

Thus, a timely predictable execution of that function can be achieved by local scheduling. Furthermore, the application designer can concentrate on developing the scheduling function such that it fulfills the consistency constraints and the optimization objectives of the application. A concrete example of a scheduling function has been presented in (Schemmer et al. 2001). From the schedule, each mobile system knows when to enter the hot spot and it can adjust its speed accordingly.

A mobile system's leaving the hot spot works similar. The mobile system either explicitly calls the Event Service or its leaving is detected by the underlying membership service. In both cases, the Event Service delivers an event and a consistent global state so that a re-scheduling can be initiated on the application layer.

It is important to note that detecting the approaching zone, computing the schedule, and adjusting the speed are all local actions. No coordination is performed on the application level at runtime. The mobile systems achieve a coordinated behavior nevertheless, because the dynamic coordination is achieved in the Event Service. As the above description shows, this scenario lays stress upon the communication part. It requires the mobile systems to achieve a tightly coordinated behavior in real-time. To this end, it employs all the communication services the middleware provides. CPU scheduling, on the other hand, is practically not a problem since neither detecting the approaching zone nor computing the schedule requires that much processing resources.

3.2 Distributed Sensor Fusion

Perception of the environment is one of the most important skills of mobile systems as it is the basis for the selection of actions. Wrong or partial perceptions may lead to inappropriate and dangerous actions being taken. Mobile systems use sensors to perceive their environment. However, a single system always perceives a situation from a single point of view, and the data its sensors deliver will not reflect the whole situation in general. Cooperation of mobile systems bears the potential to alleviate the problem. In the scenario at hand, we consider a group of mobile systems performing a distributed fusion of their sensor data (Nett and Schemmer 2003a). Each system multicasts the preprocessed output of its local sensors and makes it available to the other group members. Afterwards, each robot locally fuses the data it received. This scenario incurs a significant amount of local sensor data processing so that in addition to the communications services the task execution service plays an essential role here. We designed the application to comprise several kinds of inherent redundancy, at least part of which we believe are present or can be furnished in most cooperative mobile applications. Furthermore, it allows analyzing the typical structure of distributed sensor data processing and hence determining the characteristics of the task sets to be scheduled. Thus, it is well suited to exemplify our approach to achieving QoS for the local task execution and to present the scheduling algorithm implementing this approach.

The RoboCup can be considered as an example application. The RoboCup, a robot soccer championship, has been devised as a common benchmark for the cooperation of mobile robots in dynamic environments (Kitano et al. 1997). In particular, it was considered as an application scenario within the DFG project in the context of which part of the work presented herein was conducted. As turned out, fusing the worldviews of the teammates is crucial for a good team performance. Several RoboCup-Teams already perform a distributed sensor fusion for the ball and opponent localization (Dietl et al. 2001,Schmitt et al. 2001,Stroupe and Balch,Schmitt et al. 2002). In our work, we focus on the timing and communication aspects of the distributed sensor fusion.

3.2.1 Description of the Scenario

In our scenario, a group of robots performs a distributed fusion of their laser-scanner data. Each robot multicasts the preprocessed output of its laser-scanner and makes it available to all the group members. Each robot fuses the data it receives locally and makes it available to the application, possibly after some further local processing.

Each robot transform the raw data (a set of points) successively into more abstract representations, namely contours, geometric objects, and real world elements. The filters that perform these trasformations compose a pipeline for the processing of the sensor data. For each representation, a fusion module has been developed that processes data from multiple sensors at that level and outputs data at the same level combining all processed input data. The output of the fusion module can be fed into further filtering stages. Figure 3-3 depicts the resulting structure. Each robot is able to execute each filtering and fusion module locally. At runtime, each robot filters the data to the chosen level of abstraction (contours in Figure 3-3), multicasts it on the network (the bold horizontal line in Figure 3-3), feeds the data it receives into the fusion module at the chosen level, and filters the output of this module up to the level of interest.

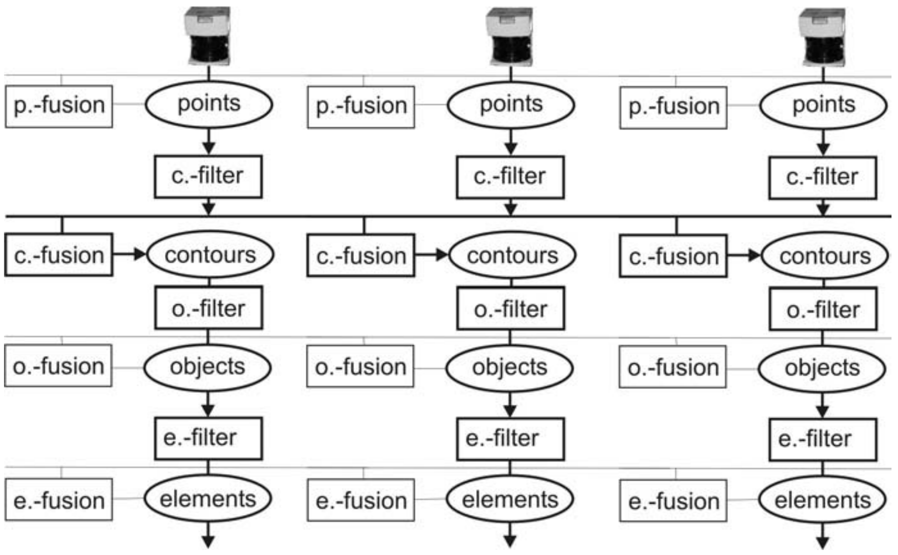


Figure 3-3. Structure of the sensor fusion (fusion on contour level)

This application contains several kinds of inherent redundancy. First, we developed all filter and fusion modules as any-time algorithms (Dean and Boddy 1988), which produce first, vague results as early as possible and then refine them iteratively (see Sub-Section 5.3.1 for more details). Even if being terminated before their completion, they provide some preliminary results. Thus, there is functional redundancy in the task instances in the sense that they may provide sufficient results even if not executed completely. Second, this application contains structural redundancy by its very nature. Whenever several robots observe the same part of the environment and one of them fails to deliver some information, the other robots can provide the missing information. In fact, this is done automatically by multicasting and fusing the local sensor data. Third and finally, the level of abstraction at which the fusion is performed can be changed. Changing the level impacts the amount of data to be transmitted and the input size of the fusion module. Generally, more abstract representations are more compact and require less processing resources. For example, representing a scan by a set of real world elements, such as balls and robots, and their positions needs by far less data than representing it as a set of points. Therefore, fusing data on a higher level of abstraction reduces the amount of resources the application requires. However, it typically implies less accurate results also. Thus, by changing the level of abstraction at which the fusion takes place, the application can adapt its resource demand at the price of a reduced accuracy. This represents another level of functional redundancy, which, as compared to using any-time algorithms, requires explicit action of the application at runtime.

In the following, we present the filtering and fusion stages in more detail.

3.2.1.1 Filtering

In the following, we briefly present the representations we use on the different levels of abstraction and the filters performing the transformations between these representations. We refer to the set of data representing a set of raw data at whatever level of abstraction as a scan. For our current implementation, we consider the RoboCup as an application example, such that the environment of the robots consists of robots, a ball, and the boards (which are replaced by white lines and rows of poles according to the latest rules).

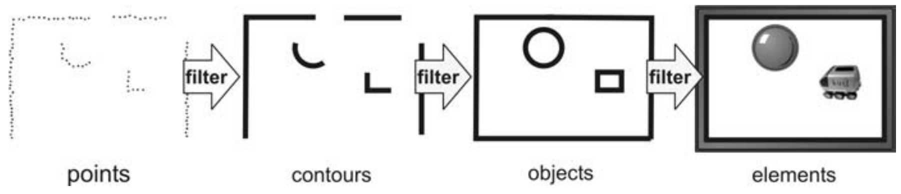


Figure 3-4 Representations and filtering

The following representations are used in the scenario (cf. Figure 3-4):

Points. Points are the representation that is delivered by the laser scanner as raw data. The scan is represented by a sequence $P = \{p_1, \ldots, p_k\}$ of points, where each p_i is a pair (x_{p_i}, y_{p_i}) of the Cartesian coordinates of the point in the scanner's coordinate frame.

Contours. On the contour level the scan is represented by a set $C = \{e_1, \ldots, e_m, a_1, \ldots, a_n\}$ of edges and arcs. Each edge is represented by tuple containing both of its vertices, each arc by its center point and radius.

The contour filter transforms raw data into contours in three steps:

1. Arcs are extracted. The algorithm we developed allows extracting arcs with known radii. Basically, the idea is to move a circle of a given radius along the measured points and count the number of points that are within a tube-like environment of the

circle. If a certain threshold is exceeded, these points constitute an arc. The width of the tube accounts for the inaccuracy of the measurements.

2. The points are converted to sequences of segments, each connecting to successive points. Each such sequence corresponds to a sequence of connected edges. A new sequence is started whenever the distance between to neighboring points exceeds a threshold indicating the start of new object contour (similar to anchor point detection in (Weber et al. 2000)).

3. Segments are combined to edges. The algorithm starts by representing a whole sequence of segments by a single edge and then successively partitions the edges until they approximate the segment sequence sufficiently good (The Algorithm is described in (Knieriemen 1991).)

Objects. On the object level the scan is represented by a set $O = \{po_1, \ldots, po_r, c_1, \ldots c_s\}$ of r polylines and s circles. Each polyline consists of a sequence $(p_{r1}, \ldots, p_{rn_r})$ of points, each circle of a center point and a radius.

The object filter first transforms arcs into circles. In the considered environment, each arc belongs to a circle. Therefore, the filter needs only check if several arcs correspond to a single circle, in which case all but one of them are removed, and transform the remaining arcs into circles.

The second step is to combine adjacent edges to polylines. This is performed by successively adding edges to a polyline as long as one of the end points of the edge is sufficiently close the one of the end points of the polyline. If no further edges can be added, a new polyline is started using the first edge that has not yet been used.

Element Filter. On the element level a scan is represented as a set of elements $E = \{e_1,\ldots,e_t\}$ each of which has two components: the element type (robot, ball, or wall) and a point denoting the elements position.

The filter represents each object by a vector of two features: the object type (circle or polyline) and the circumference of its axis-aligned bounding box. The set of all possible vectors constitutes the 2-dimensional feature space. A region in the feature space represents each element type. The size of a region depends on the maximum and minimum size of the element's bounding box and of the inaccuracy that should be tolerated. An object can be classified as being of a certain element type if its feature vector is located in the corresponding region.

3.2.1.2 Fusion

In the following the sensor fusion algorithms for the four levels of abstraction are presented.

The first step of all algorithms is to transform the coordinates of the considered entities (points, edges) to a common coordinate frame, usually, the coordinate frame of one of the sensors. As we assume that position and orientation of the sensors are known, this is easily achieved by simple coordinate transforms.

In the following, we describe how the merged representations are processed such that they appear as being observed by a single virtual sensor.

Points. To resemble the output of a real laser-scanner, the merged set of points must be sorted by increasing angles (with respect to the abscissa). However, this is not sufficient. The sequence of points is sorted in several runs to ensure that points belonging to the same geometrical primitive are neighbors. This is not necessarily true if the output of several sensors is merged, because points belonging to different arcs or edges may be interleaved, if one of this primitives is located behind the other in the common coordinate frame.

Contours. For the merged set of contours, it is checked if several primitives can be combined and represented by single primitive. Two arcs are combined if the distance between their centers and between their radii is sufficiently small. Two edges are combined if they are sufficiently parallel (the angle between the vectors indicating their direction is less than a threshold) and sufficiently close. The two edges are replaced by a new edge having the two end points with the longest distance as its end points.

Objects. Similar to the fusion of contours, in the object fusion, it is checked if several objects in the merged object set can be replace by a single one. For circles this is achieved in the same way as for the fusion of arcs. Two polylines are combined if their bounding boxes touch or overlap. Basically, this is achieved by concatenating the vertex lists of both polylines. There are, however, cases that are more complicated but have been omitted here for space limitation reasons.

Elements. For the fusion of elements no geometrical computations are necessary. Only redundant elements, i.e. objects of the same type at about the same place, are eliminated from the merged element set.

3.2.2 Application Architecture

Figure 3-5 depicts the architecture of the sensor fusion. As compared to the previous scenario, the middleware comprises a reduced set of communication services, namely the Event Service and the membership service have been omitted, but the task execution service is employed this time. The task execution service is particularly important here due to the widely varying execution times the application tasks exhibit (Measurements showing these variations will presented in Section 5.1). Representing the mobile systems, the light gray boxes illustrate that the application tasks (the white boxes) are executed by the task execution service (Figure 3-5, (1)). In fact, there are more tasks in the system than the figure displays; all of the filter boxes actually represent a number of filtering stages, each realized as a separate task pair. As said above, all filtering and fusion stages have been designed as any time algorithms. When a task pair executing such an algorithm is aborted, the exception part delivers the results computed so far to the following stage of the processing pipeline. The data exchange between successive stages of the processing pipeline implies precedence constraints between the corresponding task pairs. The execution service's being partly located beneath the communication services is intended to illustrate that the CPU demands of the communication protocols require CPU scheduling as well. The execution service has been designed to cope with both the demands of the communication protocols and the precedence constraints of the task pairs, as will be explained in sections 5.4 and 5.5 respectively.

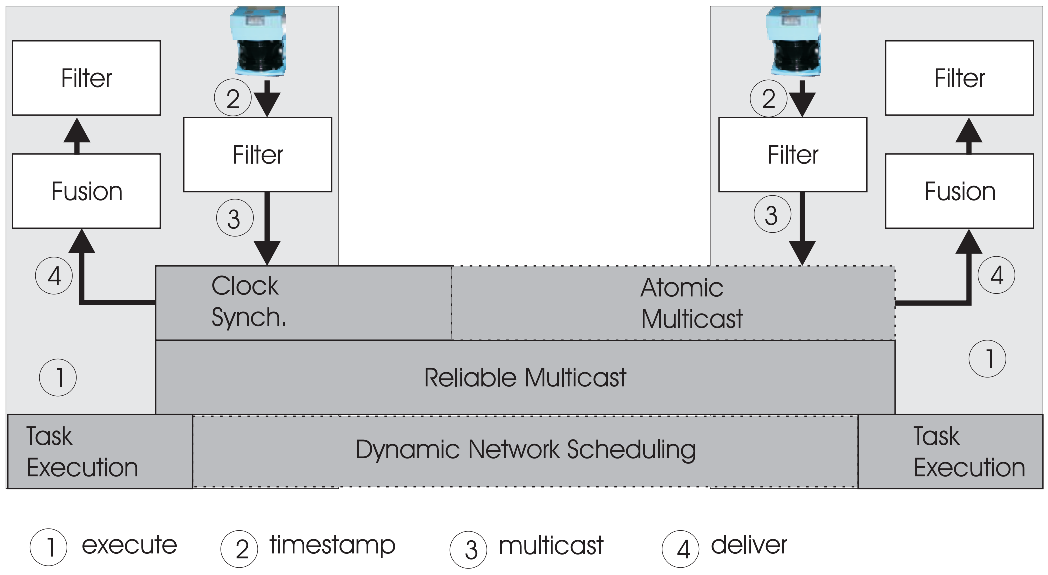


Figure 3-5. Architecture of the sensor fusion (system layer omitted)

The arrows in Figure 3-5 depict the way of the data through the application architecture. Each set of raw data provided by a scanner is assigned a timestamp from the global clock (Figure 3-5 (2)). After preprocessing the data, they are exchanged within the group, using the multicast communication services of the middleware (Figure 3-5 (3)). The timeliness of the communication services ensures that the data are delivered with a bounded delay (Figure 3-5 (4)). Each system fuses the data it receives and possibly feds them into some further filtering stages (called post-processing). Since there are no complex interactions between the systems in the application layer, the timeliness of the task execution service and the timeliness of the communication services allow achieving a predictable timing behavior for the distributed sensor fusion.

There are two possibilities to multicast the data within the group. The first, which is depicted in Figure 3-5, is using the atomic multicast service. This ensures that all mobile systems fuse the same data and allows providing to them a common external worldview. According to our approach to the coordination of mobile systems, the mobile systems can make local decisions based on such a common worldview and still achieve a coordinated behavior. However, task abortions in the fusion or post-processing stages still can lead to the mobile systems' having different worldviews. For example, if one the systems fuses all input data while another one only manages to process part of it, say, representing only the left half of the scene, they obviously come to different worldviews. This means that the more important common worldviews are for the application, the lower the probability of task abortions in fusion or the post-processing must be. Thus, if common views are required, it seems best to perform fusion on the highest layer of abstraction. For one thing, the mobile systems only have to perform the fusion and no post-processing after receiving the data. Moreover, fusion on the highest layer has a smaller execution time than on any layer below.

Another possibility is multicasting data with the reliable multicast service. This should be done if achieving common worldviews is not required. In this case, the overhead and the increased delay of the atomic multicast protocol can be avoided (the delays of both services are compared in 6.3). Thus, if short delays are more important then providing a common view to the group members another protocol stack can be configured to match this requirement.

The global time base provided by the clock synchronization protocol is required to achieve time coherence for the data to be fused. Two alternative approaches have been implemented in the prototype (Section 6.2): The first approach does not require synchronizing the observations of the systems. It uses a Kalman filter (Bar-Shalom and Fortmann 1988) to fuse scans observed at different points of time. The global clock is used to timestamp the data when they are observed. In the second approach, the clock synchronization is used to synchronize the times at which the systems observe the environment; that is, the laser scanners are triggered at common instants on the global time base. This ensures that the observed data are time coherent so that this approach can be applied when fusion takes place on lower levels of abstraction where Kalman filter cannot be used.

This sensor fusion scenario illustrates our idea of a configurable middleware. First, compared to the previous scenario, the Event Service was omitted. Whereas the Event Service provides a common view on the global state of the controlled system, observations of the external environment of the controlled systems have to be fused in the scenario at hand. Second, as was explained above, the atomic multicast service may or may not be used in the architecture, depending on whether or not a common view and coordinated behavior are requirements. Finally, dynamic network scheduling may be omitted also if fusion takes place in a static group. This shows how the middleware can be adapted to the scenario at hand with its different possible requirements.

The distributed sensor fusion scenario comprises application tasks with large and environment-dependent execution times that communicate using a reduced set of communication services. As the communication services of the middleware are considered in the context of the shared spatial resources scenario, the one at hand is mainly used in presenting and evaluating the task execution service. Besides the load characteristics, it is useful for that purpose because it was designed to include several kinds of application-inherent redundancy. In combination with the first scenario, it illustrates how the middleware can be adapted to fit the needs of different applications.

4 Communication in Cooperative Mobile Systems

Communication is essential for the cooperation of mobile embedded systems, particularly to achieve a coordinated behavior. In this chapter, we present the communication part of the middleware, which consists of a modular and configurable family of communication services. The architecture of the communication part, which extends over four layers, reflects the three main problems we address as well as our approaches to tackle them. The two bottom layers achieve a reliable and timely predictable communication over a wireless LAN characterized by a widely varying number of message losses. The two top layers provide common views to the application; while the lower of them achieves consensus about general aspects of the distributed control system, the Event Service on the highest layer turns to the more application-specific common views on aspects of the controlled system. The three bottom layers constitute the communication hardcore, which provides reliable and timely services with strong, application-independent consensus semantics and serves as the basis to realize services directly matching the semantics of the application on top of it. We call the family of protocols that implements the group communication services of the hardcore the Real-Time Group Communication Protocols (RGCP).

While Chapter 2 gave an architectural overview of the communication services, this chapter presents the communication protocols realizing these services. Our presentation comprises a description of the polling mechanism of the IEEE 802.11 Standard also, so as to describe the basic structure of this mechanism, which is common to both the original 802.11 Standard and the upcoming supplement 802.11e, and to specify those aspects which the standard leaves open, but which are important for our design. Apart from that, we focus our presentation on those protocols being developed as part of this thesis and refer to existing publications regarding the clock synchronization protocol (Mock et al. 2000b,Mock et al. 2000a).

The design of the protocols is based on a formal system model describing a synchronous system with omission failures. To match the characteristics of wireless media, this model does not make a priori assumption about the quality of the communication links; that is to say, about the number of omission failures they experience. Rather, it uses time-dependent link quality predicates to model the dynamically changing link quality. Based on such a model, the protocols are designed to accomplish a safe operation under the weak a priori assumptions, namely without assuming a fixed bound on the number of omission failures. Furthermore, they achieve progress for all those stations and times for which the link quality is sufficiently good. The atomic multicast and the membership service are fail-aware; that is, they indicate to their users whether or not they are able to guarantee progress.

This chapter is structured as follows. At first, we have to lay some foundations in sections 4.1 and 4.2. Section 4.1 introduces some preliminaries that help understanding the remainder of the chapter. It explains some basic concepts of layered architectures and system modeling, and gives a summary of the IEEE 802.11 Standard, which constitutes the basic layer of the communication part and is one of the starting points of the design. Section 4.2 presents the system model underlying the design of the protocols. Section 4.3 first gives an overview of the protocol stack as a whole, before it explains the protocols composing the stack. The description starts with the lowest layer where the polling mechanism resides and then moves up the stack to the highest layer where the Event Service is located. Finally, in Section 4.4, we discuss how related works addressed the problems this chapter deals with.

4.1 Preliminaries

The objective of this section is to provide the necessary background to understand the presentation of the system model and the protocols in the following sections. It is composed of two parts: First, in Sub-Section 4.1.1, we introduce some basic notions and terminologies from the fields of layered network architectures and system modeling. Subsequently, Sub-Section 4.1.2 gives an overview of the IEEE 802.11 Standard and its upcoming supplement 802.11e.

4.1.1 Concepts and Notions

4.1.1.1 Layered Architectures and Protocols

This clause introduces notions from the context of layered architectures, which will be used throughout this chapter. Rather than introducing our own notions, we make use of standard notions, which are in widespread use already, as far as possible.

Layering is a well-known and commonly accepted architectural principle for the design of communication protocols. According to this principle, networking hard- and software is structured as stacks of layers. Each layer provides services to the layer above it — say layer *n* for layer *n*+1. In doing so, it uses the services of its underlying layer; that is, layer *n* uses the layer *n*-1 services to provide its service to layer *n*+1. Thus, in a layered network architecture, complex communication services are realized step-by-step: Each layer starts off from what the lower layers provide and adds some new services with stronger semantics. Each service has a user and a provider. For a service of layer *n*, layer *n* assumes the role of the provider while layer *n*+1 assume the user role. Each service has an interface. The interface of a service is the set of operations the user invokes to access the service. In this thesis, we will specify interfaces by the signals exchanged between the user and provider of the service. Services are accessed at services access points (SAPs) where the provider and the user of a service transfer the corresponding signals.

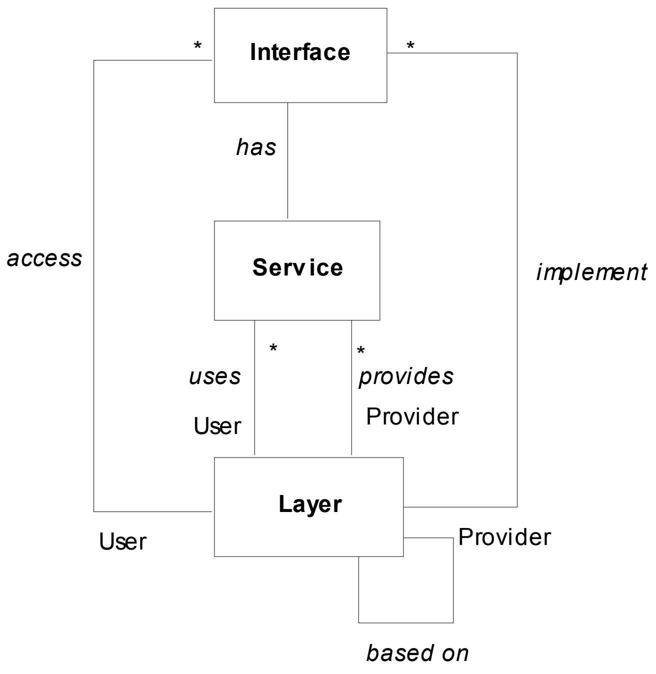


Figure 4-1. Conceptual model of a layered architecture

Figure 4-1 summarizes what we explained so far in a UML class diagram. Each layer, except of the lowest, physical layer, is based on another layer. It uses services from lower layers and provides services to higher layers. Each service has an interface. The provider of the service must implement the service's interface while the user accesses this interface.

After considering the layer as a conceptual thing, we now consider how a layer is realized in the physical system. There, a layer consists of a set of layer entities (or peers), one layer entity on each node of the network. To realize the services of the layer, the layer entities communicate among themselves using the communication services of the lower layer; Figure 4-2 represents these relations in a UML class diagram. The rules that govern the communication amongst the entities of a layer — in particular, the format and possible sequences of messages — are called the protocol of that layer. As the behavior of a layer entity is determined by the protocol it executes, layer entities are also called protocol entities. To denote entities of a certain protocol *X*, we will talk of *X* entities; for example, of a reliable multicast entity.

Each protocol entity is executed on a certain processor, which itself is part of a site or node in the network (cf. Figure 4-2). Frequently, no distinction is being made between protocol entities and the processors and nodes they are running on; rather, all these terms are used interchangeably. This is sensible and convenient, as long as a single protocol with exactly one protocol entity per node is considered. In this case, it is clear that saying "node *x* sends message *y*" means that the protocol entity of the considered protocol on node *x* sends message *y*. As we assume too that exactly one protocol entity resides on each node, we adopt this convention also: As long as it is clear from the context which protocol is considered we talk of nodes and the protocol entities running on them interchangeably. If, on the other hand, it is necessary to distinguish different protocol entities running on the same node, we will name them explicitly. Instead of the term "node", we will usually use the term "station", which is more commonly used in the context of wireless networks.

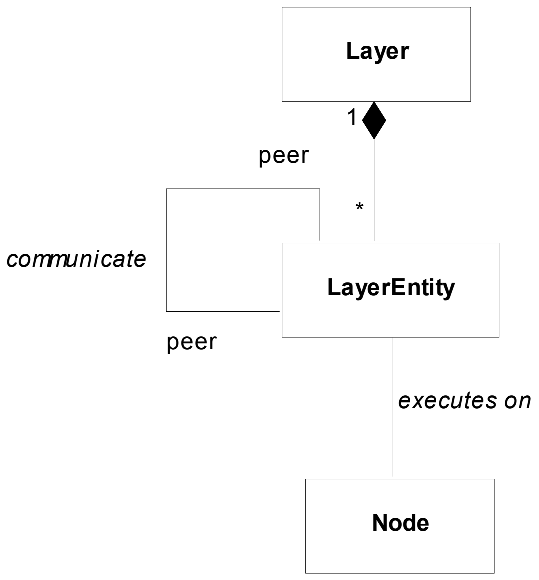


Figure 4-2. Physical realization of a layer

In this thesis we consider group communication rather than point-to-point protocols. As compared to point-to-point protocols, in group communication protocols, messages are addressed to a group of stations rather than to a single receiver. The set of participants of the protocol is called the membership of the group. A group typically refers to a set of stations that use the group communication services to cooperate in the fulfillment of some application functionality. Consider as an example a group of sensors that together provide a reliable perception of the environment or a group of vehicles cooperatively sharing a spatial resource. We distinguish static and dynamic groups. Static groups have a fixed membership, whereas the membership of dynamic groups may change at runtime. Changes in the membership are due to the following reasons:

- A further station joins the group and is added to the membership;
- A station explicitly leaves the group requesting to be excluded from the membership;
- A station is excluded from the membership because it is no longer operational (crashed) or no longer connected to the other group members, in which case the station implicitly leaves the group without requesting to be excluded.

In what follows, we will usually not distinguish whether a station explicitly or implicitly left a group.

The data units the protocol entities exchange are called Protocol Data Units (PDUs). The protocol entities use the services of the underlying layer to transfer the PDUs to their peers. Service Data Units (SDUs), on the other, are the data units the protocol entities receive from their users; that is, those data units they are expected to transfer on behalf of their users. As a layer *n* protocol uses the communication services of layer *n* – 1 to transfer its PDUs, what is a PDU from layer *n*'s perspective, becomes a SDU when handed down to layer *n* – 1 for transmission. While the structure and meaning of the PDUs is part of a protocol, the SDUs are considered as a kind of black box, a unit of data the structure and meaning of which are not known to the protocol.

4.1.1.2 System Models

The very basic prerequisite for the design of a communication protocol is to have a model of the system in which the protocol is expected to run. Relevant aspects are, for example, the available communication services and the topology of the underlying network. Since building system models is a key issue and heavily influences what services a protocol can provide, it has gained significant attention from researchers. In this section, we present some basic notions from the field of system modeling in order to lay the foundations for the presentation of our own system model (Section 4.2) and of related work on system modeling (Sub-Section 4.4.1).

Protocol design must be based on a model of the environment of the protocol entities. This means that it has to be considered which other components a protocol entity interacts with and that a, at best formal, specification of the services of these components must be provided. Regarding the specification of component services, an essential point is whether services have timing specifications or not. In the first case, the model is called *timed* whereas it is called *time-free* in the latter. In a time-free model, components are correct as long as they exhibit functional correct behavior, no matter at what times they provide their service. In the first part of this clause, we will consider which components have to be part of the system model.

If the protocol is required to provide reliable services, failures of the components it interacts with must be considered at design time. If this is done, the protocol can be designed to provide its service correctly in spite of failures of these components. A system model should therefore not assume that each component always adheres to its specification, but should take component failures into account. The model must specify in which ways components may deviate from their specification; that is, it must comprise failure model for the components. So, the second part of this clause will deal with failure models.

Elements of system models. The environment of a protocol entity consists of the following components (cf. Figure 4-3, the considered protocol entity is colored in gray):

1. The user. The user of a layer *n* entity is the layer *n+1* entity on the same node. This is represented by the link between the layer *n* entity *L(n)_e1* and the layer *n+1* entity *L(n+1)_e1* in Figure 4-3.

2. The communication sub-system. It offers communication services to the protocol and is accessed at the local layer *n-1* entity. This is represented by the link between

the entity *L(n)_e1* and the layer *n-1* entity *L(n-1)_e1* where *L(n)_e1* accesses these services.

3. The local node. The main service of the node is the execution service that runs the protocol entity. Additionally, it may provide special services, like clocks and permanent storage, which are relevant for the design of the protocol. In Figure 4-3, this is represented by the link from the entity *L(n)_e1* to the node *N1* it is running on.

4. The peers. The protocol entity interacts with its peers. It does not interface directly with them, as it does with the other components, but through the communication sub-system. In Figure 4-3, this is represented by the links from the entity *L(n)_e1* to its peer entity *L(n)_e2*.

System modeling deals with the points 2. to 4. For the communication sub-system and the node local services, formal service definitions and failure models must be provided. For the peers, the formal definition of their behavior is the main task of the protocol design and hence not part of the system model. The failure model of the peers, however, is part of the system model.

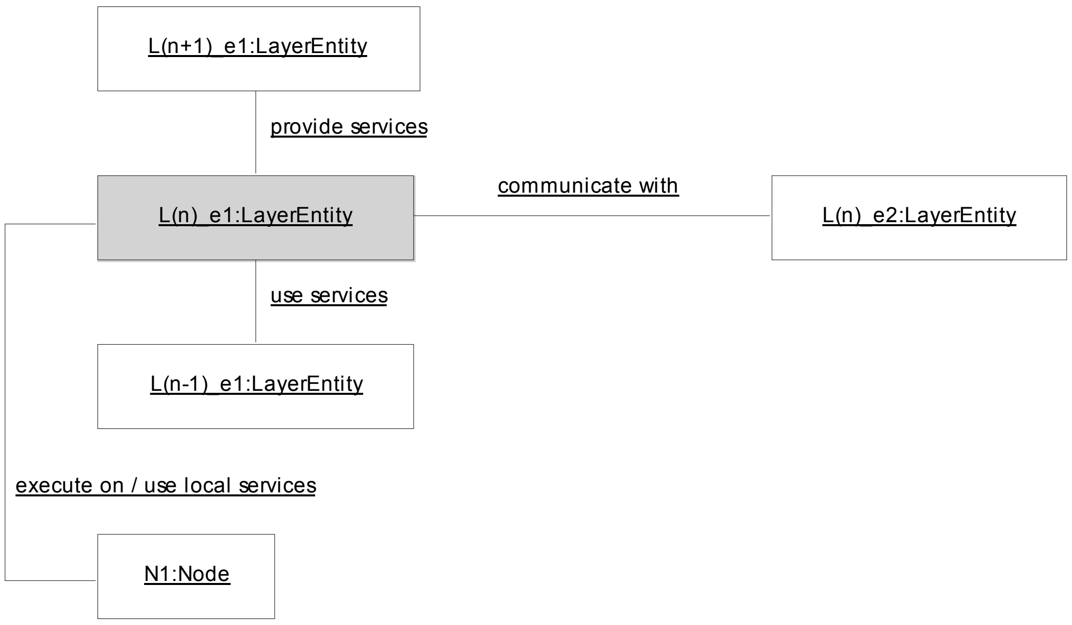


Figure 4-3. Environment of a layer *n* protocol entity

System models are usually divided into two parts: the *communication model*, which models the communication sub-system (point 2. above), and the *process model*, which includes the failure model of the peers and models the node local service (points 3. and 4. above). The interaction between the protocol entity and its user (point 1. above) is specified by the definition of the protocol services and their properties and is usually considered as a separated aspect not being part of the system model. Nevertheless, the system model and protocol service definitions are not at all independent concerns: Which services a protocol is able to provide heavily depend on the system model. For example, as (Fischer et al. 1985) pointed out in their seminal paper, it is not possible to design a protocol that achieves consensus in an asynchronous system with crash failures. Actually, it was this work that fueled a lot of efforts on system modeling.

The *process model* deals with the protocol entities and the node local services. Accordingly, the process model consists of two parts. The first, called the *processor model*, comprises the definition of the node-local services, their properties, and failure model. Such services include hardware clocks, persistent storage, or atomic send/receive actions. The second part is the failure model of the peers. It specifies how the actual behavior of the peer may deviate from the protocol specification. While it is the task of the protocol designer to provide a specification of the protocol, whether or not the protocol entities are assumed to adhere to this specification is part of the system model. A Failure of a protocol entity may have the following reasons:

1. A failure in the execution service that is executing the protocol entity. For example, it is possible that the node, and hence the execution service, crashes;

2. The implementation, which may deviate from the specification due to implementation errors;

3. Un-trusted, malevolent peers. Peers that are actually not implementations of the protocol specification may take part in the protocol and may intentionally misbehave in order to disturb the operation of the protocol.

In most cases, the first point is considered the main source of entity failures. In this case, the distinction between a processor model and a process model vanishes, because processes simply inherit the failures of the node's execution service.

The *communication model* describes the communication subsystem the protocol entities use to exchange their PDUs. It must contain two things: a specification of the available communication services and a description of the topology of the network as far as known at design time. The latter does not mean that a complete description of a fixed topology must be provided at design time. But, those invariants of the topology that influence the design of the protocol must be stated in the communication model.

Failure models. When building system models for highly reliable systems, incorrect behavior of system components should be taken into account from the very start of the design process. When a system or component deviates from its service specification, this event is called a system or component *failure* respectively. From the systems perspective, the failure of a component is a *fault* that causes an erroneous system state and may lead to a failure of the system (in-depth explanations of the terms failure, fault, and error can be found in (Nett 1991,Laprie 1992). If component failures are accounted for in the system model, the system can be designed in such a way that it provides its service according to the specification in spite of the component failures. This is referred to as fault tolerance and is crucial to accomplish a high reliability of the overall system. Although components may deviate from their specifications, it is usually not assumed that they may exhibit an arbitrary behavior. Rather, in addition to the service specifications, a failure model is provided for each component. The failure model describes in which ways the actual service of a component may deviate from its specified service. Thus, the behavior of a component falls into one of the following three classes:

- It is in accordance with the specification of the component's service. If this is the case, the component is called *correct.*

- It is not in accordance with the specification, but deviates in a way specified in the failure model. This is a *failure* from the components perspective and a *fault* that must be tolerated from the point of view of the overall system. Correct and faulty behavior are both described in the system model and together referred to as the *modeled behavior* of a component.

- Any kind of behavior that is not in the first two classes. Since this kind of behavior is not captured in the model and is not considered during system design, it may lead to critical failures of the overall. Therefore, it is of utmost importance to ensure that components always exhibit modeled behavior. The probability that this is the case is called *assumption coverage*.

The *failure model*, which is part of the system model, describes how the service of a faulty component may deviate from its specified service. Several kinds of such deviation, referred to as *failure modes* or *failure classes*, are distinguished and commonly used in failure models. To define these failure modes we adopt a model presented in (Powell 1992). The behavior of a component as observed by its user consists of the sequence of signals it sends. Each signal can be characterized by a pair consisting of the time at which the component sent the signal and a value, which comprises the type of signal as well as the values of signal parameters. These pairs are called "service items" in the cited model. What signals a component must sent at what times to fulfill its specification generally depends on the history of its inputs, i.e. the sequence of signals it received and the times of their reception. We can neglect this fact here, simply assuming that there is an omniscient observer that perceives the inputs and knows what signals the component must sent at what times; that is, the sequence of time-value pairs the component must deliver. Now, defining a failure mode amounts to defining how the actual sequence of time-value pairs may deviate from the specified one.

A single time-value pair can deviate from its specification in two ways: it may contain an incorrect value or an incorrect time, referred to as timing error and value error respectively. The general case of a value error is called *arbitrary value error*, which means that values may deviate from the specification in any possible way. There is another, more restricted kind of value error in that class, called *non-code value error.* For non-code value errors, an erroneous value always lays outside a given set of code values so that it is possible to detect the error and omit the affected signals. To realize this kind of error, redundancy techniques such as error detecting codes are applied. Timing errors are distinguished as follows.

Arbitrary timing error. The signal may be delivered at any time (including infinity).

Early timing error. The signal is delivered before the earliest point of time allowed by the specification.

Late timing error. The signal is delivered after the latest point of time allowed by the specification.

Omission error. The signal is never delivered.

In addition, we can consider that a component may deliver a time-value pair, which is not at all specified in the service of the component. This is called in *impromptu error.*

If a service has multiple users, the component sends replicas of the same signal to multiple users. Thus each service item consists of a set of time/value pairs. Consider for example a multicast service where a message must be delivered to a group of users. This extended model allows distinguishing errors w.r.t the perception by the group of service users. The key point is whether users perceive errors consistently. Consistency here means that all time/value pairs in a service item have the same value and approximately the same time; that is, all users receive the same signal at about the same time. It is important to note that a service item can be consistent, even though it is erroneous. For example, a consistent omission error corresponds to a service item in which all time/value pairs have an omission error.

Basing on the different kinds of errors in the service items presented above, we now define the failure modes that are relevant for this thesis. In his model, Powell introduces an additional intermediate step: At first, he defines assertions on the sequence of service items for both the time and the value domain. Then, he uses the Cartesian product of the assertions for the time and the value domain to define a space of possible failure modes. While this is good approach to define a very general model, we prefer focusing on those failure modes that will play a role in what follows. We are considering the following failure modes:

Crash failure. Each service item is either correct or the service item and all its successors have omission failures. This means the component delivers a correct service until a certain point of time and from then on does not deliver any service item at all.

Omission failures. Each service item is either correct or has an omission error.

Later timing failures / performance failures. Each service item is either correct or has a late timing error.

Omission failures with bounded omission degree. Each service item is either correct or has an omission error, in which case one of the following *k* service items is correct. *k* is called the omission degree.

Weak fail silence. Each service item is either correct or has on omission error, in which case one of the following *k* service items is correct, or all following service items have omission errors. This model is a combination of crash failures and omission failures with bounded omission degree: Up to a certain point a component observes omission failures with bounded omission degree and then it crashes.

Arbitrary failures. Each service item can have arbitrary timing and value failures.

Each of these failure modes can be considered for multi-user services also. For such services, it can further distinguished whether or not failures are perceived consistently.

In the literature there are more failures modes than presented here (for example, non-cooperative Byzantine failures (Masum 2000)), which are typically defined to represent some special behavior of a considered systems; yet, we restrain ourselves to those modes relevant for the thesis.

Summarizing, a system model provides service specifications and failures mode assumptions for the components the protocol entities interact with. A system model consists of two parts: the process model and the communication model. The process model deals with the node local services and provides a failure model for the peers, whereas the communication model deals with the communication services and specifies invariants of the network topology that are known at design time.

4.1.2 The IEEE 802.11 Standard

In this section, we provide a short description of the IEEE 802.11 Standard. We concentrate on those aspects relevant to the system model and the protocols presented in the remainder of this chapter. Hence, this section is not intended to be a detailed description of the IEEE 802.11 Standard with all its supplements; the interested reader is referred to (Rappaport 1996,Nett et al. 2001,Tanenbaum 2003) for more complete descriptions.

In 1997, the IEEE published the IEEE 802.11 Standard for wireless local area networks. This new member in IEEE's already well-established family of technologies for local area network essentially provides a kind of Ethernet for wireless networks and leverages interoperability for wireless LAN solutions from different vendors. It was a success right from the start, and it is today the commonly accepted technology for wireless communication. In fact, the availability of a vendor independent technology to replace or enhance existing Ethernet installations fuelled a trend to deploy wireless networks such that today wireless LANs can be considered one of the most rapidly growing segments of the IT market.

The IEEE 802.11 Standard distinguishes two different types of networks: ad-hoc and infrastructure networks. The basic building block of both is the *basic service set* (BSS), a set of stations in a common coverage area (indicated through the bounding lines enclosing the station sets in Figure 4-4) that coordinate the access to the common medium. The topology of a BSS may actually be somewhat more complex than the figure suggests. In particular, a station cannot necessarily communicate with all other stations in the same BSS. An ad-hoc network is a BSS considered as a self-contained network without connection to external stations and is referred to as an *independent BSS (IBSS)* in the Standard. Ad-hoc networks are usually set up without preplanning by simply having several stations in a common area communicate among themselves. In infrastructure networks, each BSS contains a special station, the so-called *access point* (AP), which provides distribution services to the stations of the BSS. The distribution services allow stations in one BSS to send frames to stations in another BSS. They are realized by connecting the AP to a common distribution system so that several BSS are connected to a larger network (cf. Figure 4-4). All stations in a BSS route their frames through the AP of the BSS. The AP decides whether the destination station of a frame is within the same BSS, in which case it forwards the frame within its own BSS, or the station is in another BSS, in which case it forwards the frame to the distribution system.

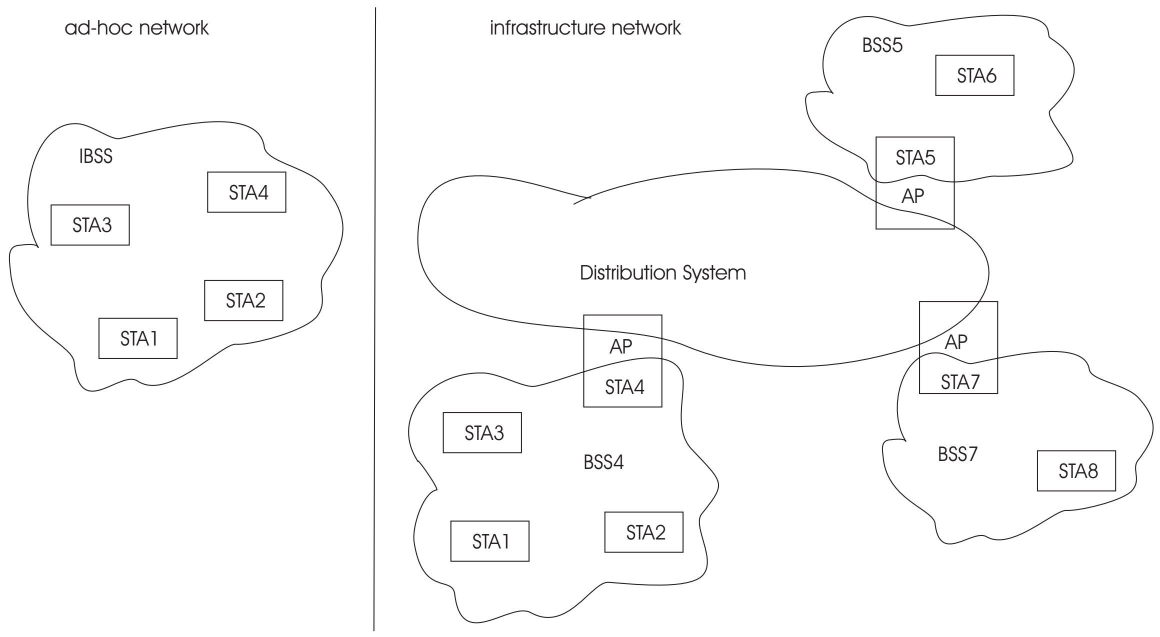


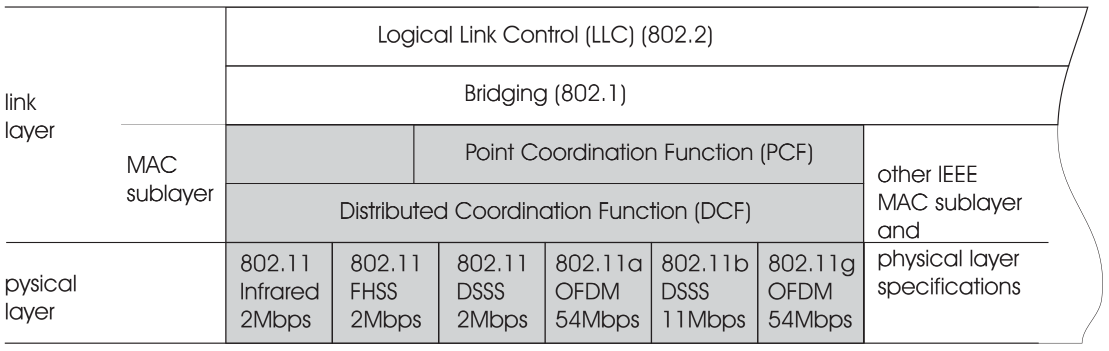
Figure 4-4 Ad-hoc and infrastructure networks (cf. (IEEE 1999))

In infrastructure networks, the source station and the transmitting station of a frame are not necessarily the same. As well, the destination station and the receiving station of a frame may be different. The source station is the station where the frame originates and where the user initiated the frame transmission. When the frame is routed through the distribution system, an AP will transmit it to its final destination. In this case, the AP is the transmitting station, but not the source station of the frame. Similarly, the destination station is the final intended recipient of the frame. Using the distribution system, the source station will first transmit the frame to an AP, which is the receiving station of the frame, but not the destination station.

An important concept in infrastructure networks is *association*. Each station that intends to send or receive messages must be associated with exactly one AP. This establishes a unique station to AP mapping, which is essential to forward frames through the distribution system. The original Standard specifies the protocol by which stations associate with APs and alter (re-associate) or terminate (disassociate) their associations, but it does not specify how the association information is propagated and used in the distribution system. Such aspects are considered in the supplement IEEE 802.11f (IEEE 2003).

The IEEE 802.11 Standard specifies the physical layer and the MAC sub-layer for wireless local area networks (sub-layers specified in the IEEE 802.11 Standard are colored in gray in Figure 4-5). Above the MAC layer, still in the data link layer, reside the bridging and logical link control (LLC) sub-layers that are common to all 802 MAC/PHY specifications and are specified in the IEEE standards 802.1 and 802.2 respectively. 802.11 itself specifies several physical layers that all operate under the same MAC sub-layer. These physical layers are characterized by different modulation/coding technologies, different frequency bands and different raw bandwidths. In the MAC sub-layer, the Standard specifies two access methods. The basic access method is the distributed coordination function (DCF), which is based on CSMA and collision avoidance. Above this basic and mandatory access method the standard specifies an optional method, the point coordination function. The

PCF is a centralized access method based on polling and intended to provide QoS in wireless networks.



Logical Link Control (LLC) (802.2)
Bridging (802.1)
link layer
MAC sublayer
Point Coordination Function (PCF)
Distributed Coordination Function (DCF)
other IEEE MAC sublayer and physical layer specifications
pysical layer
802.11 Infrared 2Mbps
802.11 FHSS 2Mbps
802.11 DSSS 2Mbps
802.11a OFDM 54Mbps
802.11b DSSS 11Mbps
802.11g OFDM 54Mbps

Figure 4-5 Structure and context of the IEEE standard 802.11 (gray)

In the remainder of this sub-section, we will first give a very brief overview of the different physical layers available in 802.11, just to present the choices available, the bandwidths they provide, and a rough idea of their technologies. (More detailed descriptions are presented in the works cited above.) We will then focus on the two the access methods specified in the standard, paying particular attention to the PCF and how it is supported by the underlying DCF. Afterwards, we explain how message losses are addressed in the standard and how stations associate with the AP. Finally, we point out what new features to support Quality of Service in wireless LAN can be expected from the upcoming supplement 802.11e.

4.1.2.1 The Physical Layer

This section gives a brief overview of the different physical layers specified in the 802.11 Standard and its supplements. As represented in Figure 4-5, six different physical layers have been specified by the time of this writing (in fact, 802.11h which is an extension of 802.11a may be considered a seventh physical layer). They will be shortly explained in turn in the following.

All physical layers, except one infrared physical layer, use radio communication and operate either in the 2.4GHz ISM band or in a 5GHz band. The 2.4 GHz is unlicensed and can be freely used for industrial, scientific, and medical (ISM) applications as long as they bound their sending power to 1W and use spread spectrum technologies to avoid interfering with other users of the band. In the 5GHz band, the regulation bodies allocated parts of the spectrum to be used for wireless LANs (HIPERLAN/2, for example, also uses these frequencies). In the following list gives a brief overview of the physical layers:

1. *Infrared*. An infrared physical layer operating at 1 or 2 Mbps. It was specified in the original IEEE Standard, but did not come into widespread use.

2. *Frequency Hopping Spread Spectrum.* This physical layer operates in the 2.4GHz ISM band and provides 1 or 2 Mbps of raw bandwidth. It sub-divides the band into 79 channels, each 1 MHz wide. The FHSS physical layer hops from channel to channel following a pseudo-random sequence. In each channel it spends a certain amount of time called the *dwell time*. With all stations using the same pseudo-

random sequence and the same dwell time, they will always be in the same channel at the same time as long as they keep synchronized. On the other hand, interference with other station operating in the same band can be limited to small intervals of time. FHSS physical layers provide good resistance against two common problems of wireless communication: multi-path fading and radio interference. As compared to DSSS physical layer, which will be described next, FHSS layers are less expensive to produce and have lower power consumption. Due to these features, it was commonly used for first IEEE 802.11 Standard compliant products, but it rapidly lost significance when higher bandwidth technologies based on DSSS came out because FHSS could not be scaled to offer higher bandwidths also.

3. *Direct Sequence Spread Spectrum* also operates in the 2.4GHz ISM band. To spread the signal over a wider spectrum, each bit to be transmitted is XORed with an 11-chip pseudo-noise (PN) code, a so-called *Barker Code*. Thus, the small-bandwidth data signal is spread into a signal of higher bandwidth. The receiver XORs the chips it receives with the same 11-chip pseudo-noise code to reconstruct the original signal. The ISM band is divided into 13 channels (11 in the US), each 22MHz wide. The center frequencies of the channels have a distance of 5MHz, such that adjacent channels overlap. To achieve interference-free coexistence, the center frequency of two channels must be at least 25MHz apart; so, only 3 LANs can be operated in a common area without interference. The original IEEE 802.11 Standard provided a DSSS physical layer capable of 1 or 2 Mbps of raw bandwidth. As stated above, at this data rate the FHSS technology was the more popular choice.

4. *Direct Sequence Spread Spectrum – High Rate (802.11b)*: In 1999, the IEEE published the supplement 802.11b, which specifies an enhancement to the DSSS technology that allows for data rates of up to 11Mbps. While the technology still operates with 11MChips/s, another coding is used to achieve higher data rates. In 802.11b physical layers, 8-Chip PN codes , called *Complementary Codes*, are being used to represent symbols so that 1.375 MSymbols/s can be transmitted. Each symbol represents 8 data bits. Together this yields a data rate of 8 Bit/Symbol × 1.375 MSymbols/s = 11Mbps. This technology rapidly came into widespread use due to its increased data rate and soon superseded the old FHSS technology.

5. *Orthogonal Frequency Division Multiplexing (OFDM) in the 5GHz band (802.11a).* As the name suggests OFDM uses several frequencies in parallel to transmit symbols. 802.11a uses 52 small-bandwidth sub-channels – 48 to transmit data and 4 for synchronization. To achieve the highest data rate of 54 Mbps, 64-QAM modulation, which uses phase and amplitude of a signal for modulation, is used in each sub-channel. This allows for 6-bit symbols on each channel and hence for $48 \times 6 = 288$ bits per OFDM symbol. To tolerate bit failures in the sub-channels a redundant coding is used for forward error correction. For a data rates of 54Mbps 3 data bits are coded into 4 bits transmitted in the OFDM symbols. Together with the symbol rate of 0.25 Msymbols/s this yields a data rate of ¾ × 288 bits/symbols × 0.25 Msymbols/s = 54 Mbps.

6. *Orthogonal Frequency Division Multiplexing in the 2.4 GHz band (802.11g).* In 2001, the IEEE standards committee approved another high data rate physical

layer, which provides 54Mbps in the 2.4GHz ISM band. It uses the same technology as 802.11a but in the narrower and more crowded 2.4GHz band.

The technology used for implementations and measurements in the context of this thesis is 802.11b DSSS in the 2.4GHz band and with a maximum data rate of 11Mbps. This was the prevailing, and for most of the time only available, high data rata physical layer for 802.11 compliant wireless LANs.

4.1.2.2 Coordination Functions

The IEEE Standard specifies two medium arbitration schemes, called coordination functions, which are applied during alternating periods. The basic and mandatory coordination function, called the *distributed coordination function (DCF)*, is based on carrier sense multiple access (CSMA) and tries to avoid, yet not completely eliminates, collisions on the medium. Hence, a period under control of the DCF is called *contention period (CP)*. The other, optional coordination function is intended to be used for real-time communication. This so-called *Point Coordination Function (DCF)* is based on polling and precludes collisions on the medium. A period under control of the PCF is called *contention free period* (CFP). In what follows, we describe both the coordination functions and how they alternate in more detail. We will pay special attention to the PCF and how it is supported by the DCF because we base protocols on the PCF.

4.1.2.2.1 Distributed Coordination Function

The basic access mechanism used in the DCF is *carrier sense multiple access with collision avoidance (CSMA/CA)*. This is a non-persistent CSMA scheme with additional measures to reduce the probability of collisions. Collisions are particularly expensive on the wireless medium since collisions are not detected during the transmission.

A station intending to transmit a frame first determines whether the medium is idle or busy (carrier sense). If the station determines the medium to be idle for a *DCF inter-frame space* (DIFS), it can start transmitting its frame. This is referred to as immediate access. If, on the other hand, it senses the medium to be busy, it invokes the *backoff procedure*. In the backoff procedure, a station randomly chooses a backoff time from the so-called *contention window*. After it has determined the medium to be idle for a DIFS, the station starts counting down its backoff time as long as the medium remains idle. In Figure 4-6, stations STA1 and STA3 invoke the access procedure while STA2 is transmitting; so, they wait for a DIFS after STA2 stops transmitting and then start counting down their backoffs. As soon as a station senses a carrier again, it stops decrementing the backoff and resumes only when the medium is idle again (like STA2 and STA3 in Figure 4-6 when STA1 starts transmitting). When the backoff time reaches zero the station commences the transmission (see STA1 in Figure 4-6). This non-persistent CSMA scheme reduces the probability of several stations accessing the medium at the same time when an ongoing transmission ceases. Additionally, the contention window, that is, the range from which backoff times are selected, is increased after unsuccessful transmissions (how the success of a transmission is determined is explained below). This reduces the probability of several stations choosing the same backoff time in high load situations where collisions already occurred.

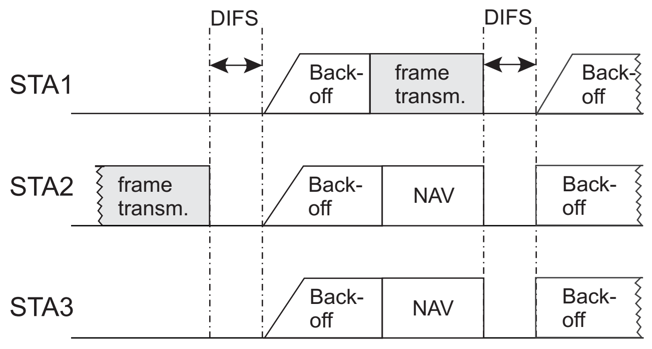


Figure 4-6. DCF Access procedure (acknowledgments omitted)

There are two additional measures to reduce the probability of collisions besides the random backoff mechanism described above. The first is the *virtual carrier*. The virtual carrier is realized through the so-called *Network Allocation Vector (NAV)*. A station marks times for which it knows the medium is reserved for other stations in its NAV. The NAV contains the number of time units during which the medium is assumed to be busy starting from the current instant. A station does not access the medium as long as the NAV indicates that it is busy, even if the station does not detect a physical carrier. The standard specifies several possibilities for a station to set its NAV. The first is the duration field in the frame headers. This field allows stations to set a virtual carrier for the length of the current frame transmission plus the time needed for frames that must follow the current frame according to the standard (such as an acknowledgement frame, as will be explained below). For example, in Figure 4-6, STA2 and STA3 set their NAV as soon as they receive the header of STA1's frame (Usually the NAV would also include the ACK, but this will be explained later and has been omitted in this figure.) An additional possibility to set the NAV is the RTS/CTS mechanism we describe in the following paragraph.

So-called *hidden stations* constitute a particular problem in wireless communication. Imagine a BSS with three stations A, B, and C, B situated between A and C. Assume that B is in the communication range of both, A and C, but that A and C are not in the range of one another (see Figure 4-7). Now, when A is transmitting a frame to B, C cannot sense the carrier. So, when C itself intends to transmit a frame, it will start transmitting during the ongoing transmission of A. At B, both these frame will collide, so B receives neither of the frames. The key problem is that a sender cannot determine whether the receiver is within the range of an ongoing transmission and it may start transmitting although there is a carrier at the receiver.

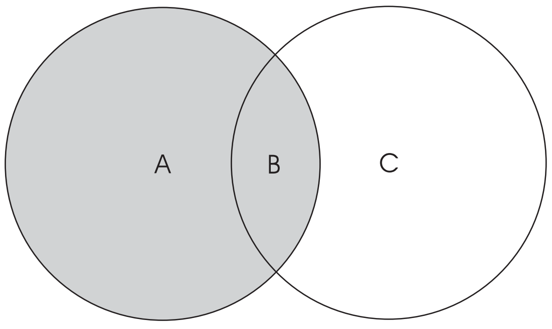


Figure 4-7. The hidden station problem

To alleviate the hidden station problem, the Standard provides the RTS/CTS mechanism. Before a station transmits a data frame, it transmits a short *request to send (RTS)* frame to the intended receiver of the data frame. The intended receiver replies to the RTS frame with a *clear to send (CTS)* frame. The duration field of both these frames is set to account for the transmission length of the following frames. A station that receives a RTS or CTS frame sets its NAV to the duration field of the frame. Thus, after a successful RTS/CTS pair, stations within the transmission range of the sender have set their NAV due to the RTS frame, while stations within transmission range of the receiver have set it due to the CTS frame. Therefore, no station being in the transmission range of either the sender or receiver will interfere with the transmission. For example, in the scenario described above, a CTS frame from B would have told C that a transmission is going on between A and B and C would have kept silent meanwhile. The RTS/CTS mechanism is not used for all data frame transmissions, but only if the data frame length exceeds a given threshold. Since collisions are still possible for RTS and CTS frames it does not totally avoid collisions, but only reduces their probability. Furthermore, the RTS/CTS mechanism is not applied for broadcast or multicast frames. So, for these frames, the hidden station problem still is an issue.

The DCF provides a completely distributed and relatively simple medium access protocol. It reduces probability of collisions and therefore exhibits a good throughput as long as the load (particularly the number of stations) is not too high. With increasing load, the number of collisions and the fraction of bandwidth wasted with unsuccessful transmissions will increase also. Thus, beyond a certain point, the achieved throughput decreases rather the increases above an increasing offered load. Due to collisions and the random waiting times of the backoff procedure, the DCF offers an asynchronous datagram service at best. It is therefore not well suited for time-critical communication. For such kind of traffic, the PCF, which we describe next, was provisioned in the standard.

4.1.2.2.2 Point Coordination Function

The IEEE 802.11 Standard specifies the PCF as an optional access method that was intended to accommodate real-time data traffic (such as voice) in the wireless LAN. In the PCF, a central entity, called the *Point Coordinator (PC)*, grants exclusive medium access to the stations by sending polling frames to them. The PCF is only available in infrastructure networks and the PC resides in the AP; so, we will talk of the AP instead of the PC in what follows. To allow the AP to gain control over the medium during the CFP (the time periods during which the PCF is executed), special features of the underlying DCF, such as shorter inter frame spaces and the NAV, are used as will be explained in the following.

Under the PCF, every station remains silent until it receives a polling frame from the AP. Upon reception of a polling frame (CF-Poll), it is allowed to send a single frame to an arbitrary destination station. In fact, a station is obliged to transmit a frame in reply to a polling frame; if it has no data or acknowledgment to transmit, it transmits a special Null frame. As the PCF operates in infrastructure mode, each data frame, no matter what its destination station is, is routed through the AP, which then forwards it towards its destination. Thus, the immediate receiver of each frame transmitted during the PCF is the AP. During the CFP, the AP is allowed to transmit data frames itself also in order to relay them to stations in the BSS. To save polling framing, IEEE 802.11 uses piggybacking. A data frame and a polling frame intended for the same station can be combined into a single Data+CF-Poll frame that carries the data and the poll.

There are two ways in which the AP maintains control over the wireless medium during the CFP: First, it uses inter frames spaces shorter than the DIFS; second, stations are forced to set their NAV such that the virtual carrier keeps them from sending unsolicited frames during the whole CFP. We consider the inter frames spaces first. When the AP transmits a polling frame to a station, it expects the station to start transmitting a frame in reply after a *short inter frame space (SIFS)*. In Figure 4-8, STA1 replies to a polling frame (CF-Poll) after a SIFS. As the name suggests, the SIFS is shorter than a DIFS. If the AP does not receive the frame after a SIFS and the medium remains idle, it starts transmitting the next frame a *PCF inter frame space (PIFS)* after the end of its last transmission. For example, in Figure 4-8, a PIFS after the AP transmitted a Data+CF-Poll frame to STA2, it polls STA1 because it did not detect the start of a transmission after the poll. The PIFS is longer than the SIFS, but shorter than the DIFS. With both the SIFS and the PIFS shorter than the DIFS, the AP is ensured to have a shorter medium access time than any other station in the BSS and hence a prioritized medium access.

The second mechanism used to avoid unsolicited frame transmissions during the CFP is the NAV. At the scheduled start of each CFP, all stations in the BSS set their NAV to the maximum duration of the CFP. Thus, the stations are forced to consider the medium busy and are kept from transmitting frames during the PCF unless they receive a polling frame from the AP. Stations reset their NAV when the AP announces the end of the CFP through a special control frame (CF-End).

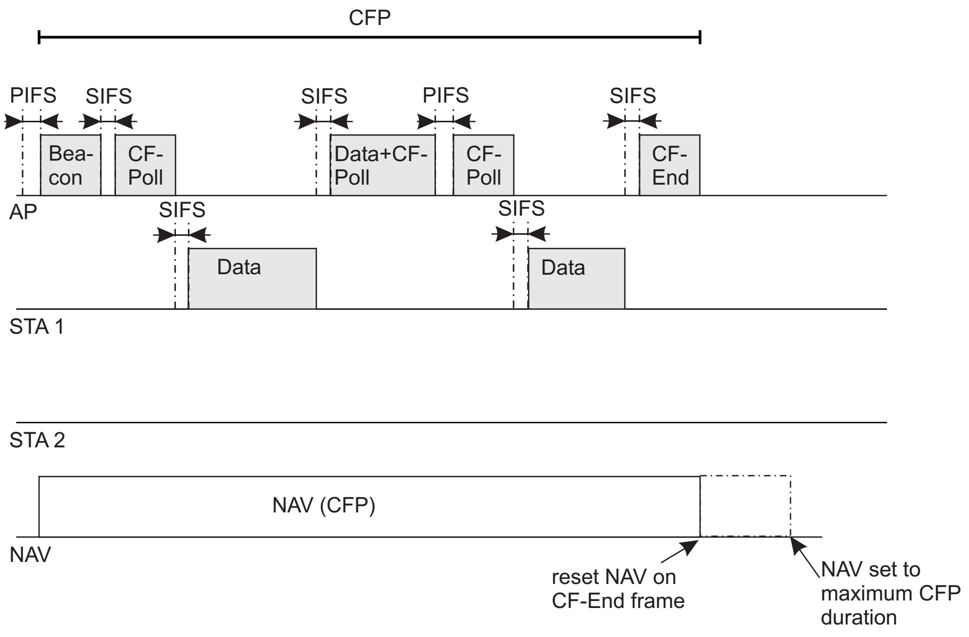


Figure 4-8. Timing of the PCF (acknowledgments omitted) (cf. Nett et al. 2001)

The Standard specifies that the AP shall use a *polling list* to control in which order stations are polled. Thus, the polling list represents the schedule of the medium. The standard, however, intentionally does not specify details regarding

1. How the polling list is established; that is, how the schedule for the medium is determined, and

2. How the AP allocates the medium based on the polling list, which, by our analogy, corresponds to dispatching in accordance with the schedule.

Thus, these things are left to the implementations of the Standard and different implementations may include different realizations of them.

4.1.2.2.3 Alternation of Coordination Periods

When both coordination functions are used in a BSS, they control the medium access in alternating periods. CFPs are scheduled to start at regular intervals, called the *contention-free repetition interval.* Between any two CFPs, there is a CP that allows for at least a single frame transmission. The alternation of CFP and CP is under the control of the AP.

Every CFP starts with the AP transmitting a beacon frame. The AP periodically transmits beacons frames to propagate operational parameters of the BSS. Not every beacon transmission starts a CFP; rather, the contention-free repetition interval is a multiple of the beacon period so that each n^{th} beacon starts a CFP (cf. Figure 4-9). The scheduled transmission time of this beacon is called the *target beacon transmission time (TBTT)* and coincides with the scheduled start of the CFP. As explained above, all stations set their NAV at the target beacon transmission to avoid unsolicited frame transmissions during the CFP. Stations learn the parameters they require to determine the TBTT when they receive a beacon frame or when they associate with the AP (see 4.1.2.4)

Although the stations in the BSS know the target beacon transmission time and set their NAVs, the start of the PCFs may be delayed. It is possible that a frame transmission that started during the CP is still in progress at the target beacon transmission time and extends into the CFP. In such a case, the AP cannot transmit the beacon starting the CFP until the ongoing frame transmission ceases (cf. Figure 4-9 wherein an ongoing frame transmission is depicted in light gray.) In fact, the delay caused by a frame transmission that extends in to the CFP may be as large as the maximum frame duration.

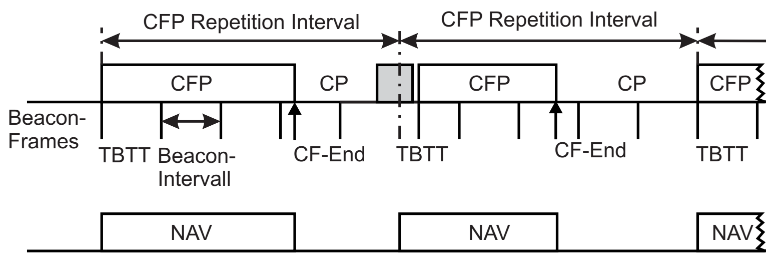


Figure 4-9. CFP/CP alternation (cf. Nett et al. 2001)

After the AP transmitted the beacon that denotes the start of the CFP, it exerts control of the medium and polls stations as described above. To end the CFP, the AP transmits a special frame (CF-End).

4.1.2.3 MAC-Layer Reliability Measures

Wireless media are particularly error prone. First, collisions are not detected during transmission, so every collision inevitably leads to a lost frame. Second, they have no shielding to prevent interference from distorting the signal. Therefore, the IEEE decided to add error control to the MAC sub-layer to increase reliability of MAC-SDU transmission as observed by the LLC sub-layer. This clause describes the error control mechanism of the IEEE 802.11 Standard.

The basic error control mechanism of the standard is *positive acknowledgment with re-transmission (PAR)* (also known as *automatic repeat request (ARQ)*). This is a dynamic time redundancy approach, in which the sender of a frame expects to receive an acknowledgement from the intended recipient; if it does not get one within a certain interval of time, it retransmits the frame. The IEEE Standard specifies that each station that receives a directed data or management frame shall transmit an ACK frame SIFS time units after the end of the frame transmission[2]. At the end of a frame transmission that requires acknowledgement, the transmitting station set an ACK timeout. If it does not receive an ACK frame before the timeout expires, the station retransmits the data or management frame. Before the station retransmits the frame, it executes the backoff procedure described above.

The number of re-transmissions of a single frame is bounded. There are two retry limits specified in the standard, the short and the long retry limit. The short retry limit applies to frames with a length not greater than a given threshold (the *RtsThreshold*), whereas the long retry limit applies to frames longer than that threshold. In can be considered one of the drawbacks of this standard that it does not allow specifying the retry limit on a per-message or per-stream basis. In fact, all 802.11 compliant network cards we had in use did not allow changing the value of the retry limits at all. Thus, messages or streams with different reliability and timeliness requirements are all handled with the same retry limits.

Re-transmitting messages to tolerate message losses can lead to duplication of messages. When the sending station transmits the same message in several frames and the receiving station receives more than one of these frames, it receives the same message several times. If the receiving station delivered the message each time it receives it, the message would be duplicated. To avoid delivery of duplicates, a receiving station must be able to detect that different frames carried the same message. For this purpose, the IEEE Standard uses sequence numbers. Stations attach sequence numbers to the data and management frames they send. Frames including the same message have the same sequence number, while frames carrying different message have different sequence number with high probability. Each station maintains a cache where it stores station address / sequence number pairs for the frames it received. For each transmitting station only the most recent pair is kept in the cache. Thus, when a new frame comes in, the receiving station is able to check whether it already received a frame with the same sequence number from the same sender. In this case, it does not deliver the included message; otherwise, it delivers it. Of course, the range of possible sequence numbers is bounded (a modulo 4096 counter is used) so that sending stations are forced to reuse sequence numbers for different messages. However, since the time to go through the whole range of sequence number once is significantly longer than the time between two transmissions of the same message, receiving stations are able to distinguish these two situations.

During the CFP, acknowledgments are used for point-to-point frames also, yet with two main differences: stations are not free as to when they retransmit frames, and acknowledgments are not necessarily transmitted in a dedicated ACK frame. Regarding the first

[2] A frame is called a “directed” frame if it has a single station address as its receiving station; this includes broadcast/multicast frames that are addressed to the distribution system and are hence transmitted to the AP at first.

point, a station cannot immediately initiate the re-transmission of a message when it detects a loss; rather, the station must wait until the AP polls it the next time. As for the second point, the Standard specifies several ways to piggyback acknowledgments so as to save the explicit ACK frames and increase performance. If the AP receives a frame after polling a station, it is allowed to piggyback the acknowledgement for that frame on the next CF-Poll, Data, or CF-Poll+Data frame it sends, no matter if this frame is addressed to the transmitter of the frame to be acknowledged. A station receiving a Data+CF-Poll frame from the AP is allowed to piggyback the acknowledgement on the Data frame it sends in reply if it has data to transmit. Thus, explicit ACK frames are usually not needed during the CFP. For example, going back to Figure 4-8, the AP would piggyback the acknowledgements for the two data frames on the Data+CF-Poll frame (sending a Data+CF-Ack+CF-Poll frame) and the CF-End frame (sending a CF-End+CF-Ack frame) respectively.

4.1.2.4 Association

Association is an important concept to support mobility of stations in infrastructure networks. Only after associating with an AP, stations are allowed to access the services that AP offers; in particular, the distribution service and the contention-free medium access. For the first service, the association establishes a unique station-to-AP mapping that is required for the distribution of frames over the distribution system. For the second service, station use the association to request admission to the polling list, that is, to allocate network resources. In this clause, we explain how stations find APs and associate with them.

The process of searching for an AP to associate with is called *scanning*. There are two modes of scanning: active and passive. In active scanning, a station transmits *probe request* frames on each channel where it searches for an AP. Probe requests may contain an individual BSS ID to search for a special BSS or a broadcast BSS ID to search for any BSS. The AP of a BSS is in charge of responding to the probe requests. In its probe reply the AP provides several important parameters to the scanning station; for example, parameters of the physical medium, supported data rates etc. One set of parameters provided in the probe response is the *CF Parameter Set*. It contains information about the time to the next CFP, the CFP repetition interval, the maximum CFP duration, and the remaining CFP duration. Thus, a station that decides to associate with the AP has all information required to adhere to PCF timing specification presented above. In passive scanning mode, a station merely listens on each channel where it is searching for an AP. As explained above, APs transmit beacon frames periodically. Like the probe response frames, beacons contain important parameters of the BSS. In both modes, the station stays on each channel to be searched for some specified time to wait for probe response or beacon frames. So, after going through all the channels, a station has learned which APs are located in its communication range and what are their key parameters. Using this information, it decides which AP it associates with.

A station that intends to associate with an AP must first authenticate. We will not explain here how authentication in 802.11 works. After authenticating, a station is allowed to associate with the AP. To this end, it sends an *association request* frame to the AP. In this frame, it announces whether it wants to be placed on the polling list. Four choices are possible: station does not support polling, station supports polling, but does not request to be polled, station supports polling and requests to be polled, station supports polling, but requests never to be polled. Additionally, the station announces the transmission rates it supports. The AP sends an *association response* in reply to the stations request. This frame contains a status code indicating whether the association was successful and, if it was not, why the AP did not admit the station. The AP may reject an association request for several reasons, for example, because it can handle no more stations, the station does not support all rates required in the BSS, the station requested polling capabilities that the AP does not provide, etc. When the association response indicates success this means that the station is now associated with the AP and that it was admitted to the polling list if it requested so.

Besides association, the Standard also allows for reassociation and disassocation of stations. A station may reassociate for two reasons: it changes from one AP to another one, or it wants to change the parameters of its associations. The latter is the case if a station already being associated with an AP wants to request to be added/removed from the polling list. Disassocation allows stations to abandon their association explicitly instead of simply stopping to communicate with the AP.

The concept of association provides the necessary information to the distribution system to accomplish its services. Furthermore, it allows stations to request admission to the polling list. It is worth noting that the protocol allows stations to give a very rough specification of their demands only — if they want to be polled or not. Actually, by the specifications in the standard, it is not even clear what — in terms of the bandwidth it gets — it does mean for a station to be part of the polling list, because neither the scheduling algorithm nor the dispatching based on the polling list is specified in detail. Thus, before higher layer protocols build on the PCF to provide timely services, a more detailed specification of the later aspects must be given.

4.1.2.5 802.11e

As this work is dealing with QoS on wireless 802.11 compliant media, the up-coming supplement 802.11e should not remain unmentioned, even though the IEEE has not yet approved it. This supplement is intended to extend 802.11's support for QoS. Avoiding details, which still may be subject to change, we will give a coarse overview about what is new in 802.11e and discuss how these new features relate to work presented herein.

Even though there is an increasing interest in industry to provide QoS over wireless LANs and although the PCF, which was intended for that field, was part of the original 802.11 Standard, it never became reality. The reason why the PCF was never implemented is not that polling per se was considered a bad technology, but that some of the details specified in 802.11 led to problems. Amongst these are:

1. The possibility that the start of the CFP is significantly delayed due to ongoing transmissions at the target beacon transmission time.

2. The fact that a polled station may transmit a frame of arbitrary length

3. The bandwidth a station gets after being admitted to the polling is not exactly specified.

4. The timing structure with one CFP and one CP during a CFP repetition interval is not very flexible.

5. There is no possibility to specify more detailed timing requirements than just requesting admission to the polling list.

So, a task group was set up to come up with an extension to resolve these points.

802.11e will support soft-real-time traffic during the DCF by providing prioritized access for several traffic classes. This enhanced version of the DCF is called *Enhanced Distributed Channel Access (EDCA)*. Under the EDCA, each of this traffic classes has its own set of DCF parameters including the inter frame space, minimum contention window, and maximum contention window. Obviously, these parameters can be chosen such that higher priority traffic classes have probabilistically shorter access delays. Furthermore, stations do not content for transmission of a single message, but for *transmission opportunities.* A transmission opportunity is an interval of time during which a station is allowed to transmit frames on the medium. This has two advantages: Stations may transmit several frames once they seized the medium, which reduces arbitration overhead. Furthermore, the maximum duration of a frame transmission can be bounded, which allows for a more predictable timing behavior.

In the IEEE 802.11e Standard, hard real-time traffic is still supported through polling. The timing of the coordination function — which is now called *HCF Controlled Channel Access (HCCA)*, where *HCF* stands for *hybrid coordination function* — has become more flexible. The AP is now free to poll stations at any time, not just during the periodical CFPs. Like in the PCF, the AP has a shorter access delay (still a PIFS) than the stations so that polling frames have a higher priority than data frames sent by stations. A station that receives a polling frame is allocated a transmission opportunity; that is, it has the right to transmit frames for a time specified in the polling frame. As explained above, transmission opportunities allow for a tighter control of the medium and improved performance as compared to PCF scheme wherein a station was allowed to transmit a single frame after being polled.

The enhancements specified in the upcoming 802.11e standard are not a replacement for the protocols that will be presented below. On the contrary, they can be considered complementary, in a sense that the protocol, with some adaptation, will work even better on an 802.11e MAC. In particular, the following benefits can be expected:

- The increased flexibility of the HCCA allows for a better integration of polling-based hard real-time traffic (as considered in our protocols) with soft and non-real-time traffic.

- The priority-based medium access under the EDCA will open up possibilities for an even more flexible scheduling of the resources. For example, it will allow transmitting requests for polling-based resource allocation at a higher priority under the EDCA. Again, this opens up opportunities to enhance the flexibility of the dynamic network scheduling layer.

In general, we intent to adapt our protocol stack to exploit these upcoming possibilities so as to achieve better integration of different kinds of traffic on the WLAN, a better tailoring of the service provision to the requirements of the different traffic classes, better bandwidth utilization, and more flexibility.

4.2 System Model

The starting point of a good protocol design is a model of the protocol's environment; that is, of the system wherein the protocol will run. In this section, we present the system model underlying the design of our protocols. As explained in Sub-Section 4.1.1, it has two major parts, the process and the communication model. Whereas the former, presented in Subsection 4.2.1, defines node local services and the failure model of processes, the latter, presented in Subsection 4.2.2, defines the topology of the underlying network, the communication services it offers, and the failure modes the links exhibit.

Since wireless media exhibit a poor and time-varying reliability, the communication model presented in Subsection 4.2.2 makes no general assumptions about the number of omission failures. Rather, we use a time and station dependent predicate *valid* to model the quality of the links. Whenever the predicate is true for a given station and a given interval of time, the number of omission failures affecting the communication between the AP and that station is bounded. In such a model, protocol design cannot be based on assuming a fixed number of omission failures. Rather, a safe operation of the protocols must be achieved for any number of omissions. Furthermore, while the model does not allow guaranteeing reliable and timely communication for all stations at any time, we require the protocols to achieve a reliable and timely communication for all those stations and times for which the link quality is sufficiently good; that is, for which the valid predicate holds.

4.2.1 Process Model

We are considering a set $\mathcal{S} := \{s_1, s_2, s_3, ...\}$ of stations, which are the nodes in the wireless network. According to the IEEE Standard we distinguish two kinds of stations, the AP and the clients. We first present the node local services available to the protocol entities and then the failure modes of the stations.

4.2.1.1 Node local services

In our model, it is the task of the execution service to execute the protocol entities in response to the occurrence of signals. Signals correspond to clock timeouts or to the reception of a frames on the network. The signals internally exchanged between the protocols of the stack are not considered here, since the whole stack appears to the execution service as a single task or process (see Section 6.3). The execution service of the stations allows for a timely execution of the processes.

Property 4-1 (Timely Execution). *There is a known constant δ_{sched} such that for each signal s addressed to a process p and received by the execution service at time t, the corresponding execution of process p is completed by time t + δ_{sched}.*

The failure model of the execution service will be presented in the following clause.

Stations have hardware clocks that have a bounded drift rate w.r.t real time. These clocks allow protocols to get readings of the current time and to use timeouts.

The services of the clock will be defined through a system variable *now* as is used in the SDL specifications. now_t denotes the value of the variable at real-time *t*. For sake of simplicity we assume that now_t is a continuous function and abstract from the granularity of the clock. The following property establishes a relation between interval lengths measured by the local clock and the corresponding intervals in real time.

Property 4-2 (Bounded drift rate). *There is a known constant* $\rho \in [0,1[$ *such that for all times t and t', t' > t holds*

$$dr(t,t') := \left| \frac{now_{t'} - now_t}{t' - t} - 1 \right| \leq \rho ,$$

where dr(t,t') is called the drift rate of the clocks in the interval $[t,t']$ *and* ρ *is the maximum drift rate.*

Furthermore the clock provides a timeout service. We use the SDL notions for working with timeouts to model this service.

Property 4-3 (Timeouts). *For all timeouts to, all real times t, and all clock values tt: if a process calls set(to,tt) at t and tt >* now_t *and t' denotes the first time such that* $now_{t'} \geq tt$ *then either the timeout mechanism issues signal to at time t' or the process called cancel(to) during]t,t'[.*

Property 4-2 ensures the existence of the time *t'* in the above definition.

Regarding the failure model of the clock, we assume that it operates correctly as long as the station is operational. This means that the protocol entities will not observe any failures of the clock. Stations have access to stable storage; that is, a storage that survives a station crash.

4.2.1.2 Failure Model

In this clause we present the failure model for the protocol entities. We assume that failures of protocol entities are due to failures of the execution service of the stations or due to failures of the stations themselves. Implementation failures or malevolent peers are not considered in the failure model. As explained in Sub-Section 4.1.1 above, in this case, the difference between the processor and the process model vanishes. Therefore, we will define the failure model in terms of stations instead of processes.

We assume an asymmetric system model, in which the AP and the clients exhibit different failure modes. We assume that the AP is not subject to any kind of failure. The IEEE 802.11 Standard, which employs the AP as a central coordinator during the CFP, already implies the assumption of a stable AP but does not indicate how this is achieved. From our point of view, a virtual AP provides the coordination functionality. It is implemented with redundant hardware using well-known fault-tolerance concepts if high reliability is require. Stations are subject to crash failures and omission failures. The latter means that the execution service may omit the execution of a process although it received the corresponding signal. The execution service decides to omit the execution of a process if it cannot ensure to complete the execution in time. How the execution service makes this decision, will be explained in more detail in the following chapter, Sub-Sections 5.4.3 and 5.4.4, which address the scheduling of the communication tasks.

Thus, for the stations we assume a synchronous system model with omission failures. This means our model assumes a timely execution of the protocol entities. This is achieved by our scheduling approach, which avoids timing failures for aperiodic tasks.

4.2.2 Communication Model

In the communication model we will specify three things

- The communication topology
- The communication service
- The failure model of the service

Before this can be done it is essential to determine on what layer of the architecture the model is situated because different layers provide different services, exhibit different failure modes, and have different topologies.

4.2.2.1 Layer of the model

As we are going to present those aspects of the PCF that deal with maintaining and processing the polling list as part of the protocol stack, our model describes a communication service within the data link layer. The interface of this service is situated above the DCF (cf. Figure 4-10). Beneath this interface, framing, addressing, and detection of transmission errors are provided as services of the underlying sub-layers, while the polling functionality, recovery from transmission errors, and other services are assumed to be located above this interface. Also the inter frame timing is assumed to be handled by the underlying DCF layer, which allows us to focus our model of the PCF to those aspects not addressed in the standard.

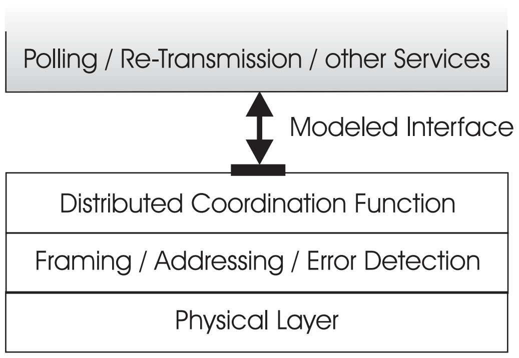

Polling / Re-Transmission / other Services
Modeled Interface
Distributed Coordination Function
Framing / Addressing / Error Detection
Physical Layer

Figure 4-10. Layer of the model

Actually, we are forced to build the physical implementation above a DCF MAC layer encapsulated in a network interface card (NIC) that does not allow using the special inter frame spaces for PCF frames. All frames transferred to the NIC for transmission, will be sent out with the complete DCF access procedure (including random backoffs) and frames with individual addresses will incur MAC level re-transmissions. This is because the MAC functions are time critical and are implemented in the firmware on the NICs. NIC vendors keep the source code for their firmware under disclosure, so enhancements cannot be added on that layer, nor can the interfaces be extended. However, this is not too much of a problem for the implementation. The timing of the protocols presented in the following only bases on the assumption that frames are not arbitrarily delayed before being transmitted on the MAC and that the probability of message losses does not grow arbitrary due to collisions. Both assumptions hold even if the PCF is implemented on top of the full DCF access procedure. The PCF still ensures that stations will not observe a busy medium when they are allowed to transmit a frame and that no collisions occur because of simultaneous access.

4.2.2.2 Topology

We model the wireless broadcast medium as a set of virtual point-to-point links that may have different frame loss rates and observe independent frame losses. This means that a broadcast frame send simultaneously over all the virtual links may be lost on some links and received on others. It does not mean a statistical independency of the frame losses. Assuming virtual point-to-point links is a more realistic model than assuming a homogenous broadcast medium where all stations observe the same link quality and frames losses are perceived consistently. In wireless communication the probability of frame losses depends on the relative spatial location of sender and receiver and on the kinds of objects in their surrounding. Hence it is a property that may be different for any pair of stations.

We are considering communication within a single BSS in which the AP controls medium access through polling. As explained above, in such a structure all communication takes place between the AP and the clients; clients do not communicate directly among themselves. Thus, only the virtual links between the AP and the clients are relevant in our model, which leads to a virtual star topology being imposed on the BSS (cf. Figure 4-11). To some extend, this virtual topology is less dynamic than the real, complete topology of the BSS; yet, links can still go down and have varying loss rate as will be discussed later in the section. The set of virtual links makes up the communication sub-system, which provides the communication service to the protocol entities.

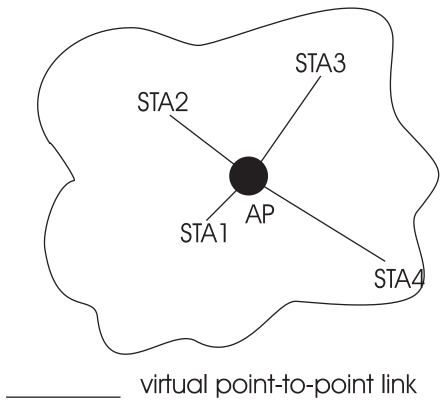


Figure 4-11. Virtual star topology of the network

4.2.2.3 The Communication Service

In the following we define the communication service through a sequence of properties (Schemmer and Nett 2003b). For these definitions, we assume that all frames sent can be distinguished. Note that this is only assumed for sake of the definitions, but that neither the service nor its user is required to provide such a unique distinction. We say that a station s_i sends frame *m* if the service user at s_i requests the communication service to transmit frame *m*, and we say that station s_i receives frame *m* if the communication service at s_i sends a receive indication for *m* to the service user. We denote by *rec*(*m*) the set of intended recipients of *m*.

Property 4-4 (Validity). *There exists a constant δ_{frame} such that for all stations s_i, s_j and all frames m, if s_i sends m at t and $s_j \in rec(m)$ and s_j is correct throughout $[t,t+ \delta_{frame}]$, then s_j receives m.*

The Validity defines the very basic property of the service; that is, the transmission of frames. It states that when a station sends a frame, all intended recipients will receive it.

Property 4-5 (Integrity). *For all stations s_i, all messages m, and all times t, if s_i receives m at t, there exists a time t' and a station s_j such that $t' < t$ and s_j sent m at t' and $s_i \in rec(m)$. Furthermore, for all stations s_i, all messages m, m', and time t, t', if s_i receives m and m' at t and t' respectively and $t \neq t'$, then $m \neq m'$.*

This property requires that the service does not modify frames, duplicate frames, or for any other reason indicates frames that were not sent. The following facts render this assumption plausible:

- The modeled sub-layer does not perform re-transmissions. As duplicate frames are usually caused by re-transmissions, this is not likely to happen here.
- Regarding modification of frames, the IEEE 802.11 Standard provides a CRC-32 (cyclic redundancy checksum) as a frame check sequence to ensure integrity. Assuming that within the modeled layer only non-code value failures occur, any modification can be detected and the service will not indicate the erroneous frames to its user. Thus, each modification that may occur during physical transmission is detected and results in no frame being delivered. If the integrity provided by the CRC-32 is not sufficient, further error detecting codes can be added, which however is not in the scope of this thesis.

Property 4-6 (FIFO). *For all stations s_i, s_j and messages m, m': If s_i sends m before m' and s_j receives m and m', then s_j receives m before m'.*

The FIFO property requires that frames from the same sender be delivered in the order they have been sent. On a single LAN, with only a single path from transmitter to receiver, distortion of the FIFO order only may occur if re-transmissions are used. As the modeled sub-layer does not perform re-transmissions, the FIFO order is always maintained.

Property 4-7 (Timeliness). *There exists a known constant δ_{frame} such that for all stations s_i, s_j, messages m, and times t, t': If s_i sends m at t and s_j receives m at t', $t' - t \leq \delta_{frame}$*

While Validity already requires that correct stations receive frames in time, timeliness ensures that no station, even a faulty one, receives a frame late. As explained above arbitrary delays are unlikely on the sub-layer on which our model resides for the following reasons

- There are no retransmissions, so only the delay of a single physical transmission is considered

- The medium access delay is bounded since under the PCF stations need not wait for the medium to become idle.

Obviously, timeliness is only ensured as long as frames are sent at a limited rate. In particular, if a second frame is sent while the transmission of the first frame is still in progress, a queuing delay at the local station will be the consequence. We therefore assume that the communication service sends a status indication to its user whenever the transmission of a frame ceases. So, the user is able to synchronize with the rate of the communication service. Figure 4-12 illustrates this idea: when the transmission of *m* ends, the communication service sends *stat_ind*. Reacting to this signal and sending the next frame takes at most δ_{sched} time units. The figure also makes clear that the indication is not expected to be sent at the same time when s_j receives frame *m*; for example, the propagation delay has to be considered, which we assume to be small however.

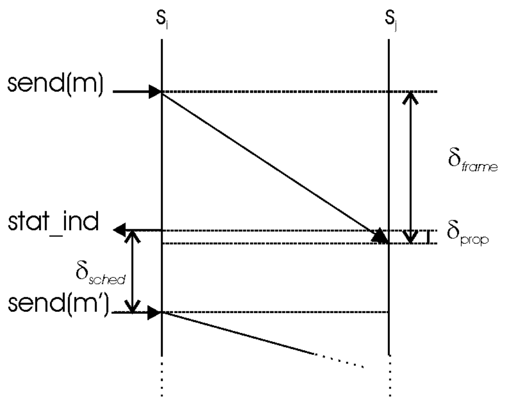


Figure 4-12. Timing of the frame transmission

The properties presented in this clause define the service provided by a correct communication link; in the following clause, we will define how a faulty link may deviate from this service.

4.2.2.4 Failure Model

For the communication service we assume omission/crash failure semantics. This means that if a station sends a frame, not all of its intended recipients may receive it. Thus, with omission failures, the validity property is not always fulfilled by the actual service of the link. We assume that omission failures may be perceived inconsistently; that is, if a frame has more than one intended recipient, some of them may receive the frame and some may not. This corresponds to the idea of modeling the broadcast medium as a set of virtual point-to-point links where a frame sent simultaneously over several links may be lost on some links and be received on the others. A crash failure of a link means after certain point of time on no frame will be received on that link. Link crashes may be due to the mobility of the stations; in particular, if a station moves thus far from the AP that no more frames will be received. Inconsistency of the failures raises another important issue, especially in cooperative applications. Inconsistent failures leave the intended recipients of the frame in inconsistent states, which may lead to inconsistent actions of the stations.

Summarizing we assume a synchronous system model with omission failures. This holds for both the process and the communication model. For both, a timely predictable behavior is assumed since the services have timing specification and do not exhibit timing failures. While our CPU scheduling achieves this for the process model, the basic medium properties under the PCF ensure it for the communication model. We do not assume a limit on the number of omission failures as part of the basic system model. Thus, the general model presented so far does not allow realizing a reliable and timely communication between the stations. In fact, in a wireless network with mobile stations, this cannot be ensured for all stations and at all times. For which stations such a service can be provided and for which not depends on the dynamically changing link properties, which will be considered in the following sub-section.

4.2.3 Dynamic Link Properties

The system model presented so far defines static properties of the communication sub-system, which are known at design time and are assumed to hold during the systems lifetime. These properties, however, are too weak to implement reliable and timely communication services above them. In this sub-section, we will explain how the model can be extended to overcome this problem.

The properties defined so far are safety properties; that is, they require that if a station receives frames these frames comply with certain requirements (they are timely, unmodified, ordered), but they do not require that stations receive any frames at all. The sole exception is the Validity property; but due to the omission failure assumption, Validity is not a static property of the communication sub-system. The key weakness is that there is no bound on the number of omission failures. For any given delay bound Δ, a protocol could run into the following situation: the protocol is trying to transmit a message (SDU) to its peer entities for Δ time units already, but every frame carrying the message was omitted. Now, the protocol can either go on retransmitting the message, in which case the message will be received late; or stop retransmitting the message, in which case it will be lost. So, to allow for reliable and timely transmission of messages we have to assume a bounded number of omission failures.

Wireless media exhibit a poor and time-varying reliability. Thus, we cannot generally assume an upper bound on the number of omission failures (so-called *omission degree*) even if such a bound is likely to exist in a certain environment around the AP. For one thing, a station may move thus far from the AP that the virtual link between the AP and that station crashes; that is, the station receives no frames at all after the link crash. Furthermore, the link may be in an intermediate state, where the station still receives some frames but may not adhere to an upper bound on the number of omission failures. The smaller the assumed omission degree, the more likely are such intermediate state. Figure 4-13 gives a spatial impression of what we explained so far. There is an environment around the AP where stations observe a bounded number of omissions only. Such stations are called *valid* stations (a formal definition will be presented below). Beyond this area, there is an area wherein stations, may still receive frames from AP, and vice versa, but without any upper bound on the number of omissions between the receptions of two frames. Such stations are called *partially valid* stations. Stations beyond this area, called *invalid* stations, receive no frames at all from the AP and the AP receives no frames from these stations. As a matter of fact, the boundaries between these areas are not as clear as the figure may suggest. Moreover, we should thing of them as being time varying since they depend on environmental settings, which are subject to change.

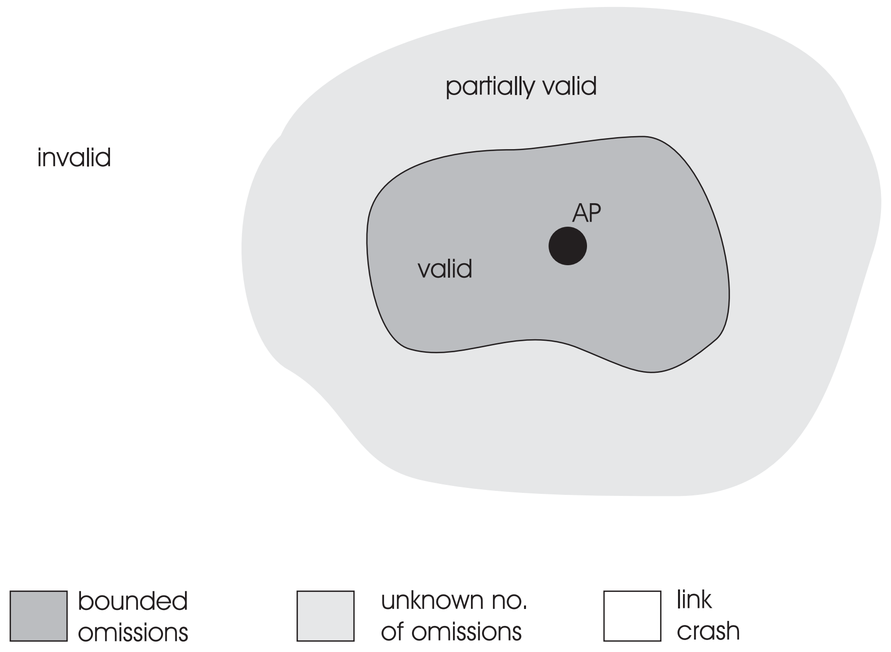


Figure 4-13. Omission failures in the AP's environment

So far, we noticed that reliable and timely communication services can only be provided as long as the number of omission failures is bounded and that we cannot assume that the number of omission failures is bounded in general. Nevertheless, as Figure 4-13 suggests, there are stations that can communicate with the AP with a bounded number of omissions failures. Now, the idea is to require the protocols to provide their services in time to these valid stations. Regarding stations that are not valid, it must be ensured that

- The presence of such stations does not prevent the protocols from providing a timely service to the valid stations, and

- The service provided to these stations fulfills certain safety requirements.

In this approach, the protocols have two kinds of properties: safety and progress properties. Safety properties require that if a service is provided it adheres to certain safety requirements; for example, if messages are delivered they are delivered in the correct order. The protocols ensure that safety properties hold for all stations at all times, no matter whether they are valid or not. Progress properties, on the other hand, require that a service is actually provided and within a given time bound; for example, that a message is delivered to its intended recipients in bounded time. The protocols ensure progress properties for valid stations only. Thus, progress properties are conditional properties.

Validity, invalidity, and partial validity are not static properties of the stations. Mobile stations are expected to move from area to area and hence to change from invalid to partially valid, from partially valid to valid, and so forth. Moreover, as explained above, the areas themselves may change dynamically such that even a station at a fixed location may change its state.

In the following we present a formal definition of validity and invalidity. We will define two special kinds of validity. One (Polling Validity) considers the poll-reply styled communication that allows clients to transmit messages to the AP. Remember that each client is expected to transmit a frame, called reply frame in the definitions, to the AP in response to each polling frame it receives. The other kind of validity considers the transmission of messages from the AP to the stations. We define the constant $\delta_m := \delta_{frame} + \delta_{sched}$, which denotes the maximum delay between the time a frame is sent and the time at which it has been processed at the receiving station.

Given a constant *OD* we define:

Definition 4-1 (Polling Valid). *For all stations s_i and all times t, t' with $t < t'$, s_i is polling valid during $[t,t']$ if and only if for each sequence of OD+1 consecutive polling frames the AP sends to s_i during $[t,t'-(\delta_m + \delta_{frame})]$ it receives at least one reply frame from s_i.*

Definition 4-2 (Receive Valid). *For all stations s_i, and all times t, t' with $t < t'$, s_i is receive valid during $[t,t']$ if and only if the following holds: If a protocol entity at the AP sends a message in at least OD+1 frames to s_i during $[t,t'- \delta_m]$, s_i processes at least one of these frames.*

Definition 4-3 (Valid). *For all stations s_i, and all times t, t' with $t < t'$, s_i is valid during $[t,t']$ if and only if it is polling valid and receive valid during $[t,t']$.*

Definition 4-4 (Invalid). *For all stations s_i and times t, t' with $t < t'$, s_i is invalid during $[t,t']$ if and only if each frame s_i sends during $[t,t'-\delta_m]$ is not processed at the AP and vice versa.*

The definition of "invalid" is not the negation "valid". There may be intervals, during which a station is neither valid nor invalid, in which case it is called partially valid. Furthermore, during intervals in which only few frames are transmitted, both predicates may be true. For example, if less than *OD*+1 polling frames are sent to s_i during an interval I, s_i could be valid or invalid. However, as long as we know that the protocols ensure certain properties if a station is valid during an interval of a specified length, it is not of so much interest that the station could also be viewed as being invalid during a fraction of the interval. In the remainder of the thesis we will usually omit the interval and only talk of valid or invalid stations where it is clear from the context which intervals are relevant. For example, if the transmission of some message is considered, the relevant interval is $[t,t'+\delta_m]$, being t the time of the first and t' the time of the last transmission of the message.

As will be pointed out in Section 4.4, the presented model is similar to timed asynchronous system model (Cristian and Schmuck 1995,Cristian 1996,Cristian and Fetzer 1999). In particular, the ideas of using predicates to describe the connectivity status of stations and to condition progress properties on these predicates are inspired by this work.

4.3 Description of the Protocols

In this section, we present the protocols that constitute the communication part of our middleware. We describe and define the main services of each protocol and explain how the protocol provides these services. A formal description of the protocols in SDL is provided in Appendix A.

At first, we give an overview of the protocol stack. We explain the interrelations of the protocols and how the service oriented architecture presented in Chapter 2 has been mapped to a stack of micro-protocols. Subsequently, in sub-sections 4.3.2 through 4.3.8 the protocols are presented bottom-up. Every protocol is presented in three steps: First, its services are defined; second, the operation of the protocol is explained for static groups; and third, we explain how the protocol is extended to accommodate dynamic groups also. Dynamic network scheduling, membership, and Event Service are not intended to be applied in static groups, so the operation of these protocols is presented in a single step. The protocol presented (Mock et al. 1999,Schemmer 2000) is a predecessor of the multicast protocols in the protocol stack. It was designed as a monolithic atomic multicast protocol for static groups and is based on a failure model with stronger, less dynamic assumptions on the link qualities than those underlying the protocol stack presented herein.

4.3.1 The Protocol Stack

To achieve modularity in the protocol stack, we designed each protocol to implement only a single communication feature; that is, it either implements some particular service for the application, like membership, or adds a special property, like atomicity, to an underlying service. This design makes it possible to realize complex services with strong semantics by combining several simple protocols. Such protocols are referred to as micro protocols (van Renesse et al. 1996), in contrast to macro protocols, which combine several services and properties, such as reliability and atomicity, in a single implementation. Micro protocols can be configured to form stacks that provide only those services and semantics the application actually requires. Besides accomplishing configurability, this kind of modular design brings about some interesting advantages: it fosters reuse of the micro protocols, reduces complexity of the protocols, and eases verification.

Figure 4-14 depicts the structure of the protocol stack including all micro protocols. Since AP and clients exhibit asymmetric behaviors, each protocol has two roles, a client role executed at the clients and an AP role executed at the AP. Therefore, the basic structure consists of two corresponding stacks connected by the wireless medium — one stack is composed of the client roles, the other of the AP roles. The figure reveals the user/provider relationships between the protocols: Each arrow between two protocols means that the upper protocol uses the services of the lower protocol. Which services the protocols provide and which services they use in doing so will be explained in detail in the following sub-sections, where we describe each of the protocols. We give here, however, some general remarks on the structure of the stacks.

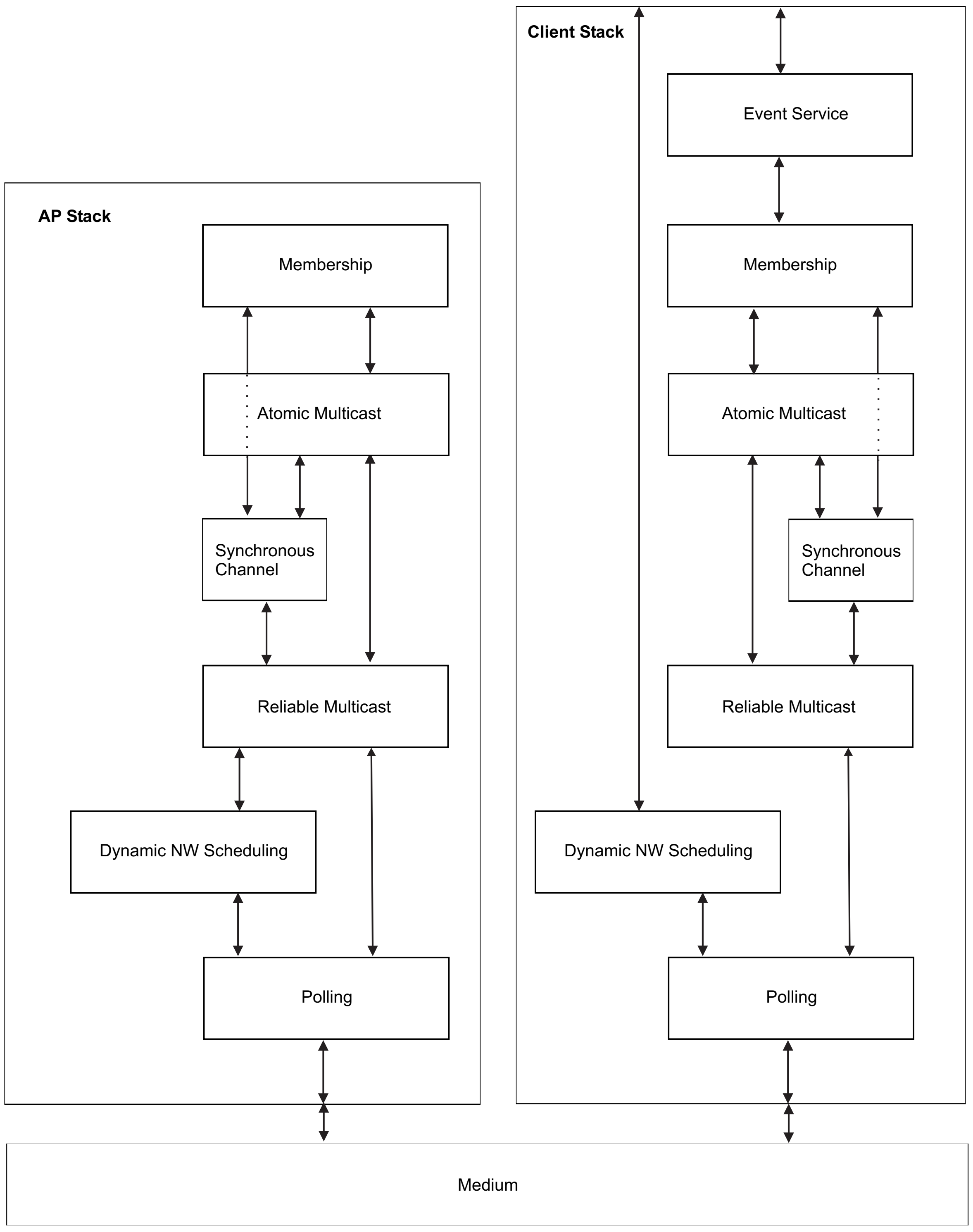


Figure 4-14. The protocol stack

In addition to their main communication services, micro protocols may provide stack internal services as well, which are intended for higher-layer micro protocols, rather than for the application. Such internal services are efficiently realized in conjunction with the main service of the protocol, so they can be considered a kind of by-product of the main service. For example, the poll-reply styled polling structure used to provide contention-free medium access can be exploited to detect stations that became invalid. The synchronous channel protocol, which conceptually belongs to the reliability layer of the middleware, provides only internal services for higher-layer protocols, namely for the atomic multicast and the membership protocol. It was therefore omitted in the architectural overview presented in Chapter 2. The synchronous channel protocol allows the reliable and timely transmission of a small amount of information from the AP to the clients.

Applications run on the clients only so that they access communication services only at the clients. Therefore, connections between the protocol stack and its environment are only depicted in the client stack; they represent the service access points (SAPs) where the applications access the communication services of the middleware. Actually, only protocols in the communication hardcore have protocols entities at the AP. Above the hardcore, the existence of an AP is transparently hidden behind the communication services of the hardcore. Therefore, the entities of the Event Service reside on the clients only. The client stack provides two SAPs to the application. At one of them, the application accesses the services of the dynamic network scheduling protocol to join the group if dynamic groups are supported. At the other, the application accesses the message transmission and membership services of the protocol stack, such as reliable or atomic multicast.

	1	2	3	4	5	6	7	8
Polling	X	X	X	X	X	X	X	X
Dyn. NW Scheduling		X		X		X	X	X
Reliable Multicast			X	X	X	X	X	X
SynchronousChannel					X	X	X	X
Atomic Multicast					X	X	X	X
Membership							X	X
Event Service								X

Figure 4-15. Configuration options of the protocol stack

Both the membership and the atomic multicast protocol provide application-independent common views and hence belong to the same layer of the middleware. Nevertheless, we decided to implement the membership protocol above the atomic multicast protocol in the protocol stack. This structure allows for a more efficient realization of agreement, total order, and virtual synchrony in the membership protocol since the agreement and total order properties of the atomic multicast protocol can be exploited.

Figure 4-15 shows possible configurations of the protocol stack. As can be appreciated from the table, it is possible to successively extent the stack by adding further micro-protocols on top of it. Usually each protocol is required by all protocols on the following layers. The dynamic network scheduling protocol, however, is only required if dynamic groups are to be supported. If static groups are considered, the reliable multicast protocol is based immediately on the polling protocol. So, using dynamic groups or not is a nearly orthogonal configuration option. The membership protocol, however, obviously only makes sense for dynamic groups. As a consequence, the Event Service too is only considered in conjunction with dynamic groups.

4.3.2 Polling

To provide predictable medium access, the original IEEE 802.11 Standard specifies a polling-based medium access method called the PCF. As well, the upcoming supplement 802.11e will support predictable access through polling (cf. Section 4.1). Thus, we use polling as the medium access method underlying our protocol stack. In this sub-section, we present the services of the polling protocol and how they are implemented. The protocol provides concrete implementations for those aspects that are intentionally left open in the IEEE 802.11 Standard; for example:

1. How the polling list is constructed from the requests of the clients;

2. How the AP decides which stations to poll and whether to piggyback data on a poll based on the polling list.

3. How higher layer protocols interface with the polling protocol to maintain the polling list.

On the other hand, low-level timing aspects, which are specified in the Standard, are not dealt with.

4.3.2.1 Service and Basic Operation of the Protocol

The basic service of the polling protocol is contention-free frame transmission. Messages (SDUs) the user passes to the polling protocol will be transmitted on the medium without collision. Contention-free access is provided according to the so-called the polling list, which represents the schedule of the medium. Contention-free access is provided to both the AP and the clients. Therefore, we actually distinguish two basic data transmission services: The client data service (*CL_DATA*), which allows clients to transmit data to the AP, and the AP data service (*AP_DATA*), which allows the AP to transmit data to the clients.

The polling list, which is the basic data structure of the protocol, determines the order in which the users are granted exclusive medium access. It consists of a sequence of entries. Each entry has an owner and one of two types: *poll* or *relay*. The owner of an entry is the station to which the medium is allocated. The type denotes whether the AP shall send a polling frame to the owner of the entry (type *poll*) or shall relay a frame on behalf of the owner. The polling entity at the AP processes the polling list in a cyclic, round-based manner. After processing all entries in the list, it starts again with the first entry and so forth. The interval of time during which frames are exchanged on behalf of the owner of an entry is called a slot. The sequence of slots corresponding to one run through the polling list is called a polling round or a round for short.

When the AP processes a polling list entry of type *poll*, its sends a polling frame to the owner of that entry. The standard requires that stations respond to polling frames, whether they have a message (SDU) to transmit or not. In the latter case, the station sends a so-called Null frame, which contains no message. Before sending a polling frame, the AP sets a timeout of length to_{Poll}. If either the polling frame or the station's response is lost, the timeout will expire and the AP will proceed to the next entry in the polling list. Otherwise, when the AP receives the client's response, it cancels the timeout and proceeds to the next entry in the polling list (cf. Figure 4-16 (a)). The length to_{Poll} of the poll timeout is set such that the timeout expires only after the latest possible reception time of the client's response. Thus,

$$to_{Poll} := (\delta_{frame} + \delta_{sched} + \delta_{frame})(1 + \rho) \,,$$

where the first factor accounts for the transmission and processing delays in real-time and the second factor accounts for the clock drift. Since the delays and the clock drift are assumed to be small, we will neglect the factor $(1 + \rho)$ in what follows for sake of simplicity.

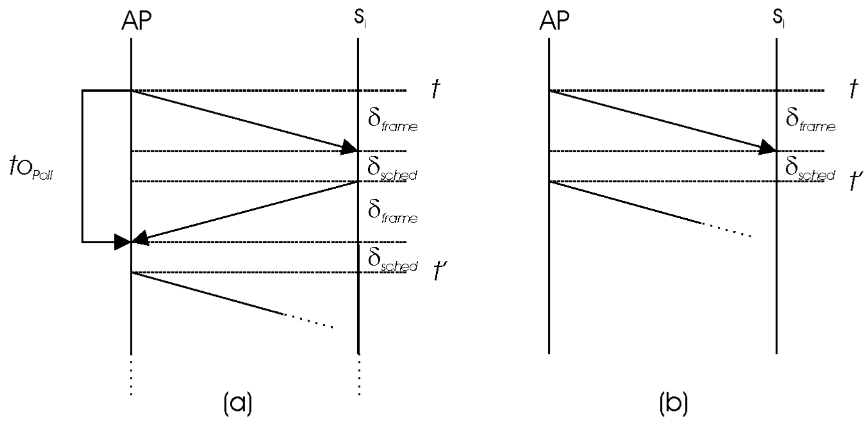


Figure 4-16. Length of a polling (a) and a request slot (b)

Figure 4-16 (a) shows that the length Δ_{Poll} of a slot corresponding to a polling entry is $\Delta_{Poll} := 2\delta_m$ ($\delta_m := \delta_{frame} + \delta_{sched}$, cf. Sub-Section 4.2.3). By time $t + \delta_{frame} + \delta_{sched} + \delta_{frame}$ the AP receives a frame from s_i or the poll timeout expires. So, by time t', δ_{sched} time units later, the AP processes the next polling list entry and sends a frame to the corresponding station.

According to the communication structure the IEEE Standard prescribes for infrastructure networks, clients send their data frames to the AP, which relays these frames to their destination station(s). For instance, when a client wants to send a multicast frame, it sends this frame to the AP as a point-to-point frame and the AP multicasts the frame to the stations in the BSS. The AP relays frames from a client when it processes a polling list entry of type *relay* owned by that client. Figure 4-16 (b) shows that the length of a slot corresponding to a *relay* entry is $\Delta_{Relay} := \delta_m$. At time t, the AP sends the relayed frame. By time $t + \delta_{frame}$, the frame transmission ceases and the communication service sends a status indication (cf. 4.2.2.3). δ_{sched} time units later, by time $t' = t + \delta_m$, the AP processes the next entry in the polling list and sends the corresponding frame.

In a network topology as considered in this thesis, routing messages through the AP tackles the reachability problem. As we do not make any assumptions about the link quality between any pair of clients of the same AP, it may be impossible for one client to transfer a data frame to another client directly. On the other hand, each valid client is by definition able to communicate with the AP with a bounded number of omission failures. Thus, it is possible for a client to transfer its data frame to the AP and for the AP to transfer this data frame to the destination station. Note that while this approach tackles the reachability problem it does not solve the problem of unreliable communication links; that is of message losses.

To increase the efficiency of the polling procedure, the IEEE 802.11 Standard specifies combined data and polling frames (Data+CF-Poll), which enable the AP to piggyback SDUs on the polling frames it sends. The polling protocol provides a service, called poll extension (*POLL_EXT*), which allows the user of the protocol at the AP to request the transmission of SDUs in the polling frames. Whenever the polling entity at the AP processes a *poll* entry, it solicits its user to request the transmission of a SDU. The polling entity transmits this SDU in the polling frame it sends to the owner of the entry.

4.3.2.2 Dynamic Group Extensions

When dynamic groups are considered the polling list is not longer static but may change when stations join or leave the group. Basically, it is the tasks of the dynamic network scheduling protocol to maintain the polling list in such cases. The polling protocol, however, supports it in doing so in the following three ways:

- The polling protocol provides interfaces for reading and setting the polling list so that the dynamic network scheduling protocol is able to modify the polling list when the membership changes.

- It provides medium access to stations that are not yet part of the polling list so that such stations are able to request to be added to the polling list.

- It detects clients that became invalid and sends a corresponding notification to the higher layers.

A distinguishing aspect in handling dynamic groups is whether the application requires valid stations to be able to join the group in bounded time. If this is the case, contention-free medium access must be provided to the joining stations. The shared spatial resources scenario is such an application. There, it is necessary to enable mobile systems arriving at a hot spot to communicate with the other approaching systems in bounded time.

Providing contention-free access to a joining station is a hard task (Schemmer and Nett 2003b). In order to request to be added to the polling list, a joining station must transmit frames to the AP; but, since the station has not yet an entry in the polling list, it has no contention-free bandwidth to transmit these frames. The AP cannot poll stations the addresses of which are unknown to it, and it cannot poll all stations that are potentially intending to be added to the polling list as well, as this would mean polling all addresses possibly in use. To solve the problem, we exploit specific properties of the shared spatial resources application. In that application, the protocol has to deal with a restricted and known number of tracks incident to a shared spatial resource. As a newly arriving mobile system is required to get its own bandwidth allocated in bounded, short time, we can assume that at each point in time, there is at most one system per track requesting to be added to the polling list. Each track has an identifier known to the mobile systems driving on it. A mobile system that wants to join the group provides the identifier of its track to the polling entity. Now, the basic idea is that the AP not only sends polling frames to stations in the polling list but also broadcasts for each track a special polling frame, a so-called join poll, containing the identifier of that track. Whenever a joining system receives a join poll containing the identifier its track it is allowed to transmit a frame to the AP. Thus, the protocol provides contention-free access to joining stations (so-called *JOIN* service). Station use the

JOIN service only as long as they are not part of the polling list; once they receive normal, directed polling frames, they stop reacting to join polls.

There are other mobile applications that have less stringent requirements regarding the dynamic allocation of communication resources. For such applications, the problem is amenable to a more general solution: Stations can request to be added to the polling list during the CP using the contention-based DCF. Actually, this is the approach adopted by the IEEE 802.11 Standard where stations associate with the AP during the contention periods and request to take part in the contention-free period while associating. The priority-based medium access procedure of the upcoming supplement 802.11e will open up new possibilities to realize medium access during the resource request phase. For example, it will be possible transmit resource requests at a higher priority under the EDCA.

When a client becomes invalid, the AP should free the resources allocated for that client to make them available for other clients. The polling protocol detects a client's becoming invalid and indicates this event to the dynamic network scheduling layer, which changes the polling list accordingly. To implement this service (called *FAIL*), the protocol exploits the communication structure. It considers the clients' replies to the polling frames as "i am alive" signs and counts the number of consecutive polling frames that were not followed by a reply from the client. If the counter assumes a value of $OD + 1$ the protocol sends an indication to its user. Thus, the protocol sends an indication whenever a client is invalid during an interval of length $\Delta_{Fail} := OD \times \Delta_{Round} + 2\delta_m$, where Δ_{Round} is the maximum length of one polling round. More formally, this service fulfils the following two properties:

Property 4-8 (Timeliness of Fail Notifications): *There exists a known constant Δ_{Fail} such that for all stations s_i and times t: if s_i is in the polling list at time t and is invalid during $[t,t+ \Delta_{Fail}]$, then the service sends a failure notification for s_i during $[t,t + \Delta_{Fail}]$.*

Property 4-9 (Justification of Fail Notifications): *For all stations s_i and all times t', if the service sends a failure notification for s_i at t', then there is a time t such that $t < t'$ and s_i is in the polling list throughout $[t,t'[$ and not valid during $[t,t'[$.*

4.3.3 Dynamic Network Scheduling

4.3.3.1 Service of the Protocol

It is the job of the dynamic network scheduling (DNS) protocol to maintain the polling list. As the polling list represents the schedule of the medium, this protocol represents the scheduler, while the polling protocol represents a kind of dispatcher. Currently, there are two kinds of events that trigger rescheduling of the medium: (i) an additional station wants to join the group and requests to be polled and (ii) a client that owns entries in the polling list becomes invalid. In response to such events, the DNS protocol reads the current polling list from the polling protocol, modifies it according to the event, and sends the new polling list back to the polling protocol.

More formally, we say that a station s_i starts joining when it requests the local DNS entity to be added to the polling list. The protocol fulfills the following three properties:

Property 4-10 (Timeliness of Reservations): *There exists a known constant Δ_{DNS}^{join} such that for all stations s_i and times t: if s_i starts joining at t and s_i is valid during $[t,t+\Delta_{DNS}^{join}]$, then s_i is added to the polling list by $t+\Delta_{DNS}^{join}$.*

Property 4-11 (Timeliness of Exclusions): *There exists a known constant Δ_{DNS}^{excl} such that for all stations s_i and times t: if s_i is in the polling list at time t and is invalid during $[t,t+\Delta_{DNS}^{excl}]$, then s_i is excluded from the polling list by $t+\Delta_{DNS}^{excl}$.*

Property 4-12 (Justification of Exclusions): *For all stations s_i and all times t', if s_i is excludes from the polling list at t', then there is a time t such that $t < t'$ and s_i is in the polling list throughout $[t,t'[$ and not valid during $[t,t'[$.*

4.3.3.2 Operation of the Protocol

Let us first consider clients that are part of the polling list and become invalid. To detect this kind of event the DNS protocol uses the *FAIL* service of the polling protocol. When the *FAIL* service indicates a client's being invalid, the DNS protocol removes the client from the polling list; that is, it removes each entry this client owns from the list. Thus, the timeliness and justification of exclusions (Property 4-11 and Property 4-12) are consequences of the timeliness and justification of the failure notifications (Property 4-8 and Property 4-9), and the time bound Δ_{DNS}^{excl} is equal to Δ_{Fail}. The DNS protocol sends a notification to its user whenever it removes clients from the polling list (so-called *EXCL* service). This allows higher-layer protocols to react to these changes. For example, protocols that expect acknowledgments from that client need no longer wait.

The medium reservation (*RESMED*) service of the DNS protocol allows clients to allocate bandwidth at the AP. When the service is invoked at a joining client, the DNS entity uses the *JOIN* service of the polling protocol to transmit request PDUs to the AP. Request PDUs contain the demand specifications of the joining client and, when received, allow the AP to perform a corresponding reservation. Currently, demand specification is only considered for clients that want to use the group communication protocols; other kinds of demand specifications can be added if required in the future. When the DNS entity at the AP receives a request PDU from a client, it first checks if the requesting client already owns entries in the polling list. If this is the case, the DNS entity simply ignores the request because in the group communication protocols each member is supposed to own exactly one pair of polling list entries (see Sub-Section 4.3.4). Other kinds of admission control can be added if required by the application. If the client is not yet part of the polling list, the protocol adds two entries to the polling list having the requesting client as their owner: First, a polling entry such that the client will be polled ones during each round and second a relay entry, so the AP can send a data frame on behalf of the client after polling it. Why this kind this particular kind of reservation is performed will be explained in the following subsection (4.3.4).

Let us now consider how long it takes a joining station to be added to the polling list. Assume a valid station calls the DNS protocol at time *t*. The DNS protocol invokes the *JOIN* services of the polling protocol, so after time *t* the station responds to the join polls it receives from the AP. The AP sends a join poll for that client by time $t + \Delta_{Round}$ and continuous to poll the client at times $t + 2\Delta_{Round}$, $t + 3\Delta_{Round}$, etc. Since the client is valid, the AP receives a reply for at least one of $OD + 1$ consecutive polls. Thus, in the worst-case, the AP receive a frame from the client by time $t + (OD + 1)\Delta_{Round} + 2\delta_m$. This frame contains the DNS PDU of the client and the DNS entity at the AP adds the client to the polling list. Therefore, Δ_{Res}^{join} can be computed as:

$$\Delta_{Res}^{join} := (OD + 1)\Delta_{Round} + 2\delta_m$$

4.3.4 Reliable Multicast

4.3.4.1 Service of The Protocol

The main service of this protocol is the reliable and timely transmission of multicast messages within the group. The service (*RELMC*) ensures that when a valid station sends a multicast each valid station in the group will deliver it (Validity). The service also ensures that reliable multicasts originating from the same station are delivered in the order in which the station sent them (FIFO), that no duplicates are delivered (Integrity), and that no message is delivered late (Timeliness).

For a more formal definition of the properties we again assume that all messages can be distinguished. If the user at station s_i calls the reliable multicast service to multicast a message m, we say that s_i multicasts m. Correspondingly, if the reliable multicast service at station s_i delivers message m to its user, we say that s_i delivers m. We denote by $\mathcal{M}(t)$ the membership view of the AP at time t; that is, the set of stations in the polling list at time t. A station s_i is said to be a valid member during $[t,t']$ if it is valid during $[t,t']$ and in $\mathcal{M}(t'')$ for each t'' in $[t,t']$.

Property 4-13 (Integrity). *For all stations s_i, all messages m, and all times t: If s_i delivers m at t, there exists a time $t' < t$ and a station s_j such that s_j multicast m at t'. Furthermore, for all stations s_i, all messages m, m', and all times t, t': If s_i delivers m and m' at t and t' and $t \neq t'$, then $m \neq m'$.*

Property 4-14 (FIFO). *For all stations s_i, s_j and all messages m, m': If s_i multicasts m before m' and s_j delivers m and m', then s_j delivers m before m'.*

Property 4-15 (Timeliness). *There exists a known constant Δ_{RelMC} such that for all stations s_i, s_j, all messages m, and all times t, t': If s_i multicasts m at t and s_j delivers m at t', then $t' - t \leq \Delta_{RelMC}$.*

Note that all three properties are very similar to the corresponding properties of the medium (Property 4-5 – Property 4-7). Basically, the reliable multicast protocol has to preserve these properties. This, however, is not trivially the case since retransmissions can lead to duplicates, distort the original FIFO order, and result in a message's being late. According to our model, we define the validity as a conditional property

Property 4-16 (Validity): *There exists a known constant Δ_{RelMC} such that for all stations s_i, s_j, all messages m, and all times t: if s_i multicasts m at t and both s_i and s_j are valid members during $[t,t+\Delta_{RelMC}]$, then s_j delivers m by $t+\Delta_{RelMC}$.*

4.3.4.2 Operation of the Protocol

It is the task of the reliable multicast protocol to deal with message losses; that is, to tolerate omission faults of the network. A typical approach to tolerate omission faults in networks is using time redundancy. For network protocols, timing redundancy means that the same message is transmitted several times to ensures that all its intended recipients receive it. The distinguishing aspect is whether the protocol performs a fixed number of retransmissions, which corresponds to a static redundancy approach, or whether it performs retransmissions only if message losses actually occur. The latter is a dynamic redundancy approach basing on error detection and recovery. Static redundancy approaches are not well suited for environments like ours where the number of message losses to be expected is by far less than the worst-case number of message losses. In such a setting, the number of message retransmissions would be far beyond the average number of omission faults and valuable bandwidth would be wasted. Hence, we are using a dynamic redundancy, where the number of retransmissions corresponds to the actual number of message losses. This is in line with our general approach to avoid using worst-case values, which are quite large and rarely ever reached in dynamic environments.

According to Sub-Section 4.3.2, each message is routed through the AP. For multicast messages this means, that the source station transmits a point-to-point data frame including the message it wants to multicast to the AP, which relays the multicast message to its destinations; that is, it sends a broadcast/multicast data frame including the message (see Figure 4-17). We refer to the point-to-point frames from the clients as *request frames* or simply *requests* and to the broadcast/multicast frames from the AP as *mc frame*s. Because of this structure, the DNS protocol (cf. Sub-Section 4.3.3) adds a pair of entries to the polling list for each joining station: the first, of type *poll,* allows the client to transmit the message to the AP after the AP polled it, and the second, of type *relay*, allows the AP to relay the message of that client to the group. We will refer to the corresponding pair of successive slots in each polling round as a RGCP slot or a slot for short. The duration Δ_{Slot} of an RGCP slot is

$$\Delta_{Slot} = \Delta_{Poll} + \Delta_{Relay} = 3 \times \delta_m$$

In the described two-staged transmission process omissions may occur during both stages — between client and AP and between AP and group. In what follows, we explain for both stages how omission faults are tolerated and a reliable transmission is achieved.

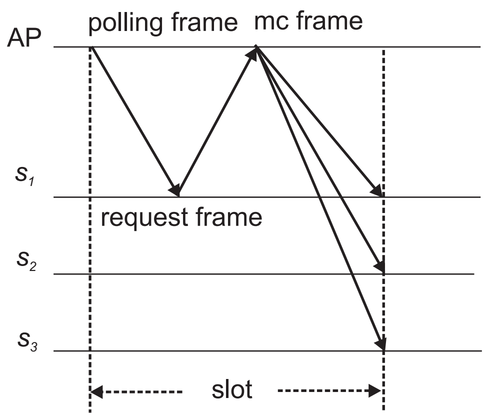


Figure 4-17. Structure of a RGCP slot

To tolerate message losses between the client and the AP, implicit positive acknowledgements are used. In positive acknowledgement approaches (also known as Automatic Repeat request, ARQ), the sending station expects a positive acknowledgment for its messages. If it does not receive an acknowledgement after a certain interval of time, it retransmits the unacknowledged message. In the reliable multicast protocol, the AP implicitly acknowledges the reception of a multicast message by sending a mc frame including that message. When the source station of the message receives this frame, it knows that the AP received the message. If the station does not receive this implicit acknowledgment, it retransmits the message when the AP polls it the next time. This protocol does not require explicit acknowledgement messages. Furthermore, the AP will receive each multicast messages of a valid station after at most *OD*+1 transmission attempts.

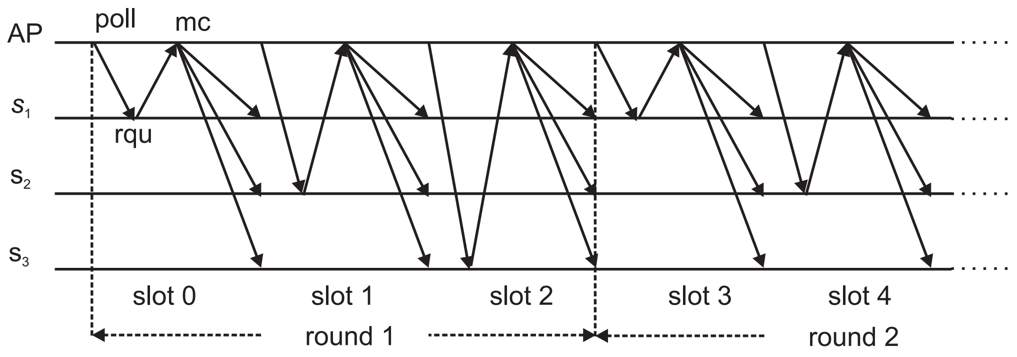


Figure 4-18. Round structure of the protocol

We use sequence numbers to avoid that retransmissions lead to duplicates of a message being delivered; that is, to ensure that the Integrity property (Property 4-13) holds. Each station assigns consecutive local sequence numbers to the messages it sends to the AP. Since the local sequence number together with the source address uniquely identifies the message, the AP can tell whether it already received the message or not. In the former case, the AP silently drops the message.

The second part of the protocol is in charge of detecting and handling message losses between the AP and the group. This part of the protocol also uses positive acknowledgments to detect message losses. When the AP relays a multicast message on behalf of a client it expects to receive acknowledgements for this message from all group members. As long as it has not received these acknowledgements, the AP retransmits the message during the next round when the relay entry of the originator of the message is processed the next time. As the AP expects acknowledgements from all clients, the design of the acknowledgment mechanism is a particularly critical point here. Avoiding explicit acknowledgment messages is the key design challenge. To tackle the problem, we exploit the round structure imposed by the polling protocol (see Figure 4-18). The communication is structured into a sequence of rounds. In the reliable multicast protocol, each group member has exactly one RGCP slot per round and each slot consists of a polling frame, a request frame, and a mc frame.

To exploit this round structure for efficient detection of lost mc frames, a bit field *ack* is provided in the requests. When a member receives a polling frame, it uses the *ack* field of its request to acknowledge each mc frame it received since the AP last polled it. To relate the bits of the *ack* field to mc frames, the AP assigns consecutive global sequence numbers to mc frames and polling frames. The polling and the mc frame of the same RGCP slot have the same global sequence number (corresponding to the slot numbers depicted in Figure 4-18). A station determines the position in the *ack* field relating to a mc frame it receives as the difference between the mc frames's global sequence number and sequence number of the last polling frame it received. When the AP receives a request, it knows that for each mc frame acknowledged therein the sender of the request has received the included multicast message. The AP knows that all valid members have received a multicast message and that it needs no longer transmit it, if either they all have acknowledged the message or if it has transmitted the message OD+1 times. Both parts of the protocol together ensure that every valid member will receive each multicast message send by a valid member. The presented acknowledgement scheme does not need extra frames for error detection because stations piggyback acknowledgments on the requests they sent anyway.

The reliable multicast service has a bounded worst-case delay Δ_{RelMC}. This essentially has two reasons: First, the number of omission faults between the AP and a valid client is bounded by the omission degree OD such that the number of transmission attempts necessary to transfer a multicast message from the source station to the AP and from the AP to the group is bounded. Secondly, the polling protocol ensures that the required number of frames can be transmitted within a bounded interval of time. Both these facts together with the structure of the protocol yield a worst-case delay of

$$\Delta_{RelMC} := 2 \times OD \times \Delta_{Round} + 2\delta_m \,.$$

According to the round structure described above, in a group with n members, the duration of a round is

$$\Delta_{Round} = n\Delta_{Slot} = 3n\delta_m \,.$$

Figure 4-19 illustrates this result. Assume a station s_i is polled to multicast a message at time t_1. If s_i is polling valid, at most OD consecutive polling/request pairs will fail so that the AP will receive the request frame of s_i by time $t_2 := t_1 + OD \times \Delta_{Round} + \delta_m$. After the reception of the request frame, the AP transmits the included message in a mc frame for the first time. OD rounds later, by time t_3, the AP transmits the message for the OD+1st time — if the message has not yet been acknowledged by all members. Each member, that was receive valid throughout [t_2,$t_3 + \delta_m$] received at least one of the transmission attempts and delivers the message by time $t_3 + \delta_m$.

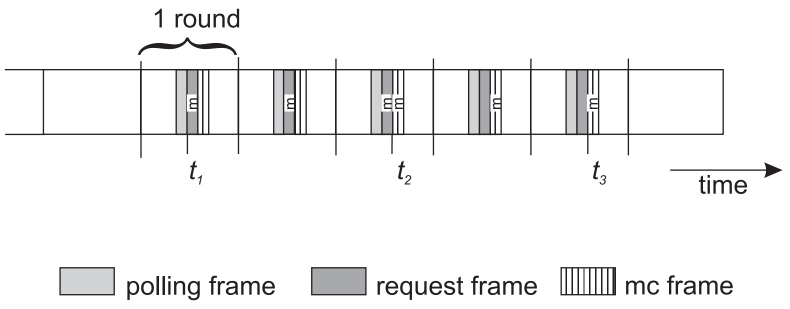


Figure 4-19. Delay of the reliable multicast protocol (OD = 2)

Although we are using a dynamic redundancy approach, the worst-case delay still depends on the worst-case number of message losses between valid members and the AP; that is, on the omission degree *OD*. This is because we must consider that in the worst-case a multicast message may need the worst-case number of transmission attempts in both parts of the protocol. As the omission degree may be quite large due to the unreliability of the medium, so may be the worst-case delay. But this is the price to be paid to ensure that each valid group member receives each multicast originating from a valid sender.

In many applications, messages have small deadlines, but no hard reliability requirements; that is, these applications can tolerate some message losses as long as the frequency is not too high. For such applications, it would be better to reduce the reliability offered by the service and provide a smaller worst-case delay instead. To allow for this kind of tradeoff, the protocol provides the parameter $res(m)$, $0 \le res(m) \le OD$, which allows limiting the number of retransmissions of a message *m* to a value smaller than *OD*. This means that clients do not transmit a multicast message *m* to the AP in more than $res(m) + 1$ rounds and that the AP does not transmit a message to the group in more than $res(m) + 1$ rounds. This results in the following new time bound, which depends on the resiliency of the multicast message instead of the omission degree. For all messages *m* with a resiliency $res(m) = r$ the time bound is

$$\Delta_{RelMC}(r) := 2 \times r \times \Delta_{Round} + 2\delta_m \,.$$

The price to be paid is a reduced reliability; that is, the conditional Validity property (Property 4-16) does not hold if a message is multicast with a resiliency smaller than *OD*; it still holds for messages with a resiliency of *OD*. Limiting the number of retransmissions to a value smaller than *OD* implies that the number of message losses affecting a multicast on its way from the client to the AP or from the AP to the group may exceed the number of retransmissions. Hence, it is possible that not all valid members receive it. Note that omission failures of the reliable multicast protocol may be inconsistent. That is, it is possible that some members deliver a multicast message and some do not. Even though there are applications that can tolerate lost multicast messages, inconsistent message losses may raise problems. This is particularly the case in our approach to achieve coordinated behavior, where coordination is achieved by providing common views to the application. The atomic multicast protocol that will be present in Sub-Section 4.3.6 ensures that agreement is achieved even in case of message losses: When not all members are able to deliver a multicast message, this protocol ensures that no member delivers the message so that the omission failure is perceived consistently. Thus, the application can trade reliability for improved timing guarantees while agreement is still guaranteed.

4.3.4.3 Dynamic Group Extensions

Supporting dynamic groups only requires minor changes of the protocol. Yet, changes in the membership have two implications:

1. The bound Δ_{Round} on the duration of a round can no longer be computed assuming n to represent the fixed number of group members. Instead, n represents an upper bound on the number of members.

2. Changes in the membership $\mathcal{M}(t)$ imply that the set of stations from which the AP expects acknowledgments changes also. Stations that are added to $\mathcal{M}(t)$ after the AP transmitted a multicast m for the first time are not guaranteed to deliver m. In fact, such stations may or may not deliver m. It is the job of the atomic multicast protocol to achieve agreement in this case.

4.3.5 Synchronous Channel

In this sub-section we describe the synchronous channel protocol. Rather than providing services directly to the application, it offers an internal service to the higher layers, namely the atomic multicast and the membership protocol. To these, it offers a small bandwidth simplex channel from the AP to the clients.

4.3.5.1 Service of the Protocol

The basic service of this protocol is a reliable and timely, small bandwidth simplex channel, called the synchronous channel, from the AP to the clients. The AP uses the synchronous channel to transmit a very small, yet essential amount of information reliably and timely to the group members. Whenever the AP sends a mc frame on behalf of a member, it can make a decision relating to that member and multicast it in the synchronous channel. The service guarantees that each valid member delivers the decisions in bounded time. It is important to note that the synchronous channel is not an additional physical communication channel, but is realized on the same medium that the other protocols use.

In addition to requiring that all valid stations deliver the decisions of the AP, we want the service to be fail-aware (Fetzer and Cristian 1996). A fail-aware service indicates to its user whether it is functioning correctly or not. This means that if a station is not valid and hence cannot deliver the decisions of the AP, the service indicates this fact to its user. To this end, the service exhibits two states to its users. Either the service indicates that it is functioning correctly, which we denote as *joined* in the context of group communication, or it indicates that it is not functioning correctly, which we denote as *stopped*. As long as the service indicates it is functioning correctly (state *joined*), it must in fact deliver the decisions of the AP in bounded time (Validity, Property 4-17 below). As long as the service is stopped, it delivers no decisions at all. We say that a station is joined if it is not crashed and the service at that station is in state *joined*. When the service changes from state *joined* to state *stopped*, it sends an indication *FAIL_ind* to its user. Now, since a stations is only required to deliver the AP's decisions while being joined, we define two properties to ensure that a station is joined whenever possible. The first property (Bounded Join Delay, Property 4-18) requires a valid station to become joined in bounded time after it started joining. The second property (Justification of Fail Notifications, Property 4-19) ensures that a station does not arbitrarily change its service state to stopped; that is to say, it only leaves state joined if there has been an interval of time since it became joined during which it has not been valid.

For the formal definition of the properties, we assume that all decisions the AP multicasts can be distinguished. There are known constants $\Delta_{SynchCh}$ and ε such that the following properties hold.

Property 4-17 (Validity): *For all stations s_i, all decisions d, and all times t, if the AP multicasts d at t and s_i is joined throughout* $[t, t + \Delta_{SynchCh} + \varepsilon]$, *then s_i delivers d by* $t + \Delta_{SynchCh}$.

Property 4-18 (Bounded Join Delay): *For all stations s_i and all times t, if s_i starts joining at t and s_i is valid throughout* $[t, t + \Delta_{DNS}^{join} + \Delta_{SynchCh}]$, *then s_i is joined by* $t + \Delta_{DNS}^{join} + \Delta_{SynchCh}$.

Property 4-19 (Justification of Fail Notifications): *For all stations s_i and all times t', if s_i signals FAIL_ind at t', then there is a time t such that $t < t'$ and s_i is joined throughout* $[t,t'[$ *and not valid during* $[t,t'[$.

To define the ordering property of the protocol, let $\prec_i$ denote the transitive reduction of the delivery order of the decisions as observed by s_i. That is, for any two decisions d and d', $d \prec_i d'$ if and only if s_i delivers d before d' and no other decisions in between. The first decision s_i delivers after joining the group is defined to have no predecessor in $\prec_i$. Let furthermore d_k, $k \in \mathbb{N}$, denote the sequence of decisions as the AP multicasts them. Since the AP multicasts exactly one decision during each slot, d_k is the decision the AP multicasts in slot k.

Property 4-20 (Strong FIFO): *For all stations s_i, all decisions d, d' and all $k \in \mathbb{N}$, if $d \prec_i d'$ and $d = d_k$, then $d' = d_{k+1}$.*

We added the adjective "strong" because the property not only requires that if a station delivers two decisions, it delivers them in the order in which the AP multicast them, but also that the station delivers all decisions the AP multicast between these two decisions. This means that this property does not permit gaps within the sequence of delivered decisions at any station. The service fulfills the properties of Integrity and Timeliness too. As the properties are very similar to the corresponding properties stated in the previous subsection, we do not provide a formal definition here. Mainly, the time bound now is $\Delta_{SynchCh}$ instead of Δ_{RelMC}.

Additionally, the service delivers an ID denoting the member to which the decision relates together with each decision.

4.3.5.2 Operation of the Protocol

Each time before the AP sends a mc frame, the synchronous channel protocol solicits its user to multicast a decision. As the synchronous channel is designed as a small bandwidth channel, there are only a small number of possible decisions, which can be coded in a few bits. Each decision relates to the member on behalf of which the mc frame is sent; that is, it relates to the owner of the corresponding entry in the polling list. The protocol piggybacks a sequence *sy* of *OD*+1 decisions on each mc frame the reliable multicast protocol sends (cf. Figure 4-20). When the AP multicasts a decision d_i the protocol inserts it at position 0 in the *sy* field and piggybacks the *sy* field on the mc frames. When the next mc frame is sent during the following slot, the protocol shifts all decisions in the *sy* field to the next higher position so that a further decision can be inserted at position 0. Thus, in this mc frame, decision d_i is transmitted at position 1 of the *sy* field. Accordingly, d_i is transmitted at position 2 during the next slot and so forth. Therefore, each decision is transmitted at positions 0,1, …, and *OD* in the *sy* field of *OD*+1 consecutive mc frames. Each valid member receives and processes at least one of these mc frames no more than $\Delta_{SynchCh} := OD \times \Delta_{Slot} + \delta_m$ time units after the AP multicast the decision.

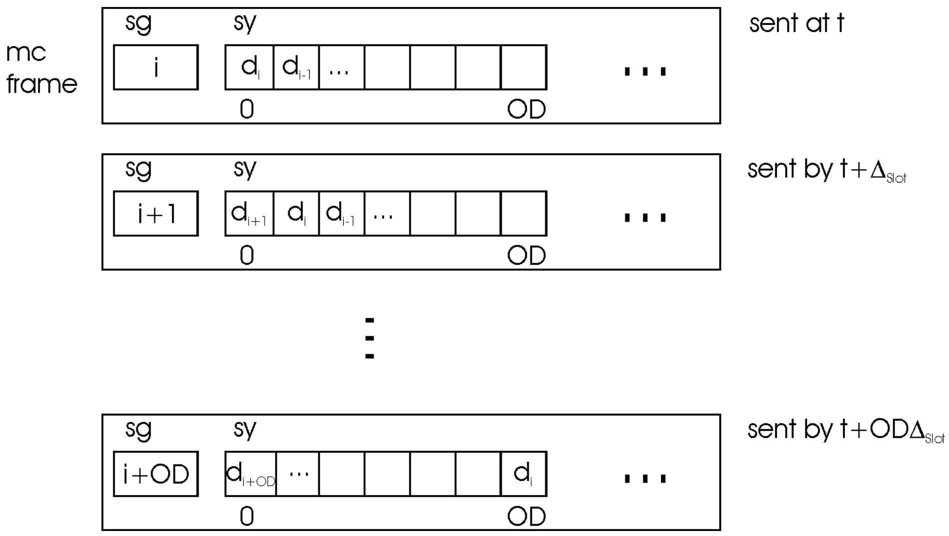


Figure 4-20. Transmission of decisions in the synchronous channel

Whenever a station receives a *sy* field piggybacked on a mc frame, it delivers exactly those decisions the AP multicast since the station received the last mc frame. The station uses the global sequence numbers in the mc frames to decide which decisions in the *sy* field it must deliver. For example, if the last mc frame the station received carried sequence number 25 and the one it currently processes has sequence number 28, the station delivers the decisions at positions 2, 1, and 0 in the *sy* field. These are the decisions the AP multicast before sending the mc frames with sequence numbers 26, 27, 28. Note that this protocol not only ensures that valid members deliver each decision the AP multicast exactly once, but also that they deliver the decisions in the order in which the AP multicast them. Using the sequence numbers, stations can also find out if they lost more than *OD* mc frames consecutively. This is the case, if the difference between the last received sequence number and the current one is greater than *OD*+1. A station that lost more than *OD* mc frames consecutively lost some of the APs decisions in the synchronous channel also. To adhere to the Strong FIFO and Validity properties, such a station stops delivering decisions, changes to service state *stopped*, and delivers a failure indication.

For the synchronous channel to be useful, stations must be able to find out which members the decisions of the AP relate to. This is because the decisions made by the atomic multicast and the membership protocol — the two users of the service — either relate to the members themselves or to their atomic multicast messages. Hence, we require the protocol to deliver with each decision a member ID that identifies the member the decision relates to. Let m_0, …, m_{n-1} be the stations in the membership view of the AP in the order in which they appear in the polling list. When the AP multicasts a decision on behalf of a member m_i, the synchronous channel protocol delivers the ID i of that member together with the decision. In static groups, stations can easily determine the IDs in the following way: Since the AP sends mc frames for the members in a round-based manner, the decision d_k relates to member $m_{k \bmod n}$. Thus, the clients are able to determine the ID ID_k to be delivered with decision d_k iteratively starting with $ID_0 := 0$ and setting $ID_k := ID_{k-1} + 1 \bmod n$ each time it has delivered a decision.

In contrast to the reliable multicast protocol where we advocate a dynamic redundancy approach for sake of efficiency, we use static redundancy for the synchronous channel. The reason for using two different approaches in the two protocols is the difference in the amount of data to be transmitted. In particular, the reliable multicast messages (possibly several hundred bytes), which are the SDUs of the reliable multicast protocol, are by far larger, than the decisions (currently, only three bits), which are the SDUs of the synchronous channel protocol. While the dynamic redundancy approach allows saving retransmissions of SDUs, it also introduces overhead for the transmission of acknowledgments. Saving retransmissions of SDUs of several hundreds bytes warrants the overhead of the *ack* field. By contrast, saving retransmissions of some three-bit decisions does not warrant additional overhead. Hence, we decided to use static redundancy to implement the synchronous channel. With this approach, we efficiently realized a reliable and timely simplex channel that enables the AP to transmit essential decisions relating to the members. This is a valuable internal service for higher layers, as will turn out in the following sub-sections.

4.3.5.3 Dynamic Group Extensions

In a dynamic group, determining the ID to be delivered with a decision gets more sophisticated. The size n of the membership, which is used to determine the IDs at the clients, is no longer fixed but subject to change. Therefore, the stations must be notified when the size of the membership changes. To this end, the protocol itself uses the synchronous channel. When a client is added to or removed from the membership view of the AP, the protocol multicasts a *new* or *exclude* decision respectively in the synchronous channel. Each member receives this decision and is therefore able to adjust its current view of the membership size accordingly. As an example, let us consider a situation where the membership size was 5 right from the start and where the protocol entity at some client is processing decision d_{25}, which is an *exclude* decision. The client iteratively updates the member IDs as described above and hence knows that $ID_{25} = 0$. The first thing the member learns is that the new membership size n' is 4. Furthermore, knowing that the member m_0 with ID 0 has been removed from the membership, the protocol entity can deduce that the successor m_1

of m_0 in the old membership has ID 0 in the new membership $m_0',\ldots,m_3'$, where $m_i' = m_{i+1}$. Generally, when a member with ID ID_{i-1} is excluded the next decision relates to $ID_i := ID_{i-1}$ mod n'. Let us now assume d_{25} were a *new* decision. First, the protocol learns that the new membership size n' is 6. Furthermore, the next decision relates to the member with ID 1 because it is the successor of the new member in the new membership $m_0',\ldots,m_5'$, where $m_i' = m_{i-1}$ for $i \in 1..5$ and m_0 is the new member. Generally, when a *new* decision relates to ID_{i-1}, the next decision will relate to $ID_i = (ID_{i-1} + 1) \bmod n'$. Thus, using the synchronous channel itself, the protocol can determine the member IDs decisions relate to a dynamic group also.

The protocol provides a notification service at the clients that informs the higher layers of additions to and exclusions from the membership. Thus, the protocol not only uses the transmission of *new* and *exclude* decisions internally, but also exploits them to offer a notification service to higher layers.

To determine member IDs in dynamic groups, it is not only necessary to inform current group members of changes in the membership, but also to provide the information they need to initialize the protocol to joining stations. In particular, a joining station must learn the following information to take part in the protocol: (a) the current membership size and (b) a pair of a decision d_k and the corresponding member ID ID_k, which together serve as a starting element for the iterative computation. To initialize joining stations, the AP piggybacks the required information on the mc frames it sends. In more detail, when the protocol entity at the AP learns that a station was added to the membership, it transmits in the following OD+1 mc frames not only the *sy* field but also the current membership size and member ID. The joining station will receive at least one of these frames. Suppose it is the frame with global sequence number k and that it carries the member ID ID and the group size gs. The joining station knows that the first decision in the *sy* field of that frame, that is, d_k, relates to the member ID $ID_k := ID$. Once the station knows this starting point and the corresponding membership size, it can iteratively determine the IDs while processing the decisions in the synchronous channel. After performing this initialization, the protocol sets the service state to *joined*.

The protocol uses piggybacking and static redundancy to transmit initialization information to joining station. The additional overhead appears to be acceptable since (a) it only becomes effective when stations join the group and (b) neither the membership size nor the member ID will incur too much overhead in the frame when efficiently coded. Each joining station that is valid for $\Delta_{SynchCh}$ time units after being added to the membership will receive at least one of the mc frames carrying the initialization information and will therefore turn its service state to *joined*.

Stations are able to detect if they are not able to provide a valid service. As explained above, by computing the difference of the sequence numbers of two consecutively received mc frames, a station is able detect if it observed more than OD+1 omissions. Since delivering decisions in this case would imply gaps in the delivery order of that station, it does not deliver any decision at all and turns its service state to *stopped*. Furthermore, the stations use a timeout to be able to indicate their failure in case they receive no more frames at all. The length of the timeout is set to $\Delta_{SynchCh}(1+\rho)$, where $\Delta_{SynchCh}$ is an upper bound on the inter arrival time of two mc frames at a valid station and $(1+\rho)$ accounts for the drift rate of the client's clocks. Since reacting to the timeout may additionally take up to Δ_{sched} time units, the duration between the reception of the last mc frame and the time when the station changes to service state *stopped* is bounded by

$$\Delta'_{SynchCh} := \Delta_{SynchCh}(1+\rho) + \Delta_{sched} .$$

Thus, the constant ε in the above definitions can be computed as

$$\varepsilon = \Delta'_{SynchCh} - \Delta_{SynchCh} = \Delta_{SynchCh}\rho + \Delta_{sched} .$$

4.3.6 Atomic Multicast

4.3.6.1 Service of the Protocol

The atomic multicast protocol provides a fail-aware service, which adds two important properties to the service of the reliable multicast protocol: agreement and total order. The property of agreement ensures that whenever a station delivers an atomic multicast, every joined station delivers the multicast also. The property of total order ensures, that whenever two members deliver the same two atomic multicasts, they deliver them in the same order and deliver the same atomic multicasts in between. Together, these properties ensure that all members deliver the same sequence of atomic multicasts in the same order, thus achieving a common view on the atomic multicasts delivered to the group. This is particularly useful when a resiliency of less than *OD* is chosen for the reliable transmission of a message. In this case, not all valid members necessarily deliver the message. If some member does not deliver an atomic multicast, the service ensures that no member delivers it so that a common view among the group members is still achieved.

More formally, the protocol fulfills the properties of Integrity, FIFO, and Timeliness. Since, it mostly "inherits" these properties from the underlying reliable multicast protocol and the definitions remain largely unchanged, we do not state them formally. We note that the time bound of the Timeliness property changes and is denoted as $\Delta_{AtomcMC}(r)$. The protocol provides a fail-aware Validity property for messages with a resiliency equal to *OD*. There is a known constant $\Delta_{AtomcMC}(OD)$ such that the following properties hold.

Property 4-21 (Validity): *For all stations s_i, s_j, all messages m, and all times t: if s_i multicasts m at t and res(m) = OD and s_i is a valid member during $[t, t+\Delta_{AtomcMC}(OD)]$ and s_j is joined during $[t, t+\Delta_{AtomcMC}(OD)+\varepsilon]$, then s_j delivers m by $t+\Delta_{AtomcMC}(OD)$.*

Property 4-22 (Agreement): *For all stations s_i and all messages m, if s_i delivers m at t, then there is a time t' such that, $t' \leq t \leq t' + \Delta_{SynchCh}$ and for all stations s_j, if s_j is joined throughout $[t', t' + \Delta_{SynchCh} + \varepsilon]$, then s_j delivers m by $t' + \Delta_{SynchCh}$.*

Agreement requires that all joined stations agree on the messages they deliver. Like the synchronous channel protocol, the atomic multicast protocol fulfills two properties, which ensure that a station is joined whenever possible. The protocol fulfills the Justification of Fail Notifications property (Property 4-19) and a Conditional Bounded Join Delay property analogous to Property 4-18, yet with another time bound.

Property 4-23 (Bounded Join Delay): *For all stations s_i, if s_i starts joining at t and s_i is valid throughout* $[t, t + \Delta_{DNS}^{join} + \Delta_{AtomcMC}(OD)]$*, then s_i is joined by* $t + \Delta_{DNS}^{join} + \Delta_{AtomcMC}(OD)$.

To define the Total Order property, we define the delivery order of atomic multicasts analogously to the delivery order of the decisions. Let $\prec_i$ denote the transitive reduction of the delivery order as observed by s_i. That is, for two messages *m* and *m*', $m \prec_i m'$ if and only if s_i delivers *m* before *m'* and no other messages in between. The first message s_i delivers after joining the group is defined to have no predecessor in $\prec_i$.

Property 4-24 (Total Order): *For all stations s_i, s_j and all messages m, m', and m''', if* $m \prec_i m'$ *and* $m \prec_j m''$*, then* $m'' = m'$.

This property requires that any two stations that have delivered the same message *m* deliver the same sequence of messages afterwards until one of them stops delivering messages. This means that partially valid or invalid stations either stop delivering messages or they deliver the same messages in the same order as the valid stations.

4.3.6.2 Operation of the Protocol

If a message has a resiliency smaller than *OD*, it is not ensured that the reliable multicast protocol delivers that message at all valid members. Thus, if a station delivers a reliable multicast it cannot be sure whether all valid members will deliver this message also. Therefore, the atomic multicast protocol does not immediately deliver the messages it receives from the reliable multicast protocol to its user. Rather, it delays the delivery until the protocol entity at the AP decides whether all members shall deliver the message (*accept* decision) or not (*reject* decision). So, it is up to the AP to decide whether all members received an atomic multicast, in which case it decides to accept the message, or whether this might not be the case, in which case it decides to reject the message. After making the decision, the AP multicasts it in the synchronous channel. Upon reception of the decision, the members either deliver the atomic multicast message if the decision is *accept*, remove it if the decision is *reject*, or wait for another transmission attempt if the decision is *no_dec.* With all members acting according to the decisions of the AP, agreement is ensured. Furthermore, as all members deliver the atomic multicasts in the order in which the AP accepts them, total order is accomplished as well.

To decide whether to accept or reject a message, the AP must know whether or not it can expect all members to have received this message. The acknowledgement mechanism of the reliable multicast protocol provides this information. Whenever the reliable multicast protocol stops transmitting a multicast message it indicates to the atomic multicast protocol whether

1. All group members acknowledged the message;
2. The message was transmitted *OD*+1 times;
3. The resiliency of the message was exceeded, and the resiliency was smaller than *OD*.

In cases 1. and 2., the AP decides to accept the message; in case 3., it rejects the message. In case 1., the AP is sure that all members received the message because they all acknowledged it. In case 2., this is ensured because each valid member must have received the message after OD+1 transmission attempts by definition.

The AP uses the synchronous channel (Sub-section 4.3.5) to multicast its decision to the members. The properties of the synchronous channel ensure that all valid members receive the decisions in bounded time. Moreover, the members deliver the decisions according to the Strong FIFO property (Property 4-20). Thus, the members deliver the atomic multicasts in the order in which the AP accepts them. When the reliable multicast protocol sends a status indication for a message of some member (say, with ID ID_k) and the atomic multicast protocol makes the corresponding decision (say, d_k), the AP is just about to transmit a mc frame on behalf of the member. Thus, the synchronous channel protocol will solicit the atomic multicast protocol to multicast a decisions on behalf of that member. The atomic multicast protocol multicasts decision d_k and all valid members will deliver d_k together with the ID ID_k. Hence, the members know that they have to accept or reject the last message originating from the member with ID ID_k.

The delay bound $\Delta_{AtomcMC}(r)$ of the atomic multicast protocol can be determined as follows: If a message with resiliency r is multicast at time t, the latest possible time at which the message may be received and delivered to the atomic multicast protocol by some station is $t' := t + \Delta_{RelMC}(r)$. The AP receives the acknowledgements for the last transmission of the message by time $t'' := t' + \Delta_{Round} - \delta_m$. At this time, it multicasts its decision in the synchronous channel. So, each valid station will deliver and process the decision by time $t''' := t'' + \Delta_{SynchCh}$. Thus, the time bound can be computed as:

$$\begin{aligned}
\Delta_{AtomcMC}(r) &:= \Delta_{RelMC}(r) + \Delta_{Round} - \delta_m + \Delta_{SynchCh} \\
&= 2 \times r \times \Delta_{Round} + 2 \times \delta_m + \Delta_{Round} - \delta_m + \delta_m + OD \times \Delta_{Slot} \\
&= (2r + 1) \times \Delta_{Round} + 2\delta_m + OD \times \Delta_{Slot} \\
&\leq (2r + 1) \times \Delta_{Round} + (OD + 1) \times \Delta_{Slot}
\end{aligned}$$

4.3.6.3 Dynamic Groups Extensions

Now let us consider the new challenges that appear when considering dynamic groups. The main problem here is that we can no longer assume that a client is a valid group member right from the start of every atomic multicast transmission. Rather, when the AP adds a further client to the group, several atomic multicasts may be in progress; that is, for such atomic multicasts several transmission attempts may already have taken place, such that it cannot be guaranteed that the newly added client will receive these messages. This is not so much of a problem for atomic multicasts with a resiliency smaller than OD: If the AP does not get an acknowledgement for the message from all members (including the newly added one), it rejects it and no member will deliver it. But, if such a message has a resiliency of OD, the AP must decide to deliver it. Thus, for all atomic multicasts with resiliency OD that are in progress when a client is added to the group, it could happen that the message is accepted but the new member cannot deliver it; we call such messages “pending multicasts” henceforth. In particular, as long as a pending multicast exists, it could happen that a new member delivers some atomic multicasts and some not. Of course, a situation in which a new member delivers some atomic multicasts but does not deliver others, which all the other members deliver, should be avoided at any rate.

To avoid an inconsistent behavior of joining stations as described above, the protocol ensures that a station only starts delivering atomic multicasts when it is able to deliver all atomic multicasts that the other members deliver henceforth, as long as it stays valid. To achieve this, the AP delays accepting a joining station's atomic multicast until it has ensured that there are no more pending multicasts. The joining station, in its turn, does not deliver any atomic multicast message before its own one is accepted. When the atomic multicast of a joining station is accepted, the joining station changes its service state to *joined*. To ensure that this will happen whenever the joining station is valid, the first atomic multicast of the joining station is transmitted with a resiliency of OD. Determining and maintaining the pending multicasts for a joining station works as follows: When the DNS protocol signals that a new station was added to the group, the reliable multicasts exports the current set of pending multicast messages; or, to be more precise, the set of stations, that have pending multicasts. When the atomic multicast protocol receives the signal, it imports this set and stores it as the pending multicast set of the joining station. Thus, each joining station can have its own set associated. The AP changes this set under two events: First, whenever the joining station acknowledges the multicast of a station, this station is removed from the pending multicast set of the joining station because now the joining station is able to deliver the message of this station. Second, whenever the AP decides to accept or reject a message, it removes the originator of the message from the pending multicast set of all joining stations; none of the joining stations will have to deliver the message. Note that the pending set of a joining station will be empty when the atomic multicast of that station has been transmitted OD+1 times. In this case, all pending multicasts must have been accepted already. Therefore, the multicast of a joining station is never delayed beyond the time bound $\Delta_{AtomcMC}(OD)$ so that a bounded join delay can be guaranteed. This protocol not only ensures that a member delivers all messages in agreement with the other group members starting from the first message it delivers, but also that the members know they are in agreement with a new member after delivering the first atomic multicast of that member.

The atomic multicast protocol achieves agreement among the members, even if not all valid members receive a message. As members process the decisions in the synchronous channel in the order in which the AP made them, it achieves total order as well. The analysis of the timeliness of the atomic multicast service yields a worst-case delay of $\Delta_{AtomcMC}(r) = (2r + 1) \times \Delta_{Round} + (OD + 1) \times \Delta_{Slot}$. To achieve these properties no additional frames were introduced.

4.3.7 Membership

The membership protocol provides an up-to-date and agreed view on the current group membership to all group members (Schemmer and Nett 2003b). Furthermore, it ensures that all member deliver atomic multicasts in the context of the same membership view. This is a very strong semantics, which significantly facilitates the development of distributed applications. As the membership protocol is intended for dynamic groups only, this sub-section directly addresses the dynamic group case.

4.3.7.1 Service of the Protocol

The fail-ware service of this protocol is to provide an up-to-date and agreed view on the current membership to the group members. Up-to-date means that there is a known constant Δ_{Mem}^{join}, such that each station which tries to join the group and is valid for at least Δ_{Mem}^{join} time units becomes part of the membership view of every joined member, and a known constant Δ_{Mem}^{excl}, such that a member that was invalid for at least the last Δ_{Mem}^{excl} time units will not be in the membership view of any joined member. When we say that views are agreed, we mean that any two joined members deliver the same sequence of membership views. An additional property of the service, which is defined in conjunction with the atomic multicast service, is the so-called virtual synchrony. Virtual synchrony ensures that all members deliver the membership views at the same position in the sequence of atomic multicasts; or, to put it the other way round, it ensures that all members deliver the atomic multicasts in the context of the same membership view. Together with the Agreement and Total Order properties of the atomic multicast protocol (Property 4-22 and Property 4-24), it ensures that all members share a common view on the sequence of atomic multicast messages and membership changes.

For the formal definition of the protocol's properties, we define that a membership change message is a pair (n,m), where m is a set of stations representing the membership view and n is a unique identifier. If the protocol delivers the same membership view several times — for example, a station leaves and later rejoins the group — the identifier allows distinguishing the membership change messages. As the atomic multicasts are delivered at the same interface, the application observes a single sequence of atomic multicast and membership change messages. We say that a station s_i delivers a message m if the membership protocol delivers a membership change or atomic multicast message m and that s_i delivers a membership view m if it delivers a membership change message containing m. Let $\mathcal{M}_i(t)$ be the last membership view s_i delivered by time t. The first two properties define the service the protocol provides at joined stations. There exist known constants Δ_{Mem}^{excl} and Δ_{Mem}^{join} such that the following properties hold.

Property 4-25 (Validity): *For all stations s_i, s_j and all times t, t' with $t' \geq t$, if s_i is joined throughout $[t, t' + \varepsilon]$ and*

(1) s_j started joining by $t - \Delta_{Mem}^{join}$ and is valid during $[t - \Delta_{Mem}^{join}, t']$, then s_j is in $\mathcal{M}_i(t'')$ for each $t'' \in [t,t']$.

(2) s_j is invalid during $[t - \Delta_{Mem}^{excl}, t']$, then s_j is not $\mathcal{M}_i(t'')$ for each $t'' \in [t,t']$

Property 4-26 (Agreement): *For all stations s_i, and all membership change messages (n,m), if s_i delivers (n,m) at t, then there is a time t' such that, $t' \leq t \leq t' + \Delta_{SynchCh}$ and for all station s_j, if s_j is joined throughout $[t', t' + \Delta_{SynchCh} + \varepsilon]$, then s_j delivers (n,m) by $t' + \Delta_{SynchCh}$*

Like the fail-aware services of the atomic multicast and the membership protocol, the membership protocol fulfills the Justification of Fail Notifications property (Property 4-19) and a Conditional Bounded Join Delay property analogous to Property 4-18.

Property 4-27 (Bounded Join Delay): *For all stations s_i, if s_i starts joining at t and s_i is valid throughout $[t, t + \Delta_{Mem}^{join}]$, then s_i is joined by $t + \Delta_{Mem}^{join}$*

Like for the synchronous channel and the atomic multicast protocol, we define the delivery order of membership change messages. Let $\prec_i$ denote the transitive reduction of the delivery order as observed by s_i. That is, for any two membership messages (n,m) and (n',m'), $(n,m) \prec_i (n',m')$ if and only if s_i delivers (n,m) before (n',m') and no other membership changes messages in between. The first membership change message s_i delivers after the membership service becomes *joined* is defined to have no predecessor in $\prec_i$.

Property 4-28 (Total Order). *For all stations s_i, s_j and all membership change messages (n,m), (n',m'), and (n'',m''): If $(n,m) \prec_i (n',m')$ and $(n,m) \prec_j (n'',m'')$, then $(n',m') = (n'',m'')$.*

Total Order ensures that two stations, once they have delivered membership change messages with the same identifier, deliver the same sequence of memberships henceforth until one of them is no longer joined.

Property 4-29 (Virtual Synchrony). *For all stations s_i, s_j, and all membership change messages (n,m), (n',m'): If $(n,m) \prec_i (n',m')$ and $(n,m) \prec_j (n',m')$, then s_i and s_j deliver the same set of atomic multicasts between (n,m) and (n',m').*

4.3.7.2 Operation of the Protocol

To explain how the membership protocol accomplishes its service, we first point out how stations leaving the group are handled and then consider joining stations. There are two reasons for a station's no longer being a part of the group. First, the station may voluntarily decide to leave the group, and second, it may become invalid. In both cases, the protocol must deliver new membership views at the remaining members. For sake of simplicity, we do not distinguish between these cases.

To handle leaving members, the main point to be addressed is how the remaining members learn that a certain station left the membership; then, they are able remove this member from their local membership view and deliver the changed view to the user. The first station that notices that one of the members became invalid is the AP. As described above (Sub-Section 4.3.3), the dynamic network scheduling entity at the AP provides an exclude notification whenever the polling entity informs it of a member that became invalid. To propagate this exclude information from the AP to the clients the membership protocol uses the synchronous channel. Actually, as explained above (Sub-Section 4.3.5), the synchronous channel protocol itself already distributes these change notifications for its own purposes and issues corresponding notifications at the clients. Thus, using the synchronous channel protocol, all that remains to be done for the membership protocol is to change the membership view whenever it receives an exclude indication from the synchronous channel protocol and deliver the new membership view to the user. The Validity property of this protocol is ensured by the timely detection of invalid members (Property 4-8) and by the Validity of the synchronous channel (Property 4-17). Thus the bound Δ_{Mem}^{excl} can be determined as follows:

$$\Delta_{Mem}^{excl} := \Delta_{Fail} + \Delta_{SynchCh} := OD \times \Delta_{Round} + (OD + 1) \times \Delta_{Slot}$$

Furthermore, since the exclude decisions are transported in the synchronous channel together with the AP's *accept* decisions and since all members process the decisions in the order in which the AP made them, the protocol ensures virtual synchrony. To achieve timely and virtual synchronous delivery of exclusions, the membership protocol does not introduce additional frames.

The second kind of membership change that must be dealt with is the joining of new members. To understand how joining works, it is best to think of it in the following way: A station joining the group uses the atomic multicast service to multicast a message (with resiliency *OD*) to the current membership plus itself. Its address is transmitted implicitly with the multicast message. When the atomic multicast service delivers the message of the joining station, all members as well as the joining station add the joining station to their membership view, deliver a membership change message, and then deliver the atomic multicast message.

Let us now consider the joining of a new member in more detail. A station that wants to join the group calls the dynamic network scheduling protocol to allocate bandwidth. Basing on the *JOIN* service of the polling protocol the DNS protocol can make this allocation in bound time ($\Delta_{Res}^{join} := (OD + 1)\Delta_{Round} + 2\delta_m$, see Sub-Section 4.3.3). Once the AP made that allocation, it starts polling the joining station. Upon reception of the first polling frame, the membership at the joining station solicits its user to transmit an atomic multicast. The protocol sets the resiliency of this atomic multicast to *OD* to ensure that the AP will accept it and invokes the atomic multicast protocol to transmit it. From then on, everything works as for a normal atomic multicast. Sooner or later, all members plus the joining station will deliver the atomic multicast together with the address of the originating station. Now, they add this address to their membership view and deliver it to the application followed by the atomic multicast. Thus, the delay bound Δ_{Mem}^{join} can be computed as follows:

$$\begin{aligned}
\Delta_{Mem}^{join} &:= \Delta_{Res}^{join} + \delta_m + \Delta_{AtomcMC}(OD) \\
&= (OD + 1)\Delta_{Round} + 2\delta_m + \delta_m + (2OD + 1) \times \Delta_{Round} + (OD + 1) \times \Delta_{Slot} . \\
&= (3OD + 2)\Delta_{Round} + (OD + 2) \times \Delta_{Slot}
\end{aligned}$$

The term δ_m accounts for the first regular polling frame the AP sends to the joining station.

When a joining station delivers its own first atomic multicast, it is expected to deliver a membership view; but, how does a joining station know which other stations are members of the group. The AP is in charge of providing the necessary membership information to joining stations. Whenever the AP sends a mc frame on behalf of a joining station, it piggybacks a copy of its current membership view. Thus, when a joining station has acknowledged its own atomic multicast message, the AP knows that the station has received the membership view together with the atomic multicast. The membership entity at the AP can stop piggybacking the membership view when it learns that the atomic multicast of a joining station was accepted. This information is provided by the atomic multicast protocol through a status indication. Once a joining station has received a mc frame including a current membership view, it can keep this list up-to-date using the decisions it receives in the synchronous channel. Thus, when the station receives the *accept* decision for its own atomic multicast, it has an up-to-date membership view, which it can deliver to the user.

The protocol allows valid station to join the group in bounded time. Furthermore, as joining stations are allowed to transmit a first atomic multicast while joining the group, the protocol needs no additional frames as long as a joining station has indeed some application message to transmit. In this case, all information is piggybacked on frames that have to be sent anyway. Virtual synchrony is efficiently achieved through the total order property of the atomic multicast protocol.

4.3.8 Event Service

4.3.8.1 Service of the Protocol

The Event Service allows the reliable, timely, and totally ordered transmission of events and associates with each event it delivers a global system state (Schemmer et al. 2001). The global system state of a group of mobile systems consists of the local states of its members w.r.t the same point of time on the global time base, which is provided by the clock synchronization protocol. The local state of a mobile system consists of those state variables of the system that serve as input for the control application. The delivery of events is either explicitly triggered by the user who requests the transmission of an event or implicitly triggered by changes in the membership. Thus, the Event Service combines event and state semantics. According to our approach, the common views on the global system state allow the mobile systems to decide locally yet achieve a coordinated behavior. Furthermore, attaching global states to events is well suited for dynamic groups, where a newly joining station may not have sufficient state information to interpret pure event information.

Regarding the reliable, timely, and totally ordered transmission of events, the Event Service mainly relies the corresponding properties of the atomic multicast and the membership protocol. We therefore focus on the main new feature the Event Service provides — the delivery of global states. To explain the notion of global states, we introduce a computational model for the state of the controlled system. To this end, we define the local state $z_i(t)$ of a mobile system s_i to include exactly those state variables of s_i that serve as an input for the control application. Let $\mathbf{g}(t) := (s_{i_1},...,s_{i_n})$ be a vector comprising the cooperating mobile systems at time t. The global state $\mathbf{z}(t) := (z_{i_1}(t),...,z_{i_n}(t))$ of this group is the vector of the local states of the stations in $\mathbf{g}(t)$. A concrete example of a global state for the shared spatial resources scenario was defined in Section 3.1, where the local states of the mobile systems were defined to consist of their position and speed. The global state changes in two ways: discretely and continuously. In particular, the joining and leaving of mobile systems constitute discrete state changes, since they alter the dimension of the global state. Discrete state changes are represented as events. Since event delivery within the group is based on the atomic multicast and the membership protocol, we suppose that all members perceive a totally ordered sequence of events e_i, $i \in \mathbb{N}$. The time at which event e_i is observed will be denoted as t_i. Whenever it delivers an event e_i, the Event Service determines the global state $\mathbf{z}(t_i)$ and delivers it together with the event to the application. When the event e_i and the global state $\mathbf{z}(t_i)$ are delivered at mobile system s_j, the control application makes a decision $d_j(\mathbf{z}(t_i))$, which determines how the local state of s_j evolves until the next event is received. This decision represents the output of the application to the underlying controllers, such as the scheduled enter time for the hot spot in the shared spatial resources scenario (Section 3.1). The local states are assumed to evolve continuously during the intervals between two events. We assume that this kind of state changes, which we denote as the dynamics of the local states, is described by a known function F, which is defined as follows: For each event e_i, $i \in \mathbb{N}$, each mobile system $s_j \in \mathbf{g}(t_{i-1})$, and each time $t \in [t_{i-1}, t_i]$ holds:

$$z_j(t) = F\big(z_j(t_{i-1}), d_j(\mathbf{z}(t_{i-1})), t_{i-1}, t\big) ,$$

where $\mathbf{z}(t_0)$ is the state of the initial group, which is assumed to be empty. Note that the local state of s_j during $[t_{i-1}, t_i]$ depends on the global state $\mathbf{z}(t_{i-1})$. This is because, during this interval the systems exhibit a coordinated behavior, where, according to our approach, each system decides about its behavior locally based on the last computed global state. Consider again the shared spatial resources scenario as an example. When a further system enters an approaching zone, this constitutes an event e_i, which is delivered together with the global state $\mathbf{z}(t_i)$. Based on this global state, each mobile system decides locally at what time it will enter the hot spot. This decision is represented by $d_j(\mathbf{z}(t_i))$ in the computational model; in the application scenario, this function is called the scheduling function since it determines the schedule for the hot spot (actually d_j is a projection of the scheduling function comprising only that component of the schedule which corresponds to s_j). From then on, until the next event is delivered, speed and position of s_j can be predicted based on its position and speed at t_i and the enter time for the hot spot scheduled at t_i.

In this model, it is the task of the Event Service to deliver the global state $\mathbf{z}(t_i)$ with each event e_i. But, as a matter of fact, the global state $\mathbf{z}(t_i)$ cannot be observed exactly in a distributed system. For one thing, stations do not have an exact notion of $\mathbf{g}(t_i)$ when they deliver e_i. Rather, a station s_j delivers e_i in the context of a membership view $\mathcal{M}_j(t_i)$. However, due to the Virtual Synchrony property of the membership protocol all stations agree on the membership view in the context of which they deliver e_i so that e_i is delivered in a unique membership view $\mathcal{M}(t_i)$. Furthermore, due to the Validity of the membership protocol, $\mathcal{M}(t_i)$ is an up-to-date approximation of $\mathbf{g}(t_i)$. Another deviation from the computational model is that e_i is actually not delivered at exactly the same time t_i at all stations. The Agreement properties of the atomic multicast and the membership protocol, however, ensure that the difference between the delivery times is bounded by $\Delta_{SynchCh}$. Finally, the global time base is also an approximation so that precision and drift of the global clock cause deviations from the ideal global state. Again, due to the properties of the clock synchronization protocol, both precision and drift are bounded.

4.3.8.2 Operation of the Protocol

To avoid complex and communication intensive protocols for global state determination, we exploit the known dynamics F of the local states and base our protocol on local computations as far as possible. In this approach, event transmission and global state determination works as follows: To multicast an event, a group member s_j sends an atomic multicast containing the event type, its current local state $z_j(t')$, and a timestamp from the global clock t'. When a member delivers the atomic multicast at global clock time t_i, it uses the last computed global state and the function F to compute $\mathbf{z}(t') := (F(z_{i_1}(t_{i-1}), d_{i_1}(\mathbf{z}(t_{i-1})), t_{i-1}, t'), ..., F(z_{i_n}(t_{i-1}), d_{i_n}(\mathbf{z}(t_{i-1})), t_{i-1}, t'))$ and replaces the component of $\mathbf{z}(t')$ corresponding to s_j with the value $z_j(t')$ received in the atomic multicast. Afterwards, it computes the global state $\mathbf{z}(t_i)$ based on $\mathbf{z}(t')$ and F, and delivers it together with the event to the application. Thus, the protocol manages to update the global state with a single atomic multicast. It allows the sender of an event to update its component in the global state with a more up-to-date value.

Delivering global states when a station leaves the group works similar. When the membership protocol delivers a membership change message at global clock time t_i indicating that a station s_j left the group, the Event Service computes the current global state $\mathbf{z}(t_i)$ based on $\mathbf{z}(t_{i-1})$ and F. It removes the component corresponding to s_j from the global state and delivers the global state together with the event to the application.

When joining stations are considered the problem gets more complicated. In particular, a joining station cannot compute the current global state as described above because it does not know the last computed global state. Hence, it is necessary to provide the last computed global state to the joining station. To this end, we apply the following protocol when a station is joining the group. While joining the group, a station s_j sends an atomic multicast, called a request message, containing its local state $z_j(t)$ and a timestamp t from the global clock ($rqu(s_j,z_j(t),t)$). The membership protocol delivers the request message immediately after the first membership view containing s_j. A single member is elected to respond to the request message. Each member decides locally whether it is in charge of sending the reply by applying the following common rule: If the member is part of the global state already, it takes over responsibility if it was the last station added to the global state. If it is not yet part of the global state, it takes over responsibility if it is the “oldest” station in the membership view; that is, if all other stations in the membership view joined after it. Due to the Virtual Synchrony of the membership protocol, a unique member is elected. The elected member multicasts a so-called “in message” in reply to the request message. The in message contains the name of the sender of the request message, the local state and timestamp that have been provided in the request message, and the last computed global state $\mathbf{z}(t_{i-1})$ together with the timestamp t_{i-1} ($in(s_j,z_j(t),t,\mathbf{z}(t_{i-1}),t_{i-1})$). The reception of the in message constitutes the event e_i corresponding to s_j’s becoming part of the group; all three kinds of messages indicating events that have been discussed so far a referred to under the common term event messages. When delivering the in message, the joining system learns the last computed global state $\mathbf{z}(t_{i-1})$ and the timestamp t_{i-1}. With this information it is able to compute the current global state $\mathbf{z}(t_i)$ as described above. Likewise, all members that already knew $\mathbf{z}(t_{i-1})$ are able compute $\mathbf{z}(t_i)$ when they receive the in message. A joining station will obviously not deliver an in message if it is the only station in the membership since there is no member to reply to its request. A joining station detects this situation when it delivers the first membership view and delivers a global state consisting of its own local state only.

We now consider the case in which two stations, say s_j and s_k, start joining at approximately the same time. Suppose that s_j’s request message is received first and that the system replying to s_j’s request with an in message receives s_k’s request before it receives the in message corresponding to s_j’s request. If s_j replies to s_k’s request immediately, it sends the same global state $\mathbf{z}(t_{i-1})$ as in the previous in message, since a new global state $\mathbf{z}(t_i)$ has not yet been determined. However, if first the in message for s_j and then the in message for s_k was delivered (events e_i and e_{i+1} respectively), s_k would compute the current global state from $\mathbf{z}(t_{i-1})$, not $\mathbf{z}(t_i)$. Since $\mathbf{z}(t_{i-1})$ does not include the local state of s_j, s_k would compute a global state $\mathbf{z}(t_{i+1})$ not including the local state of station s_j.

One approach is to prohibit members to reply to request messages between multicasting an in message and delivering it; that is to say, between reading and updating the global state. This approach, however, would restrict concurrency and would thus result in increased delays for event propagation. Furthermore, it would still be possible that other event messages would be delivered between the request and the corresponding in message. Hence, we decided to let the members reply to request messages immediately. According to what was said above, this means that a station cannot be sure that the global state included in the in message corresponding to its request is the last computed global state of the group. Rather, it is possible that other in messages have been delivered between multicasting and delivering this in message and that several new global states have been computed meanwhile as a consequence. However, since request messages and in messages are delivered totally ordered, the joining system knows which in messages have been delivered between its request message and the corresponding in message. Likewise, it knows of any other event message that is delivered between its request message and the corresponding in message.

We exploit this fact in the following way. The joining system stores the event messages it delivers between its request message and the corresponding in message in a FIFO queue. Together with each message it stores the global time at which it delivered the message. When it delivers the in message corresponding to its request, it uses the global state provided therein as an initial global state. Now, it successively computes the global states corresponding to the event messages in its FIFO queue. The last event message in this sequence is the in message corresponding to its request. After processing this last in message, it delivers the global state to the user.

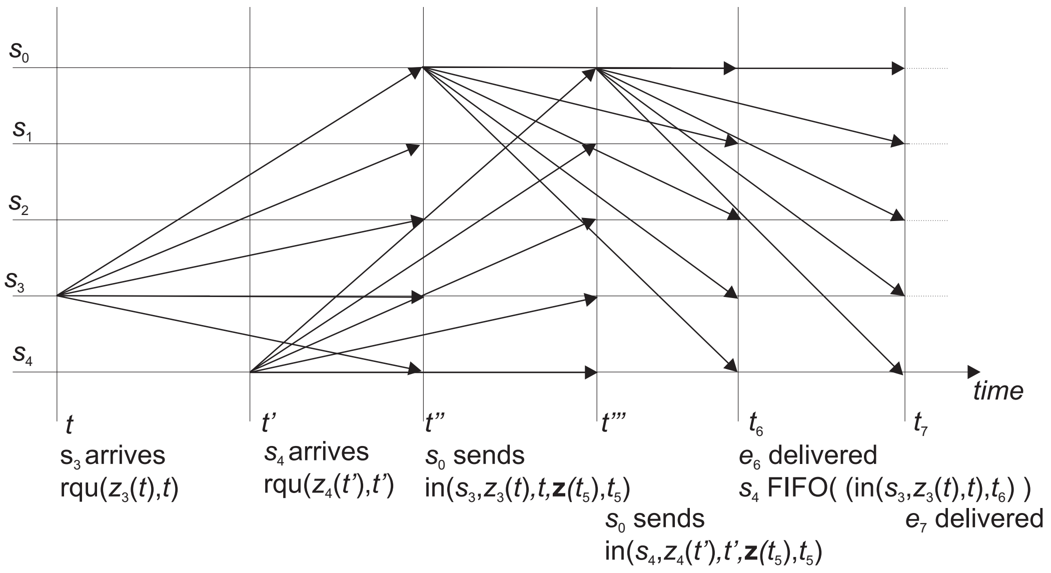


Figure 4-21. Systems s_3 and s_4 join the group.

Consider, for example, the situation depicted in Figure 4-21. The mobile systems s_3 and s_4 start joining the group at times t and t' respectively and both multicast a request message (denoted as $rqu(s_3,z_3(t),t)$ and $rqu(s_4,z_4(t'),t')$ in the figure). System s_0 replies to both of these requests at times t'' and t''' respectively. Assume that the last computed global state at t'' is $\mathbf{z}(t_5)$, which was delivered together with e_5 at some previous global time t_5. So, s_0 includes the same global state $\mathbf{z}(t_5)$ in both in messages. When, at time t_6, s_3 delivers the in message corresponding to its request, it uses the global state $\mathbf{z}(t_5)$ with timestamp t_5 and the local state $z_3(t)$ with timestamp t to compute $\mathbf{z}(t_6)$ and delivers e_6 together with $\mathbf{z}(t_6)$. Station s_4, on the other hand, stores the information s_3, $z_3(t)$, and t together with time t_6 in its FIFO queue. At time t_7, when it receives the in message corresponding to its request, station s_4 computes the global state $\mathbf{z}(t_6)$ using the global state information $\mathbf{z}(t_5)$ with timestamp t_5 in the in message and the information $(z_3(t),t,t_6)$ it stored in its queue. Afterwards, it uses the information $(z_4(t'),t')$ provided in the in message to compute the global state $\mathbf{z}(t_7)$, which it delivers together with the e_7 to the user.

There are two approaches to deal with situations in which the station elected to respond to a request becomes invalid before the in message is delivered. First, due to the timeliness of the underlying atomic multicast protocol a joining station can detect in bounded time when the in message is not delivered. It indicates this fact to the application, which is able to perform a timely exception handling; for example, by bringing the mobile system into a safe state. Second, since the membership protocol notifies the remaining members that some member left the group, they can react by electing a new member that is in charge of sending the in message. This approach allows ensuring that each valid joining station will deliver a global state. The delay, however, depends on the number of stations that become invalid after being elected to respond to an in message. Which approach is more appropriate depends on the application at hand, particularly on the timing constraints. In fact, a combination of both approaches can be adopted where a bounded number of responder failures are tolerated.

Obviously, the actual behavior of a physical system will always deviate marginally from the behavior specified by its dynamics. As long as the time intervals considered are short this effects can be neglected; we did this, for example, in the hot spot prototype. To accommodate longer intervals of prediction, it is possible to extend the computational model so that it captures the increasing uncertainty stemming from increasingly far-ranging state predictions explicitly. For example, a Kalman filter based approach would allow modeling this kind of uncertainty through Gaussian distributions with increasing standard deviations. Furthermore, the Event Service itself can be used to react to local states that deviate from the model too much. If each station monitors its local state and compares it to the model, it is able to issue an event in case the deviation between model and reality exceeds a given threshold. As explained above, this event carries the current local state of the station so that all group members can update their estimates with the new value.

The presented protocol only needs one or two atomic multicasts for event transmission and global state determination. Since the Event Service is based on the membership and the atomic multicast protocol, reliability, timeliness, and total order are achieved by the properties of those protocols. Not only is the state prediction of the protocol a very efficient way to determine global state, it also has the following two advantages:

1. When states are transmitted in state messages the temporal consistency achievable is bounded by the delay of the messages. Using state prediction allows overcoming

this limitation, because the prediction model accounts for the changes of the observed state during the transmission of the state message (Mock 2003).

2. State prediction is used to relate local states observed at different points of time to the same reference time so that they can be combined into a time coherent global state.

4.4 Related Work

The presentation of the related work pursues a bottom up approach. At first, Sub-Section 4.4.1 deals with related works regarding the underlying system modeling. In Sub-Section 4.4.2, we consider approaches to real-time communication over wireless LANs, where our approach is to use the contention-free access method specified in the IEEE 802.11 Standard and its up-coming supplement 802.11e. Subsequently, in Sub-Section 4.4.3, we turn towards reliable multicasts. The main aspect to be considered here is how the problem of achieving a reliable and timely transmission in the presence of a varying number of message losses is addressed. The following sub-section (4.4.4) examines how total order and agreement is achieved by other protocols, before membership is considered in Sub-Section 4.4.5. In Sub-Section 4.4.6, we compare our Event Service to other communication paradigms for cooperative systems.

Our work on developing the group communication protocols in a modular manner was inspired by the Horus system (van Renesse et al. 1996). Horus comprises several micro protocols that can be stacked on top of each other like “Lego blocks”. Their objective was a maximum of flexibility; so, different stacks can run concurrently and can be configured at runtime. Our objective is configurability at compile time and a predictable and performant timing behavior at runtime. Other group communication protocol also provide services with differently strong semantics; the basic idea was already present in the ISIS toolkit (Birman and van Renesse 1994). xAMP (Rodrigues and Verissimo 1992) provides multiple primitives and Timewheel (Mishra et al. 2002) allows using three different ordering and three different agreement semantics. Both of these protocols, however, implement all the services in a single core protocol so that the protocols are not modular.

4.4.1 System Modeling

As explained in Sub-Section 4.1.1, a basic distinction in system modeling is whether services or components have timing specifications. System models without timing specifications are referred to as time-free asynchronous systems[3]. In *time-free asynchronous systems* (Fischer et al. 1985) each component is correct as long as it delivers a functionally correct service, no matter at what point of time it is delivered. Although this model has been adopted in many works on group communication, it is not a viable foundation for the design of the communication protocols in our middleware for the following two reasons:

[3] It is frequently just called the asynchronous system model; (Cristian, 1995) added the additional adjective “time-free” when he introduced the timed asynchronous system model.

First, as the timing behavior of time-free asynchronous systems is not specified such that even correct systems may exhibit a completely unpredictable timing behavior, it is not possible to design protocols observing a predictable timing behavior in this model. Second, according to a well-known result of (Fischer et al. 1985) consensus cannot be achieved in such systems if at least one process may be faulty.

Synchronous systems (Cristian 1991,Cristian 1996,Galleni and Powell 1996) can be considered the reverse end of the spectrum. In synchronous systems, service specifications are timed. Communication is assumed to be reliable and timely (also called *certain* (Cristian and Fetzer 1999)). Obviously, assuming reliable and timely communication is not reasonable in a wireless environment. Weaker synchronous system models have been proposed also, in which the communication service may exhibit a bounded number of omission failures (Verissimo et al. 1991). The upper bound is referred to as the *omission degree*. In these models, certain communication can be achieved using timing redundancy (Rodrigues and Verissimo 1992,Kopetz and Grünsteidl 1993). In our environment we cannot assume that an upper on the number of omission failures hold for all stations at any time. The quality of the communication link between two mobile systems changes with their relative positions and may observe any state between disconnection and nearly reliable communication. On the other hand, our model is similar to a synchronous model with bounded omissions in the sense that our protocols achieve certain communication as long as an upper bound holds.

(Cristian and Schmuck 1995,Cristian 1996,Cristian and Fetzer 1999) presented the *timed asynchronous system model*, which is situated between the synchronous and the time-free asynchronous system model. In fact, the model was introduced even earlier (Cristian 1989), however, not explicitly named and distinguished from the time-free asynchronous system model. In this model processor and communication services have timing specifications. The timing specifications, however, represent likely time bounds holding most of the time rather than worst-case bounds. Therefore the components may be subject to timing failures. In fact, the possibility of timing failures can be considered one of the distinctive features of this model: Whereas time-free systems have no timing specifications and hence no timing failures, synchronous systems have a timing specification but are assumed to meet it. The other distinctive feature of the timed asynchronous system model as compared to the time-free model is the existence of hardware clocks having a bounded drift w.r.t real-time. These clocks can be used to detect timing failures of the components.

The timed asynchronous system model, like the time-free asynchronous system model, still describes a system that exhibits an unpredictable timing behavior in general. Timely progress can be guaranteed neither for the computation nor for the communication services. Yet, timing specifications are assumed to be chosen in such a way that the system alternates between long phases during which services adhere their timed specifications and only short phases where they do not. The basic idea regarding service guarantees is to give a conditional guarantee for a timely progress of the services instead of giving no guarantees at all: The protocols guarantee a timely progress whenever the underlying communication and processor services exhibit a "sufficient synchrony". What "sufficient synchrony" means is described by *stability predicates*. So, the protocols guarantee a timely progress whenever the system, or a part thereof, is stable for a sufficiently long interval of time. Safety properties, on the other hand, are guaranteed to be met always, whether the system is stable or not. For example, in the group membership service defined in (Cristian 1996), agreement on group membership is an unconditional property — any two stations joined to the same group agree on its membership —, whereas a bounded join delay is a conditional property — two processes connected throughout some interval *I* are guaranteed to be joined to the same group after *I* if the system is stable during *I*. Working with conditional properties has two advantages

- Even though a general guarantee cannot be given, the protocols provide guarantees for those times at which the system is sufficiently synchronous, rather than giving no guarantees at all. In a well tuned system, which is stable most of the time, this may be sufficient for many applications

- The conditional properties can be converted into unconditional ones by strengthening the underlying model without changing the protocols. This is achieved by adding progress assumptions, which essentially require that the system eventually becomes stable for a sufficiently long time. Obviously, this means that the underlying system must be designed or tuned to warrant the progress assumptions.

Since a timed asynchronous system alternates between two phases it can also be considered as a heterogeneous or asymmetric system. There are intervals of time during which it behaves like a synchronous system and there are intervals of time during which it behaves like an asynchronous system. Which phases prevail depends on the choice of the timing specifications and on the runtime conditions of the system.

The timed asynchronous system model is targeted to match most of existing "run-off-the-mill" distributed systems consisting of workstations without a particular real-time OS connected by networks that exhibit phenomena like congestion and hence a hardly controllable timing behavior. It models a basically asynchronous system, which provides no guarantee the any progress is achieved in time and in which any component may be late. Though generally detectable, timing failures cannot be detected and handled in a timely manner. Regarding timeliness, a model with stronger guarantees is better suited to match our intended system environment where real-time scheduling of CPU resources is performed (see Chapter 5) and a contention-free access to a single LAN is provided. This allows guaranteeing a timely progress for at least part of the computational tasks and giving an upper bound on the delay of messages on the network. W.r.t timeliness we therefore adopt a stronger model, which allows focusing the protocol design on the main source of unpredictability in the wireless LAN — the omission failures.

Using stability predicates and conditional properties is a promising approach to design protocols for unpredictable environments. It leverages protocols providing safety under a wide range of conditions and provide progress whenever possible. We adopt this approach in our system model. In particular, we introduced a *valid* predicate similar to the notion of *F-connected* introduced in (Cristian and Fetzer 1999) and progress properties conditioned on that predicate. Furthermore, we adopt the idea of making services fail-aware (Fetzer and Cristian 1996). Fail-awareness allows strengthening the conditional properties in the sense that if some station does not fulfill the condition (is not valid in our model) and hence is not able to fulfill its progress properties, it indicates this fact to its user, thus enabling the user to react to this situation. It should be noted, however, that in a timed asynchronous system model, a timely reaction to such an exception indication cannot be ensured.

(Verissimo et al. 2000,Casimiro and Verissimo 2001) also present a heterogeneous model, called the *Timely Computing Base (TCB) model*, which is a continuation of their previous work an the so-called *quasi-synchronous system model* (Almeida and Verissimo 1996). In this model the system is statically divided into two parts: a payload part and a control part. The payload part may have any degree of synchronism, whereas the TCB components together constitute a synchronous subsystem. The payload part can be modeled by a timed asynchronous are quasi-synchronous system model. Since the TCB exhibits a synchronous behavior all of the time and allows executing well-defined functions in bounded time, it allows for stronger timeliness properties than the timed asynchronous system model. Particularly, it allows detecting and handling timing failures in time. While regarding the local task execution, our system would fit to the TCB model, namely the timely exception handling of TAFT already includes a similar idea, we cannot assume a TCB model for the wireless network. That is, the synchronous control network, which interconnects the components of the TCB, cannot be realized on a wireless LAN. The TCB model suggests realizing the control network as a small bandwidth dedicated network or by using the highest priority in the payload network. Both concepts cannot be used to realize a synchronous network on a wireless LAN. In a wireless network of mobile stations, whether or not a synchronous communication channel can be established between to stations is a dynamic property of the network. Yet, we adopted the idea to distinguish between the payload — the atomic multicast messages — and a small bandwidth synchronous channel, which is used to achieve agreement in case an atomic multicast message cannot be delivered at all members. In our system model, the synchronous channel does not provide a guaranteed synchronous service, but a conditional, fail-aware service, which can be implemented in a dynamically changing wireless network.

Finally, we shortly note, that there is a lot of further work on partial synchrony models (Dolev and Dwork 1987,Chandra and Toueg 1991,Chandra et al. 1992), which for the most part aims at adding just enough synchrony to an asynchronous system so that consensus can be solved, but not sufficient to provide a predictable timing behavior.

4.4.2 Real-Time Communication in Wireless LANs

Our approach is using a standard physical and MAC layer for wireless communication, namely, we base our protocols on the IEEE 802.11 Standard, which is commonly accepted and for which off-the-shelf hardware components are widely available. To achieve a controlled, contention-free medium access, we use the polling-based access method introduced under the name PCF in the original 802.11 Standard and updated in the supplement 802.11e. (Cirrus Logic,Plenge 1995,Ergen et al. 2002,van Hoesel et al. 2003) suggest own, non-standard physical or MAC layers for wireless real-time communication. We think that using standard, off-the-shelf components is advantageous because it does not require building, or using custom-built, hardware components. (Franz et al. 2001) basically employ the UTRA TDD Mode of UMTS (Haardt et al. 2000), but develop a special ad-hoc mode, which is not originally part of the technology, for their system.

(Sobrinho and Krishnakumar 1996b,Baldwin et al. 1999) consider using the DCF of the 802.11 Standard for the transmission of real-time messages. Both propose extensions to the DCF to achieve improved real-time performance. These extensions, however, only implement a best-effort approach to increase timeliness and do not guarantee a predictable timing behavior. (Sobrinho and Krishnakumar 1996a,Sobrinho and Krishnakumar 1996b) additionally proposes a negative acknowledgment scheme to tolerate message losses, which, however, is designed for point-to-point communications only.

(Cavalieri and Panno 1997) consider using an IEEE 802.11 Standard compliant wireless LAN as backbone for connecting field-busses. They propose using the CFP for periodic field bus traffic and concentrate on analyzing whether the CP can be used to transfer aperiodic field bus traffic. (Coutras et al. 2000) analyze the schedulability of constant bit rate and variable bit rate traffic during the PCF. (Sharon and Altman 2001) consider the effect of idle stations on the performance of the PCF. Our work focuses on two other aspects that the above works do not consider: Basing on the contention-free access provided by the polling mechanism, how can a reliable and timely transmission of multicast messages be achieved, and, with our shared resources scenario in mind, how can we enable stations to request admittance to the polling list in bounded time.

4.4.3 Reliable Multicast

In this sub-section we present related work w.r.t the reliable and timely transmission of multicast messages. In doing so, our main interest is whether and how the problem of achieving reliable and timely transmission in spite of varying loss rates is addressed. Gossip-based protocols, like (Sun and Sturman 2000,Kermarrec et al. 2003), are intended to provide probabilistic properties in large-scale systems and are hence out of the scope of this discussion.

The first point we are interested in is how message losses are tolerated. Regarding this point works can be distinguished as follows: (i) they do not consider message losses at all; (ii) they tolerate message losses using a static redundancy approach; or (iii) they tolerate message losses using a dynamic redundancy approach. Dynamic redundancy in the context of communication protocols means detecting message losses by way of acknowledgements and retransmitting affected messages. For our environment dynamic redundancy appears to be the most appropriate solution because the number of retransmissions is determined by the actual, not the worst-case number of messages losses and they allow detecting situation in which the number of retransmission were not sufficient. Both reasons owe their particular importance to the fact that tight worst-case bounds on the number of message losses cannot be assumed for wireless media. The efficiency of the acknowledgement mechanisms is a key factor in dynamic redundancy approaches. It will be important to consider, if the acknowledgement schemes of the discussed works are suited for the particular polling-based communication structure of the IEEE Standard.

The second point we are interested in is whether the protocols explicitly allow handling the inherent tradeoff between reliability and timeliness. We examine whether they provide the means to relax reliability requirements in order to achieve shorter delays. In the following sub-section, which deals with atomicity, we will then consider if the protocols provide agreement in case a message with a reduced number of retransmissions in not received by all group members. This point is important, since as long as agreement and total order are guaranteed, common views and hence consistent actions can be achieved even in case of multicasts being lost. For those protocols that combine ordering and reliability in a single protocol, we will explain as much of the ordering mechanism as is necessary to understand how reliability is achieved in the section at hand.

Reliable communication channels. There are protocols designed assuming that communication between any pair of connected stations is reliable; that is, communication channels are either assumed to be stable or exhibit crash failure semantics (Birman et al. 1991,Ezhilchelvan et al. 1995,Chockler et al. 1998). Obviously, these protocols do not deal with omission failures. They are designed to tackle other problems like agreement in case of sender crashes, ordering semantics, etc. (Cristian et al. 1985) achieve reliable communication through spatial redundancy, which ensures that the sub-graph consisting of correct processes and communication links is always connected. So, they do not address omission failures of the communication links too. (Kopetz and Grünsteidl 1993,Kopetz 1997) use spatial redundancy as well, but take message losses into account additionally.

Static time redundancy. Static redundancy can be used to achieve reliable communication if there is a known bound on the number of omission failures. For example, in TTP (Grünsteidl and Kopetz 1991,Kopetz and Grünsteidl 1993,Kopetz 1997) each message is transmitted a fixed number of times on physically redundant channels. Transmitting each message r+1 times allows tolerating up to r consecutive message losses. (Bar-Joseph et al. 2000) use forward error correction (FEC) techniques to tolerate up to r losses out of $k + r$ consecutive messages. For each sequence of k original messages, r redundant messages are constructed in such a way that the receiver can reconstruct the original messages from any subset of size k of the $k + r$ messages sent. The performance of static redundancy protocols is determined by the worst-case number of message losses. Since in a wireless network a very large worst-case bound must be assumed, which significantly exceeds the average number of losses, adopting this approach would result in a poor average performance of the protocol. Furthermore, both protocols do not allow choosing a number of transmissions that is smaller than required by the worst-case number of message losses.

Dynamic time redundancy. The xAMP (Verissimo et al. 1991,Rodrigues and Verissimo 1992) is based on a synchronous system model with a bounded number of message omission failures. It uses a positive acknowledgement scheme to detect message losses. Each receiver sends an individual positive acknowledgment for each message it receives. This means that the number of acknowledgement messages is proportional to the number of group members, which results in a significant overhead. Furthermore, this mechanism is not well suited to be used in the CFP, since each receiver would have to wait until the AP polls it before sending the acknowledgement message. Therefore, after each multicast, the AP would have to poll each station for the transmission of its acknowledgement message. Thus, in the best (fault-free) case, at least 2n+2 messages are needed for a single multicast with n intended recipients (the first polling message, the broadcast message itself, n polling messages for the acknowledgements, and the acknowledgements). The protocol does not ensure agreement if the number of retries is not sufficient to ensure that all stations receive the message. Therefore, if agreement is required, the retry limit must be set to the omission degree. In our architecture, this is not required, since the atomic multicast protocol achieves agreement even if a retry limit smaller than the omission degree is chosen.

There are many reliable multicast protocols developed for asynchronous systems that employ dynamic redundancy to tolerate an a priori unknown number of message losses and guarantee delivery of multicasts as long as sender an receiver are correct and connected. (Inoue et al. 1998) enhanced reliability of multicasts in wireless networks using a representative acknowledgement scheme to reduce the number of acknowledgements. They subdivide the group into subgroups, each of which is assigned a representative. The representative is responsible for sending positive or negative acknowledgements when being polled by the sender after a broadcast.

The protocols presented in (Peterson et al. 1989,Melliar-Smith et al. 1990,Amir et al. 1992) exploit the partial causal order (Lamport 1978) to detect message losses. Stations piggyback positive acknowledgements on their broadcasts. Together with a message all causal predecessors of that message are acknowledged. So, this scheme is a kind of a cumulative acknowledgement scheme. Stations are able to detect their missing a message by detecting gaps in the ordering graph; that is, when they receive a message, but have not yet received the causal predecessors of that message. In this case, they send negative acknowledgements to explicitly request the retransmission of the messages they missed. Since our protocols are not intended to provide causal ordered delivery, transferring and maintaining the necessary context information would introduce an unnecessary overhead.

There are several protocols that impose a logical ring structure on the multicast group (Chang and Maxemchuck 1984,Cristian and Mishra 1995,Jia et al. 1996,Mishra et al. 1997,Mishra et al. 2002). According to this structure, the group members take turns in assuming the role of a central sequencer (or token site). The sequencer is in charge of assigning global sequence number (a.k.a ordinals) to the multicasts messages. Although primarily introduced to order multicasts and to distribute the load associated with being the sequencer equally among the group members, all protocols exploit the structure to realize an efficient acknowledgment mechanism. The following main ideas contribute to this objective: (i) rotating the right to order messages (i.e. the token) is used as an implicit acknowledgment mechanism. By assuming the role of the sequencer, a station acknowledges multicasts that have been ordered so far. (ii) Global sequence numbers are being used to detect message losses. If the difference between the sequence numbers of two consecutively received multicasts is greater than 1, the receiving station knows that it must have lost some message and it knows the global sequence numbers of these messages also. (iii) Global sequence numbers can be used as cumulative acknowledgments. By announcing the highest in order global sequence number it has received, a station is able to acknowledge all messages up to that sequence number. The different protocols add specific optimizations to these basic mechanisms. For example, in the RMP (Jia et al. 1996), stations include a so-called safe parameter in their messages to support a fast assessment of the stability of a message (the message has been received by all group members). In the protocol suggested in (Mishra et al. 2002), the sequencer sends a list with selective acknowledgements for all messages, which are not deemed stable at that moment. For each such message a bit vector *ack* is included where $ack[i] = true$ if the ith member on the ring has acknowledged the message. This idea is similar to the way stations transmit acknowledgments in our protocol. Protocols with a rotating central sequencer have turned out to exhibit a good performance in settings where the ring is stable most of the time. Their drawback is that a ring reformation is required when the (implicit) token is lost or the sequencer crashes. They are not well suited for a wireless environment since message losses are frequent and may trigger ring reformation if the token is affected. Furthermore, dynamic changes in the topology due to locomotion of the systems may result in two successive stations in the ring no longer being connected, which would trigger a ring reformation too. Thus, ring reformations are quite likely and impair the efficiency and the predictability of the protocol.

(Kaashoek and Tanenbaum 1991) also adopt a central sequencer approach, but in their protocol the role is fixed to a certain station. A station that wants to multicast a message sends it to the sequencer, which associates a global sequence number with that message and multicasts it to the group. We adopted that communication structure with the AP acting as the central sequencer since it fits very well to the structure of the underlying network:

- In a BSS (cell) of an 802.11 Standard infrastructure network all frames have to be routed through the AP anyway.
- Routing frames through the AP ensures that each stations in the BSS are within the range of the broadcaster
- Since the centralized communication structure of the PCF already implies the assumption of a stable central station, we can best profit from that fact by making this station the sequencer also. In particular, this makes dealing with station crashes easier and allows achieving a more predictable timing behavior in this case.

In this structure, two kinds of message losses have to be considered: (a) request messages transmitted from the sending station to the sequencer, and (b) broadcast message from the sequencer to the group. To protocol uses implicit positive acknowledgements to detect the first kind of message losses. The sequencer acknowledges reception of a message by broadcasting it (message identifies allow recognizing the message). To deal with the second kind of message loss, negative as well as positive acknowledgments are employed. A station detects that it lost some multicast message if there is a gap between the global sequence numbers of the messages it receives (as described above). In this case, it sends an explicit negative acknowledgement for the missed messages to the sequencer to request a copy of those messages. Since in this approach the sequencer must store messages it broadcast for the purpose of later retransmission, there must be some means for the sequencer to learn that it need no longer store a message in its buffer. To this end, the protocol uses cumulative positive acknowledgments. A station includes in each request message it sends the highest in order sequence number it received so far. Similar to the safe parameter mentioned above, the sequencer determines the minimum of all sg_i, where sg_i is the last acknowledged sequence number the sequencer received from station s_i. Each message with a sequence number not greater than that minimum can be safely purged from the sequencer's buffer. While we adopt their approach to dealing with the first kind of message losses (a), we use another approach for the second kind (b). Instead of explicit negative acknowledgment message we use piggybacking since each explicit message requires polling and adds to the overhead of the protocol. Moreover, we use selective acknowledgments instead of cumulative ones since the former are better suited for environments with a large number of message losses

The protocols described above, which deal with unbounded message omission failures, are designed for asynchronous environments and implement eventual termination semantics; that is, they ensure that messages sent by a correct station are eventually delivered by each correct intended recipient as long as both are connected. Since timeliness is not an issue here, none of the protocols provides an explicit parameter to bound the number retransmission in order to improve timeliness. There typically is an implicit bound in the sense that a station that continuously fails to acknowledge a message will sooner or later be considered as having crashed or being disconnected from the group and it will be removed from the membership. But, even if this means that the message delays will not grow arbitrarily due to message losses, there is no way to bound message delays independent of the notion of group membership.

RTCAST (Abdelzaher et al. 1996) is based on a synchronous model with a finite but not explicitly bounded number of omission failures. Retransmissions are not handled in the multicast protocol. Instead, it assumes that a bounded number of retransmissions may be performed on a lower layer to reduce the probability of message losses on the multicast layer. The basic idea to deal with message losses is to maintain agreement among the group members. To achieve this, each station detecting that it missed a message takes itself out of the group. So, this can be considered as a model with reliable links and stations observing pseudo receive omission failures. Similar to the protocols discussed above, the drawback of this approach in the context of our environment is that there is not distinction between not receiving a message and being excluded from the group. So, either the number of retransmission must be chosen quite high, even if messages do not have high reliability requirements, or the station will be excluded from the group frequently. The protocol therefore does not allow deciding the reliability/timeliness trade-off for messages independent of deciding when a station is considered to be disconnected.

4.4.4 Atomic Multicast

Regarding total order, the special communication structure underlying our protocols, where a stable central station coordinates access to the medium, strongly suggests using a central coordinator, called sequencer, for ordering messages. Using a fixed sequencer was already considered in the protocol family proposed by (Chang and Maxemchuck 1984) and was adopted and optimized by (Kaashoek and Tanenbaum 1991). (Rodrigues et al. 1995) also suggest using a central sequencer approach within a wireless cell. There is quite a number of other approaches used to achieve total ordering; for example, approaches in which the right to be the sequencer, called the token, circulates in a logical ring structure (Chang and Maxemchuck 1984,Amir et al. 1995,Cristian and Mishra 1995 #691,Abdelzaher et al. 1996,Jia et al. 1996,Mishra et al. 1997,Mishra et al. 2002) or approaches (Peterson et al. 1989,Melliar-Smith et al. 1990,Amir et al. 1992) in which constructing a total order is based on the causal order (Lamport 1978) which constitutes a partial order on the messages. We will not further discuss these approach here, since, as explained above, the central sequencer approach fits best to the structure of the underlying network; we refer the interested reader to (Defago et al. 2000) for a comprehensive survey. What we would like to mention is that our ordering approach exhibits some similarity with the two-phase approach suggested in (Rodrigues and Verissimo 1992). In this approach, the position of a message in the delivery order is fixed only after all stations have acknowledged the message. This is similar to the way our protocol works. This approach has the advantage that messages that have been acknowledged by all group members need not wait on messages that are still lacking acknowledgements.

All of the reliable multicast protocols presented in the previous sub-section have a semantics requiring that each correct intended recipient receives a multicast as long as the sender is correct and both remain connected. So, as long as the sender remains correct, all correct and connected recipients will deliver the message and will therefore be in agreement. These protocols therefore are mainly concerned with achieving agreement in case of sender crashes. The basic idea to tackle this problem is to make each station that delivers a message also responsible for retransmitting that message in addition to originating station. For schemes with negative acknowledgments, like (Peterson et al. 1989,Melliar-Smith et al. 1990,Amir et al. 1992,Jia et al. 1996) for example, this means that each receiver is prepared to answer retransmission requests. For schemes with positive acknowledgments and automatic retransmission, like (Rodrigues and Verissimo 1992), this means that each receiver also sets a retransmission timeout and waits for acknowledgements. In (Cristian et al. 1985), a more static approach is used, where each receiver immediately forwards the message to all its neighbors. A potential additional problem is that after some stations delivered the message, not only the originator but also the receivers of the message crash. In such a situation additional measures are necessary to ensure uniform agreement; that is, to ensure that all correct stations deliver a message, if any station, even a faulty one, delivered the message. The basic approach here is to deliver only stable messages. A message is said to be stable if all group members have received it. Thus, even if the originator of the message as well as some stations that delivered it will crash, still, all correct stations have received the message and will eventually deliver it, so that uniform agreement is achieved. To determine whether a message is stable the acknowledgment schemes described above are being used. For example, in a token-based protocol a message is known to be stable once the token has traveled one round through the ring after the message was broadcast (Chang and Maxemchuck 1984,Amir et al. 1995,Cristian and Mishra 1995,Jia et al. 1996,Mishra et al. 1997,Mishra et al. 2002). In our protocols problems are different. First, sender crashes do not impose an agreement problem. If the originator of a message crashes while trying to send the message to the AP, the AP has either received the message by that time or not. In both cases, agreement among the station is ensured. The AP, on the other hand, is assumed to be stable, and therefore does not crash. This also means that stability of messages is not required to provide uniform agreement in spite of receiver crashes. Receivers are not in charge of retransmitting messages anyhow; rather does the AP ensure that all correct group members will receive the message. On the other hand, the protocols described above do not consider the case in which only part of the correct group members received a message because the correct sender stopped retransmitting it. There is a similarity between our approach and the notion of stability, since in both approaches stations deliver a message only after it has been determined that all correct members are able to deliver it. Most similar is the protocol in (Kaashoek and Tanenbaum 1991) since in this protocol, like in ours, a central station is responsible to establish stability and then inform the group members by way of an accept message. But remember that (a) in (Kaashoek and Tanenbaum 1991) stability is used to tolerate crashes of the sequencer, (b) they use different mechanisms to transmit the acknowledgements and the accept message, and (c) there is no idea of rejecting a message.

(Kopetz and Grünsteidl 1993) adopt another approach. They use a static TDMA approach where each node has a fixed sending slot. Whenever a node does not receive a message during the sending slot of another one, it removes this node from its membership vector. Each node transmits it membership vector together with each message it sends. Nodes receiving a message with a membership vector different from theirs reject the message and remove the sending node from their membership list. A node that rejects messages from a majority of the group members puts itself into an inactive mode. This ensures that only nodes which are in agreement can communicate with each other and that each node which is not in agreement with the majority removes itself from the group. Similarly, in (Abdelzaher et al. 1996), each station that misses a message removes itself from the group to ensure agreement among the remaining group members. In both approaches a message not being received by part of the group leads to changes in the membership. So, this corresponds to a model where communication is reliable and omissions are either attributed to the sending or the receiving node.

Our work on agreement on multicast messages that are not received by all station, is comparable to the system presented by (Almeida and Verissimo 1995,Almeida and Verissimo 1996). They also propose an approach that allows group members to agree on the failure of a message. While we are considering omission failures, their approach is more tailored towards timing failures. In their system, the stations are connected through a small bandwidth synchronous channel, which allows reliable and timely transmission of small amounts of information. By periodically exchanging status information over the synchronous channel, stations timely detect whether all group members received a multicast message in time or not. In the latter case, all stations can agree not to deliver that message. Generally, this concept also covers cases, in which excessive delays are caused by an excessive number of retransmissions, so that in this case station can agree not to deliver the message. However, the underlying multicast layer does not stop transmitting the message in this case, so that bandwidth may be wasted. Furthermore, there are obviously differences in realizing the synchronous channel. For one thing, we have to provide reliable and timely transmission on the same wireless medium used by the payload. To keep the overhead small we use piggybacking for the messages transferred in the synchronous channel so that no extra messages are required. Furthermore, stations are not statically connected to the synchronous channel, but the connection between the AP and a station may crash when the station moves out of reach.

4.4.5 Membership

There has been quite some work on the specification of membership services. (Hiltunen and Schlichting 1995a,Hiltunen and Schlichting 1995b,Chockler et al. 2001) provide comprehensive overviews based on formal models. Our definitions are most closely related to those presented in (Cristian 1996). This is because our system model is similar to Cristian's timed asynchronous system model as was point out above. In particular, we use unconditional safety and conditional timeliness properties to define the services and adopt the concept of fail-aware. The conditions are based on a predicates describing the "amount of synchronism" in the system. Whereas the properties in (Cristian 1996) are conditioned on the stability of the system our properties are conditioned on the quality of communication between individual stations an the AP. In (Killijian et al. 2001) the notion of proximity groups is suggested, which are defined by specifying an area. In our approach it is up to the application (or a layer between the application and the membership service) to call the join procedure when the mobile system enters a certain area. So, the semantics of how the group is defined remains separated from the communication service.

Membership protocols for synchronous systems that guarantee a bounded join delay use static bandwidth allocation (Ezhilchelvan and de Lemos 1990,Kopetz 1997). Each station has a fixed sending slot, which remains reserved for that station even if it is removed from the membership. Hence, it can use its sending slot to rejoin the group after a restart. (Cristian 1991) also assumes that each joining station has sufficient bandwidth to send a synchronous atomic multicast when a membership server sends an invitation message. Our requirement is that a priori unknown stations may join the group at any time and stations may leave the group forever so that a fixed allocation of bandwidth cannot be used. The protocol presented in (Abdelzaher et al. 1996) allocates bandwidth dynamically. However, the joining stations uncoordinatedly compete for a certain amount of bandwidth allocated to all joining stations. Therefore, bandwidth allocation to joining stations is not predictable and a bounded join delay is not guaranteed. This is similar to using the contention period of the IEEE 802.11 Standard to request addition to the polling list. The membership protocol in (Mishra et al. 1998,Mishra et al. 2002) was designed for the timed asynchronous system model and provides conditional timeliness guarantees (Cristian 1996). Since the protocol is required to achieve a bounded join delay only if the joining station can communicate with the group members in a timely manner, no bandwidth allocation is necessary to ensure that the joining system is actually able to transmit messages in time. In fact, the model was devised to match typical networks that do not use bandwidth allocation. Obviously, the same is true for all those protocols not providing timeliness at all.

Regarding the dissemination of membership change information our protocol is most similar to those approaches basing on an underlying atomic multicast protocol (Cristian 1991,Moser et al. 1994,Abdelzaher et al. 1996). These approaches exploit the properties of the underlying atomic multicast protocol to achieve agreement and total order for membership changes. Thus, the mechanisms implemented to achieve these properties for the atomic multicast protocol are reused to realize the services of the membership protocol. Since the atomic multicast protocol establishes a total order between the application messages and the membership change messages virtual synchrony is achieved also. Furthermore, this approach allows transmission of atomic multicasts to continue while membership changes are in progress. This, however, requires that station failures cannot block the total ordering protocol. Our protocol ensures this through the failure detection provided by the polling protocol. Our approach is similar to the one presented in (Abdelzaher et al. 1996) where a joining station transmit a request to some group members, which sends a membership change message on behalf of that station. In our protocol, stations send a request message to the AP, which starts multicasting the membership on behalf of that station. To improve the performance, both the request message of the station and the multicasts sent by the AP may contain an application message from the joining station. If this is the case, it means that the membership change information is piggybacked on the multicast message of the joining station. Furthermore, no explicit membership change message is multicast when a station is removed from the membership; rather, the AP uses the synchronous channel to disseminate such change information.

In (Ezhilchelvan and de Lemos 1990,Grünsteidl and Kopetz 1991,Kopetz and Grünsteidl 1993) the synchronous TDMA-based communication structure is used to maintain a consistent and up-to-date view on the membership. In this communication structure, each station must regularly transmit a message during its sending slot. Thus, stations are able to detect the failure of a station if they do not receive the expected message during the sending slot of this station. All correct stations detect the failure of a sending station during the same slot and hence at the same position in the sequence of messages. Since send omission failures are assumed to be perceived consistently all correct station will agree on the failure of the station. If a station is subject to a receive omission it may wrongly suspect other stations to have failed and disagree with the correct station about the membership. Both protocols provide mechanism to ensure that such stations will be detected and removed from the membership. (For (Grünsteidl and Kopetz 1991,Kopetz and Grünsteidl 1993), this mechanism was sketched in the preceding sub-section.) Since this approach is based on the assumption of reliable communication and on the consistent perception of send omission failures we cannot adopt it in our protocol.

Other protocols, like (Chang and Maxemchuck 1984,Amir et al. 1995,Cristian and Mishra 1995,Jia et al. 1996,Mishra et al. 1997,Mishra et al. 2002), which are based on logical ring structures and are not based on an underlying atomic multicast protocol, typically require complex reformation protocols. Even if a bounded termination time can be ensure for such a protocol (Amir et al. 1995,Mishra et al. 2002), they interfere with transmission of the atomic multicasts and hence add to the delay of the messages.

4.4.6 Communication Paradigms for Cooperative Systems

The synchronous[4] communication semantics of the RPC (Bakre and Badrinath 1995,Kümmel et al. 1996) used in the client-server paradigm of contemporary middleware (Microsoft Corporation 1996,Object Management Group 2002) is not well suited to support cooperation in groups of mobile systems. (Mock 2003) provided a formal model to specify the real-time requirements pertaining to the coordination of autonomous systems and showed in this model that the client-server paradigm does not match the requirements imposed by the kind of coordination in groups of mobile systems we consider in this thesis. In particular, he pointed out that the point-to-point semantics and the distinction between clients and servers inherent in this paradigm can hardly be mapped to the cooperation in groups of systems. Point-to-point communication does not support the mobile systems in coordinating their worldviews or their actions. Publisher/Subscriber paradigms (Oki et al. 1993,Rajkumar et al. 1995,Piaggio et al. 1999,Piaggio and Sgorbissa 2000) provide many-to-many communication and are therefore better suited. Object-oriented middleware has been extended to support this paradigm above the underlying client-server paradigm (Harrison et al. 1997). Publisher/Subscriber paradigms foster scalability, and their asynchronous, event-based communication semantics allows preserving the local autonomy of control. Yet, without providing ordering and agreement semantics for the delivered events, they do not directly support the cooperating systems in achieving coordination; that is, in fulfilling the consistency constraints that the cooperative application imposes on their actions. Furthermore, tight coordination in local groups typically requires explicit knowledge of the group of cooperating mobile systems — consider for example the shared spatial resources scenario. Therefore, our approach is to build event-based communication services on top of local group communication services. (Verissimo et al. 2003), for example, suggest a similar approach to support the real-time and consensus requirements of tightly cooperating groups. The underlying group communication services provide a timely and reliable communication as well as ordering and agreement semantics for the transmission of events. On top of them, event-based communications services are provided in the CADI layer (see Chapter 2). Their particular feature is the provision of common views, which allow resolving the autonomy/cooperation tradeoff as explained in chapters 1 and 2. For tight cooperation in local groups, the Event Service provides common views on the global system state; so, in fact, it provides a combined event- and state-based semantics, which is particularly suited for dynamically changing groups where newly joining systems may not have sufficient context information to interpret pure event information correctly

[4] Here, "synchronous" means that communication implies synchronization between the communicating entities; it does not refer to the timeliness of the communication as above.

5 Task Scheduling for Mobile Cooperative Applications

In mobile cooperative applications, all tasks controlling a mobile system's interactions with its physical environment must be performed reliably and in real-time. This comprises such tasks as avoiding collisions with static and moving obstacles, finding and following a path towards a goal, tracking objects, etc. These real-time and reliability requirements are not limited to the tasks directly controlling the motion of the mobile system, but apply to the sensor data processing tasks as well because in a dynamically changing environment the control tasks depend on online information about the system's environment.

The main problem in meeting these requirements lies in the hardly predictable execution times of the application tasks. In our intended application domain, most applications include tasks, which exhibit environment-dependent and hence widely varying execution times. Sensor-data processing tasks, for example, which form an essential part of all mobile applications, particularly exhibit this property. Using the prototype of the distributed sensor fusion as a concrete instance, we analyzed this problem. It turned out that both the size of the input and its content impact the execution times of the sensor processing tasks so that they are varying widely in a dynamically changing environment. For tasks with this characteristic, conventional approaches to real-time scheduling, which rely on known worst-case execution times (WCETs), cannot be adopted. Assuming known WCETs, if they can be determined at all, would result in a poor CPU utilization and only a very small task set being accepted for execution.

We apply the TAFT (time-aware fault-tolerant) scheduling concept to achieve a timely predictable execution of tasks with hardly predictable execution times (Nett and Gergeleit 1997,Nett et al. 1997,Gergeleit 2001,Becker et al. 2003). TAFT uses realistic expected-case execution times (ECETs) instead of WCETs to specify the resource demands of the tasks. This implies that a task instance may need more than the specified resources, which we refer to as a resource fault of that instance. As a consequence, it possibly cannot be completed by its deadline. In such cases, TAFT ensures that the task instance is aborted and an exception handling is performed by the task's deadline, so as to preserve a timely predictable behavior. Furthermore, it ensures that the fault does not propagate and affect other task instances. With timeliness, exception handling, and fault isolation ensured by the scheduler, application-inherent redundancy is exploited to tolerate the abortion of task instances.

Adopting TAFT for our application domain, we have to consider what kinds of application-inherent redundancy can be exploited there. To this end, we used the distributed sensor fusion scenario as an example for our investigation. Although it surely depends on the specific application at hand how redundancy can be exploited to tolerate task abortions, we believe that the kinds of application-inherent redundancy we found are typically present in cooperative mobile applications or can be furnished by a careful design. We will show how the different kinds of application-inherent redundancy can be used in combination with TAFT to achieve a reliable execution of the application. In doing so, we use the distributed sensor fusion scenario as an example to illustrate the suggested solutions. Accordingly, we use the prototype of the scenario to show the viability of this approach (see Chapter 6).

The second point that needs to be resolved when adopting TAFT for the task execution service of the middleware is developing a scheduling algorithm that implements the TAFT concept and supports a task model suitable for our application domain. In (Becker and Gergeleit 2001,Becker et al. 2003,Gergeleit et al. 2003), a scheduling algorithm was presented for a task model with sets of independent periodic tasks. For the intended application context we have to extend this model to accommodate the following two requirements also:

1. To schedule the CPU demands of the communication protocols of the middleware, the model must encompass aperiodic requests in addition to the periodic tasks.

2. To allow using the results of one task pair as input for another one, the model must allow for precedence constraints between the task pairs.

This chapter is structured as follows. In Sections 5.1 through 5.3 we explain how a timely predictable and reliable execution of the application tasks is achieved in spite of the unpredictable execution times they exhibit. Sections 5.1 illustrates the problem with measurements from the prototypical implementation of the distributed sensor fusion scenario. The measurements show that the execution times of the tasks are environment dependent and widely varying. Section 5.2 then describes the TAFT concept. It makes clear how TAFT achieves a predictable timing behavior for tasks with hardly predictable execution times. Furthermore, describing TAFT in sufficient detail is an important prerequisite to understand the extensions of the task model and the scheduling algorithm we present in the remainder of the chapter. In Section 5.3, we examine what kinds of application-inherent redundancy can be found in cooperative mobile applications and how they can be exploited in conjunction with TAFT to tolerate task abortions. Afterwards, we turn towards developing a scheduling algorithm that implements TAFT for a task model matching for our application context. This is done in two steps. First, in Section 5.4, we develop a scheduling algorithm, called TAFT-IPE, that implements TAFT and supports the execution on aperiodic requests. Second, we extend this scheduling algorithm to allow for precedence constraints between periodic task pairs (Section 5.5).

5.1 Environment-Dependent Execution Times in the Distributed Sensor Fusion

To analyze the problem of hardly predictable execution times in a real application context, we measured the execution times of the filter and fusion modules in the prototype of the distributed sensor fusion. The results we present in this section have been measured on an AMD Athlon 700MHz CPU with 128MB memory. They show that the execution times depend on the following two aspects.

The first aspect that impacts the execution times is the amount of input of the module. For example, Figure 5-1 depicts the execution times of the object filter plotted against the number of input structures (arcs and edges detected by the contour filter). Despite minor variations, a clear trend of increasing execution times can be observed for an increasing number of input structures.

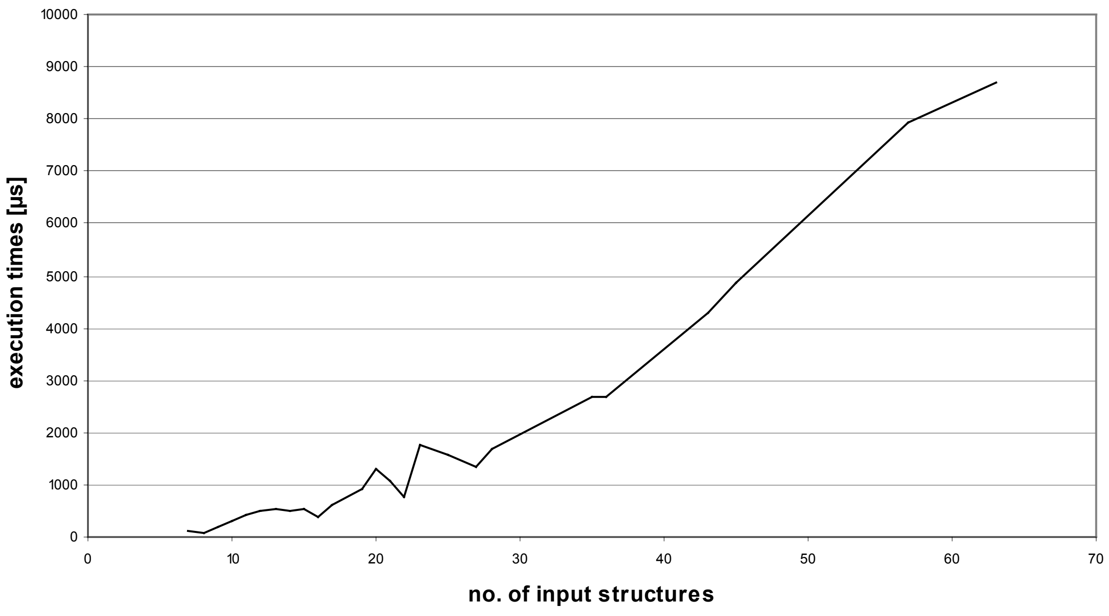


Figure 5-1. Execution times of the object filter against input size

The second aspect impacting the execution times is the content of the input. Aspects such as the shape of objects as well as their number and distance are of importance. A good example is the arc filter (In what follows we go on using the arc filter as an example because it has by far the largest execution times of all modules). Even though it has a constant input size, 361 points always, its execution times are varying widely depending on the environment observed. Figure 5-2 shows how the execution times of the arc filter vary over a sequence of scans observed by a laser-scanner moving through a scenario with three rectangular objects and a ball. The measured execution times have a range of 136.62 ms, which is large as compared to a mean of 21.78 ms.

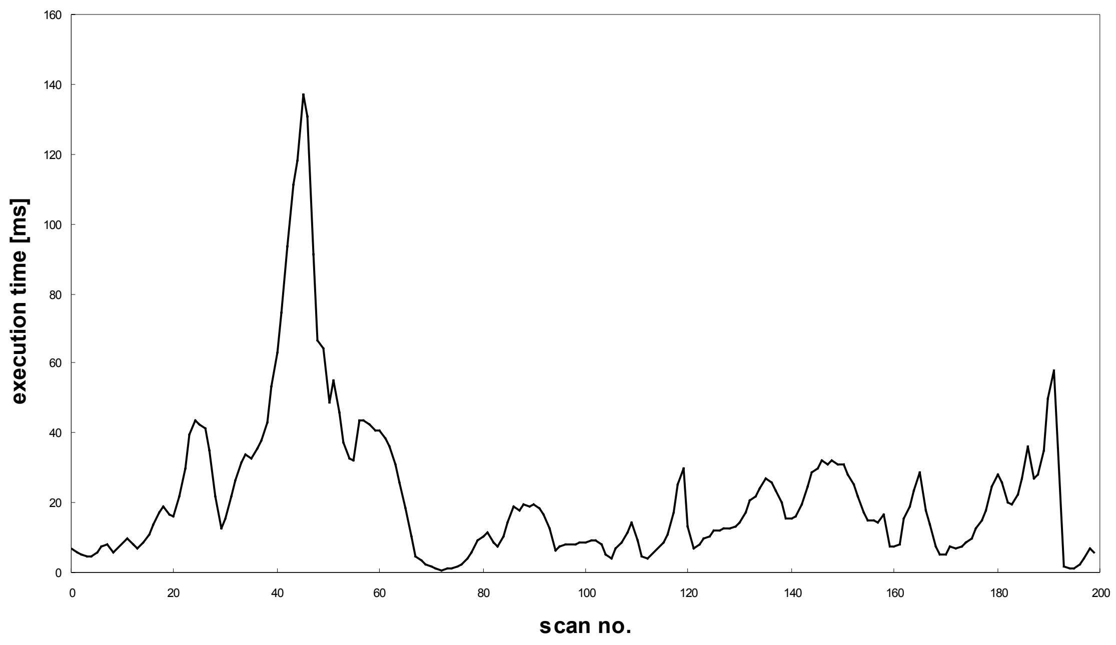


Figure 5-2. Execution times of the arc filter

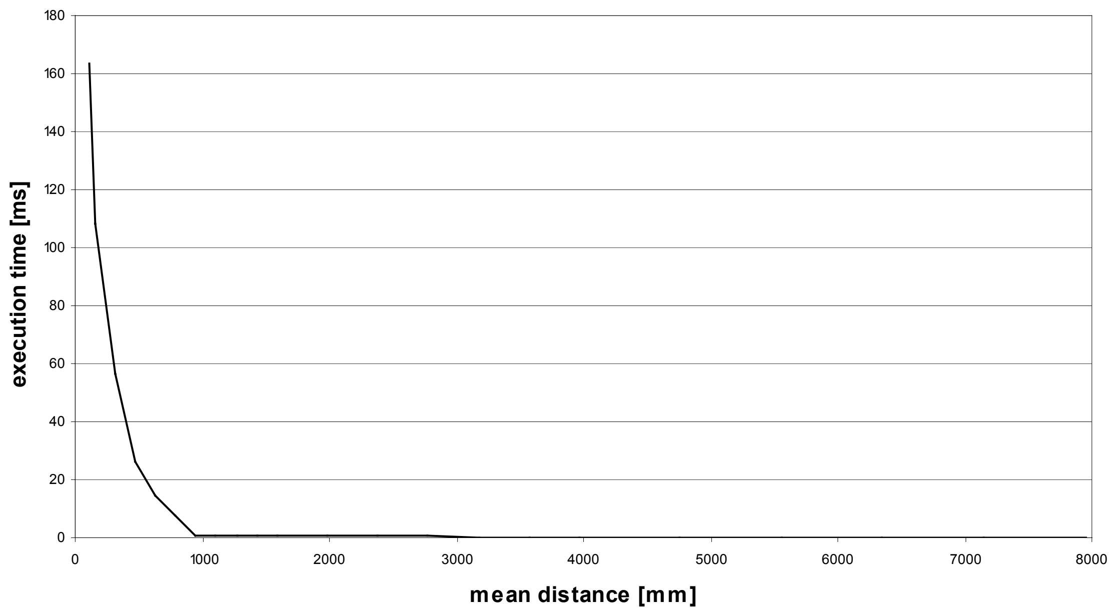


Figure 5-3. Execution times of the arc filter against mean distance

Figure 5-3 makes clear that one parameter of the input's content that heavily influences the execution times of the arc filter is the mean distance between the points observed and the scanner. This can be attributed to the way this filter detects arcs: It constructs circles with a given diameter for pairs of points with a certain distance. For each such circle, it determines the points located in a certain environment along the circle. If there are enough such points and if the evaluation of their position w.r.t the circle exceeds a given threshold, they are assumed to belong to an arc. As an optimization, the algorithm does not consider all points for each circle, but only those lying within a certain sector of the scanner's field of view. This sector is lying between two rays starting at the scanner's origin and being tangent to the constructed circle. Hence, the nearer the circle, the wider the sector, and the more points have to be considered.

In addition to the causes of the variations in the execution times, their durations must be distinguished. Figure 5-2 shows that the arc filter usually exhibits short-term peaks in its execution times. For example, execution times of more than 60ms only occur in a single, small peak (10 scans only). This shows that situations with exceedingly long execution times frequently have a short duration. Nevertheless, as execution times are environment dependent, we must assume that long execution times may also be persistent for extended intervals of time. For example, consider a situation in which a robot is forced to keep near to some objects in its environment over an extended period of time.

The results corroborate our point that the execution service must cope with execution times that vary significantly as the mobile systems are moving in a dynamically changing environment. Execution times not only depend on the amount of input but also on its contents. It turned out, for instance, that the mean distance between the laser scanner and the shapes it observes is an important parameter for the contour filter's execution time. In such a setting, it is a hard task to determine WCETs for modules, for they would have to be based on assumed worst-case environments, which are hardly ever observed. Furthermore, our measurements indicate, that particularly long execution times frequently prevail for short intervals of times only. Hence, allocating these excessively high execution times, which are still not WCETs, would mean reserving resources that are hardly ever needed.

5.2 TAFT

As the measurements in the preceding section illustrate, many tasks in cooperative mobile applications exhibit environment-dependent and widely varying execution times. Worst-case execution times (WCETs) are not suitable to specify the resource demands of such tasks. If they can be determined at all, WCETs are likely to be far beyond any realistic value. Therefore, the scheduled resources would by far exceed the actual demands of the tasks. Thus,

- Only few tasks could be scheduled, as compared to what would be possible with more realistic demands specifications;

- The real resource usage of the accepted tasks would require only a small fraction of the available CPU resources such that a poor CPU utilization would result.

For these reasons, we aim at providing a task execution service that does not require the specification of worst-case execution times, but achieves a predictable timing behavior of the tasks nevertheless. To this end, we adopt the TAFT scheduling concept, which has been developed with these requirements in mind (Nett and Gergeleit 1997,Nett et al. 1997,Becker and Gergeleit 2001,Becker et al. 2001,Gergeleit 2001,Becker et al. 2003,Gergeleit et al. 2003,Becker et al. to appear). Though working with more realistic

ECETs instead of WCETs, it achieves a timely predictable task execution. In this section we explain the TAFT scheduling concept.

The very basic concept underlying TAFT is the *task pair (TP)*. A task pair consisting of a *main part (MP)* and an *exception part (EP)* represents each task. The main part provides the intended functionality of the task, whereas the exception part includes exception-handling code that is executed when the main part cannot be completed before the task pair's deadline. The CPU demand of the main part is specified through a so-called *expected-case execution time (ECET)*. ECETs are determined by sampling actual task execution times, which can be performed off-line (Schemmer and Nett 2003a) or through online monitoring (Gergeleit 2001). The ECET models the resource demand of the main part more realistically than a WCET. In general, it is significantly smaller than the latter. The exception part, on the other hand, will usually consist of only very few emergency actions so that a WCET can be used to specify the CPU demand of this part.

As the specified resource demand of the main part does not represent the worst case, the instances of the main part possibly require more than the specified resources to be completed. We will refer to this event as a *resource fault* in what follows. If not handled, a resource fault can have the following two severe consequences:

- There may be not enough resources available to complete the main part by its deadline, in which case it would terminate at some time after the deadline. At which point of time the main part actually terminates is not predictable in this case, for it very much depends on the main part's unpredictable execution time. Thus, the task pair would exhibit an unpredictable timing behavior. Considering a task pair executing on some CPU as a software component and referring to the failure modes presented in Sub-Section 4.1.1, we refer to this situation as timing failure of the task pair.

  Figure 5-4 shows an example for a resource fault resulting in a timing failure. In this figure upward arrow denote the release times and deadlines of the two periodic task pairs τ_1 and τ_2. Boxes about the time axis of a task indicate that this task is running. If the boxes have a white filling the task is still within the specified ECET, while a light gray filling means that the task exceeds its specified demand. We will use the same kind of representation also latter. In the depicted situation, task τ_2 needs three time units more than allocated. As a result it is completed only after its deadline.

- If a main part occupies the CPU longer than its specified execution time, these CPU times may be missing for the execution of other tasks. Thus, the resource fault of one task may cause other tasks to exhibit timing failures. This is called fault propagation, or *domino effect* in the context of real-time scheduling. It may even happen that the faulty task itself is completed before its deadline, but causes timing failures in a number of other tasks. This kind of implicit dependency between otherwise independent task makes system behavior particularly complex, unpredictable, and hard to analyze.

  Figure 5-5 shows an example. In the depicted situation, task τ_1 is subject to a resource fault, yet still completes before its deadline. However, due to τ_1's consuming more than the allocated resources, there are not enough resources left to complete τ_2 in time. So, τ_2 is subject to a timing failure caused by the resource fault of τ_1.

TAFT avoids both these problems. It keeps the main part's resource faults from causing timing failures of the task pair and from propagating to other task pairs. We call the latter property *fault containment* since the resource fault of some task pair does not lead to another task pair's observing a timing failure or getting less than the specified resources. How TAFT achieves this is considered in what follows.

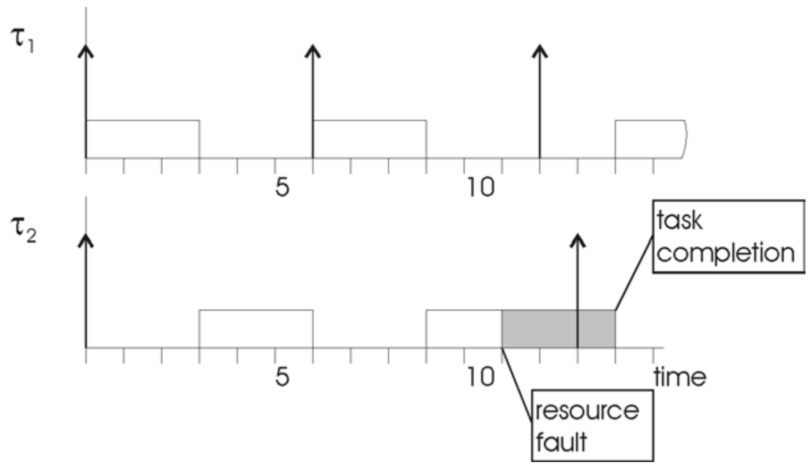


Figure 5-4. Resource fault resulting in a timing failure of the same task

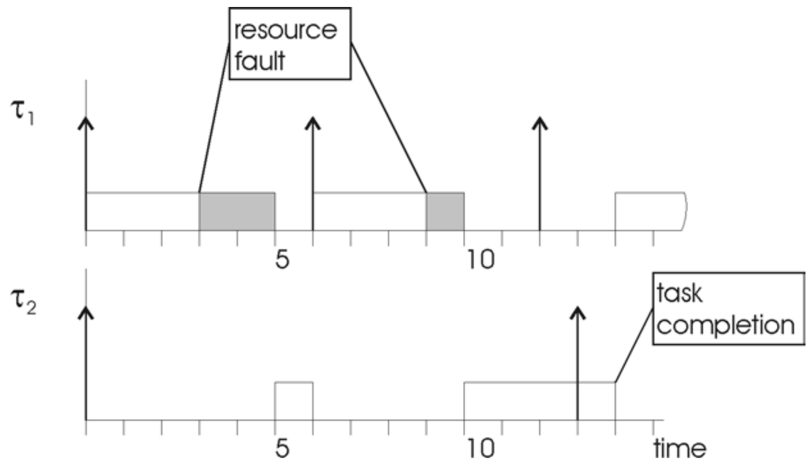


Figure 5-5. Resource fault resulting in a timing failure of another task

TAFT prevents timing failures in the presence of resource faults. TAFT monitors the execution of the main parts and aborts them if it they are about to miss their deadline. So, a main part is never executed after its deadline. Simply aborting the main part, however, will frequently not suffice, at least for the following two reasons:

- Tasks frequently have side effects that need to be reset when the task is aborted. Side effects may be internal to the control system; for example, if the task performs temporary updates to internal state variables. Side effects may affect the control system's environment too; for example, consider a task that starts an actuator, such as a motor. Before the task is aborted, the motor must be stopped to leave the controlled system in a safe state.

- Tasks may have computed valuable results at the time at which they are aborted. Simply aborting the task may lead to all these result being lost.

To avoid such drawbacks, TAFT executes the exception part when it aborts the main part. It ensures that the exception part is completed by the task pair's deadline. This is possible because the WCET of the exception part is assumed to be known. This assumption is justified since the exception part only includes a few deterministic actions to avoid the above-mentioned problems. For example, it may stop a motor, reset some internal variables, or make available intermediate results.

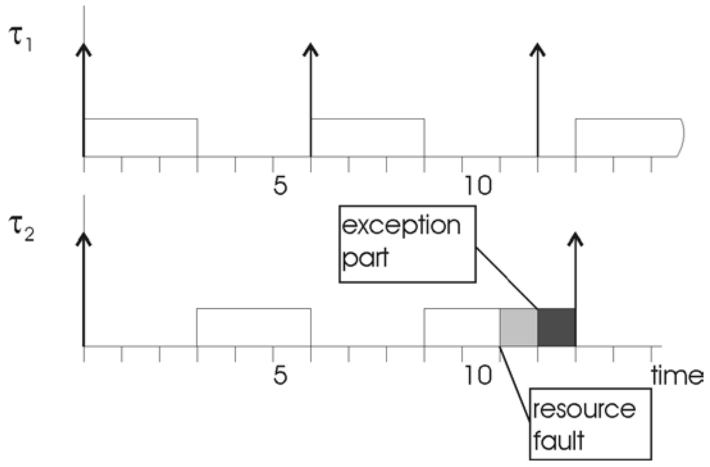


Figure 5-6. TAFT avoids timing failure of faulty task pair.

Figure 5-6 revisits the scenario depicted in Figure 5-4; this time, however, with TAFT being used. At time 12, TAFT detects that it cannot complete the main part of τ_2 by the deadline. So, it triggers the exception part of τ_2 (depicted in dark gray). As it does this sufficiently early, the exception part is completed by τ_2's deadline.

To avoid propagation of resource faults, TAFT schedules faulty main parts on a priority level lower level than that of the non-faulty ones — which still have not consumed their specified resources — and of the exception parts. Hence, faulty main parts can never keep non-faulty main parts from getting their specified resources. Note that this also means that a task pair is not necessarily aborted when it becomes faulty. It remains in the pending queue, but on a lower priority level. So, the scheduler may still assign resources to it and complete it by its deadline. If this is the case, the resource fault does not cause a failure of the task pair, which means it is tolerated.

Figure 5-7 revisits the situation depicted in Figure 5-5. This time, however, TAFT avoids the propagation of the resource fault. At time 3, when τ_1 becomes faulty, TAFT sets it to the lowest priority level. Hence, τ_2 gets access to the CPU first and the faulty main part of τ_1 cannot steal resource allocated to τ_1. When TAFT detects that it cannot complete τ_1 in time it executes the exception part. Similar at time 9, when TAFT again sets the faulty main part of τ_1 on the lowest priority level. Again, τ_1 gets the CPU and this time runs to completion. Under the control of TAFT, the faulty main parts in the instances of τ_1 do not cause a timing failure of τ_2.

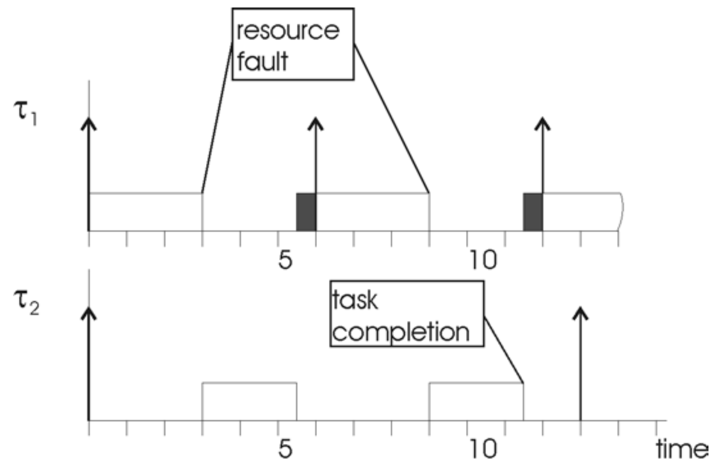


Figure 5-7. TAFT avoids propagation of resource faults

After having explained the basic concepts of TAFT, we now describe more formally the underlying task model. This serves as a foundation for the presentation of formal results in the remainder of this chapter. A task pair τ_i, $i \geq 1$, consists of two parts, the main part MP_i and the exception part EP_i. It is characterized by the following timing parameters: C_i is the expected-case execution time of MP_i, and E_i the worst-case execution time of EP_i. If the task pair is periodic, T_i denotes its period, whereas T_i represents a minimal inter arrival time if the task pair is sporadic. According to common terminology we call the instances of a task pair jobs. $J_{i,k}$, $k \geq 1$, denotes the k^{th} instance of task pair τ_i. Each job has a release time here $r_{i,k}$ and deadline $d_{i,k}$, and consists of two parts $MP_{i,k}$ and $EP_{i,k}$. For a periodic task pair τ_i, the release time of its k^{th} instance is given by $r_{i,k} := (k-1)T_i$.

Using a model in which the WCETs of the main parts are not known, it is not possible to guarantee for a given set of task pairs that each instance $MP_{i,k}$ is completed by its deadline. This means that the feasibility of the schedules cannot be guaranteed. It does not mean, however, that TAFT provides no guarantees at all. For a set of task pairs $\{\tau_i \mid i \in 1..n\}$ that passed an acceptance test TAFT produces schedules with the following properties: For each job $J_{i,k}$

- The main part $MP_{i,k}$ is guaranteed to be completed unless its actual execution time exceeds the specified resource demand C_i *(completion of correct jobs)*;
- The exception part is completed if (and only if) the main part $MP_{i,k}$ is not completed *(exception handling)*;
- Neither the exception nor the main part is ever executed after the deadline $d_{i,k}$ *(timeliness)*.

To capture these properties under a common notion, we say that such a schedule achieves *timely completion of correct jobs and exception handling*. The acceptance test can be performed before runtime for a static task set or dynamically at runtime whenever a new task arrives. In the latter case, the test is applied to the set of already accepted tasks plus the newly arriving task. If the whole set passes the test the new task is accepted, otherwise it is rejected. How the acceptance test actually works depends on the scheduling algorithm used to implement TAFT. For example, we will present a utilization-based test for the scheduling algorithm proposed in Section 5.4. As a prerequisite for some formal proves we will provide in Section 5.4, we now define more formally the property of timely completion of correct jobs and exception handling. Before doing so in Definition 5-1, we have to introduce some notions:

For a set $\{\tau_i \mid i \in 1..n\}$ of task pairs, let $\mathcal{E} := \{MP_{i,k} \mid i \in 1..n, k \geq 1\} \cup \{EP_{i,k} \mid i \in 1..n, k \geq 1\}$ be the corresponding set of all scheduled entities. For any given schedule σ for $\{\tau_i \mid i \in 1..n\}$, let $completes_\sigma$ be a predicate over $\mathcal{E}$ such that $completes_\sigma(e)$ is *true* if and only if e is completed in σ, $running_\sigma$ a predicate over $\mathcal{E} \times \mathbb{T}$ [5]such that $running_\sigma(e,t)$ is *true* if and only if e is running at time t in σ, and c_σ be a mapping from $\mathcal{E}$ to $\mathbb{R}_0^+$ such that $c_\sigma(e)$ is the total amount of processing time assigned to e in σ.

Definition 5-1 *A schedule* σ achieves timely completion of correct jobs and exception handling *for a set* $\{\tau_i \mid i \in 1..n\}$ *of task pairs if and only if for all* $i \in 1..n$, $k \geq 1$, *and all times* t:

(1) $completes_\sigma(MP_{i,k})$ *or* $c_\sigma(MP_{i,k}) \geq C_i$

(2) $(running_\sigma(EP_{i,k},t)$ *or* $running_\sigma(MP_{i,k},t)) \Rightarrow t \leq d_{i,k}$

(3) $($*not* $completes_\sigma(MP_{i,k}) \Rightarrow completes_\sigma(EP_{i,k}))$ *and*
$(running_\sigma(EP_{i,k},t) \Rightarrow$ *not* $completes_\sigma(MP_{i,k}))$

Summarizing the description of TAFT, one can say that TAFT allows scheduling the main functionality of a task, which is realized in the main part of a task pair, with a realistic resource demand instead of a WCET. Even though this implies that main parts may be subject to resource faults, TAFT ensures that task pairs do not exhibit timing failures and resource faults do not propagate to other task pairs. Thus, TAFT achieves a predicable timing behavior of the tasks.

Nevertheless, TAFT may still have to abort main parts. But without resorting to WCETs no scheduler is able to guarantee that all jobs are completed by their deadline. According to the TAFT scheduling concept, application-inherent redundancy is used tolerate task abortions. The timely exception handling of TAFT provides the means to support this approach. In the following section, we examine what kinds of application-inherent redundancy are present in cooperative mobile applications and suggest how they can be exploited to tolerate task abortions.

[5] $\mathbb{T} := \mathbb{R}_0^+$ represents the set of points of time (cf. Chapter 3)

5.3 Exploiting Application-Inherent Redundancy

In order to preserve a predictable timing behavior, TAFT may have to abort faulty main parts. According to the TAFT scheduling concept, application-inherent redundancy is exploited to tolerate such *overload* situations in which the scheduler is no longer able to complete all main parts by their deadlines. Of course, the exploitation of application inherent-redundancy is by its very nature dependent on the application at hand; what kinds of redundancy are present and how they are best exploited is specific to the application. Nevertheless, there are several kinds of redundancy that are commonly found in cooperative mobile applications. Actually, knowing that task abortions may be unavoidable, the inherent redundancy can be provided by a careful design so that it can be exploited at runtime. In this section, we analyze what kinds of application-inherent redundancy can be provided and how they can be exploited in conjunction with TAFT. We use the distributed sensor fusion scenario as a concrete example for this analysis. Chapter 6 presents measurement results from the prototypical implementation of this scenario.

Three kinds of application-inherent redundancy can be exploited to tolerate abortions of main parts:

- *Functional redundancy* within the main parts, which refers to the fact that a main part not necessarily needs to be completed to provide sufficient results;

- *Spatial redundancy* within the group of cooperating mobile systems, which means that within a group the same part of the environment is frequently observed by several mobile systems; so, even if a main part on one of these systems does not deliver sufficient results, the others can provide the missing information; and

- *Time redundancy* w.r.t to the number of task instances that must provide sufficient results. If a task pair is scheduled with a frequency higher than required to ensure the safety of the systems, a certain number of task instances not providing sufficient results can be tolerated.

These kinds of redundancy realize a two-level fault-tolerance approach. Functional redundancy allows tolerating the faults directly within the task pair before they become visible at the task pair's interface. This means that an aborted task instance still provides sufficient results and hence does not exhibit a failure. Spatial and time redundancy allow tolerating the faults, once they led to failures of the task pair. The former allows tolerating the fault if other members provide for the missing information, whereas the latter does as long as the number of faults remains below a certain threshold.

Exploiting the above-mentioned kinds of redundancy does not require explicit runtime actions on part of the application. As long as they are sufficient, we talk of a *transient overload*; otherwise, we call this a *persistent overload*. This means we are faced with a persistent overload if the number of consecutive executions of the application that do not provide sufficient results exceeds some application-specific threshold. Applications being able to reduce their resource demand by gracefully degrading their service, exhibit another kind of functional redundancy that can be exploited during persistent overload. As compared to the functional redundancy above, this one requires the application to adapt to the current load situation explicitly. The exception parts of the task pairs, which are able to detect and signal persistent overload situations, can trigger this adaptation.

5.3.1 Functional Redundancy Through Anytime Algorithms

Functional redundancy within the instances of the main parts is the first kind of application-inherent redundancy we consider. As it is known at design time that the main parts might be aborted at runtime, it is wise to design them in such a way that partly executed instances are not completely useless. The exception part provides a means to make intermediate results available if the main part was designed in such a way that preliminary results are available before the task completes. *Anytime algorithms* are the means to achieve this goal.

The basic idea of anytime algorithms is to compute some first results as soon as possible and then iteratively improve these results until the best possible result of the algorithm is achieved (Dean and Boddy 1988,Boddy and Dean 1989). This way of realizing an algorithm has the advantage that the algorithm may be stopped at any time and always has some preliminary result to deliver. There are the following measures for the quality of preliminary results (cf. (Russel and Zilberstein 1991)):

- *Certainty*. This means that the result represents a kind of "first guess" of the algorithm. The result has a certain probability of being correct, though it may not yet have the highest probability the algorithm can achieve. Other possible solutions still have to be evaluated, and it may turn out that they have a higher likelihood of being correct. For example, the arc filter in the distributed sensor fusion successively evaluates a sequence of estimated arc positions. If it is aborted prematurely, the estimate with the highest evaluation may not yet have been found.

- *Accuracy*. This means that the algorithm is still reducing the error of the result, which refers to the distance (in a general sense) between the result and the real-world entity it represents. For example, the algorithm may still be able to reduce the spatial distance between the estimated position of some object and its real position.

- *Completeness*. This means that the algorithm delivers results that only represent a part of the real world entity they relate to. For example, if the expected result is a perception of the environment represented by a set of edges, an incomplete result may contain only some of the edges.

- *Specificity*. This means that the results, while representing the whole real world entity they relate to, are not yet as detailed as possible.

Despite being preliminary in the above senses, the results delivered by an aborted anytime algorithm may still prove useful or even be sufficient. If the latter is the case, all computations the algorithm performs after achieving the sufficient result can be considered as functional redundancy. Anytime algorithms allow exploiting this redundancy. All-or-nothing algorithms either provide no result at all or the best possible result (including all the redundancy); a sufficient preliminary result is not available. Plotting the value of an algorithm against its execution time, an all-or-nothing algorithm exhibits a single step when it is completed, whereas an anytime algorithm exhibits a continuously increasing curve or a number of small steps. With some consideration at design time, many algorithms can be designed in such a way. For example, all filters in the distributed sensor fusion adopt this paradigm. Furthermore, (Bade 2003,Herms 2004) show how it can be applied in stereovision and complex planning applications.

When scheduling anytime algorithms with TAFT, the main part consists of the anytime algorithm, whereas the exception part delivers the results computed so far when the main part is aborted. In this approach, the anytime algorithm is automatically stopped and its results are automatically delivered when it cannot be completed by its deadline. Thus, the scheduler decides how long the anytime algorithm is run and ensures that it delivers its results in time.

Using this approach, it is easy for application designers to employ anytime algorithms. They only need to design the algorithm and provide the deadline and expected-case execution time to the scheduler, which makes the decisions at runtime. The ECET allows application designers to specify a minimum time for the algorithm to be executed. This ensures that it is not terminated arbitrarily early, in which case it might deliver no or only very-low-quality results. By contrast, a high probability that sufficient results are delivered can be achieved. In fact, if a WCET required to compute at least a sufficient result was known (as is assumed in (Lin et al. 1987,Liu et al. 1994)), the ECET of the main part could be set to this WCET and the main part would always yield sufficient results.

Realized as anytime algorithms, main parts that have to be aborted may still deliver sufficient results. If so, the task pair exhibits no failure, since it provides a service according to its specification. The resource fault of the main part, therefore, has been tolerated. Thus, the approach exploits the functional redundancy within the main parts to tolerate resource faults. Considering the task pair as a component of the application, one can say that the approach increases the reliability of this component, for it increases the ratio between the number of task pair instances providing sufficient results and the number of all released instances. For example, measurements conducted in the prototype of the distributed sensor fusion show how using anytime algorithms allows increasing the reliability of the arc filter (cf. Section 6.2).

The increased reliability notwithstanding, there may be task pair instances not providing sufficient results to meet their specification. This is unavoidable unless WCETs for a logical mandatory part are assumed. The following clause will show how such component failures can be tolerated.

5.3.2 Spatial and Timing Redundancy

When an instance of a main part is aborted without providing sufficient results this represents a failure of the corresponding component of the application. This constitutes a fault from the perspective of the overall application. In this sub-section, we consider how such faults can be tolerated.

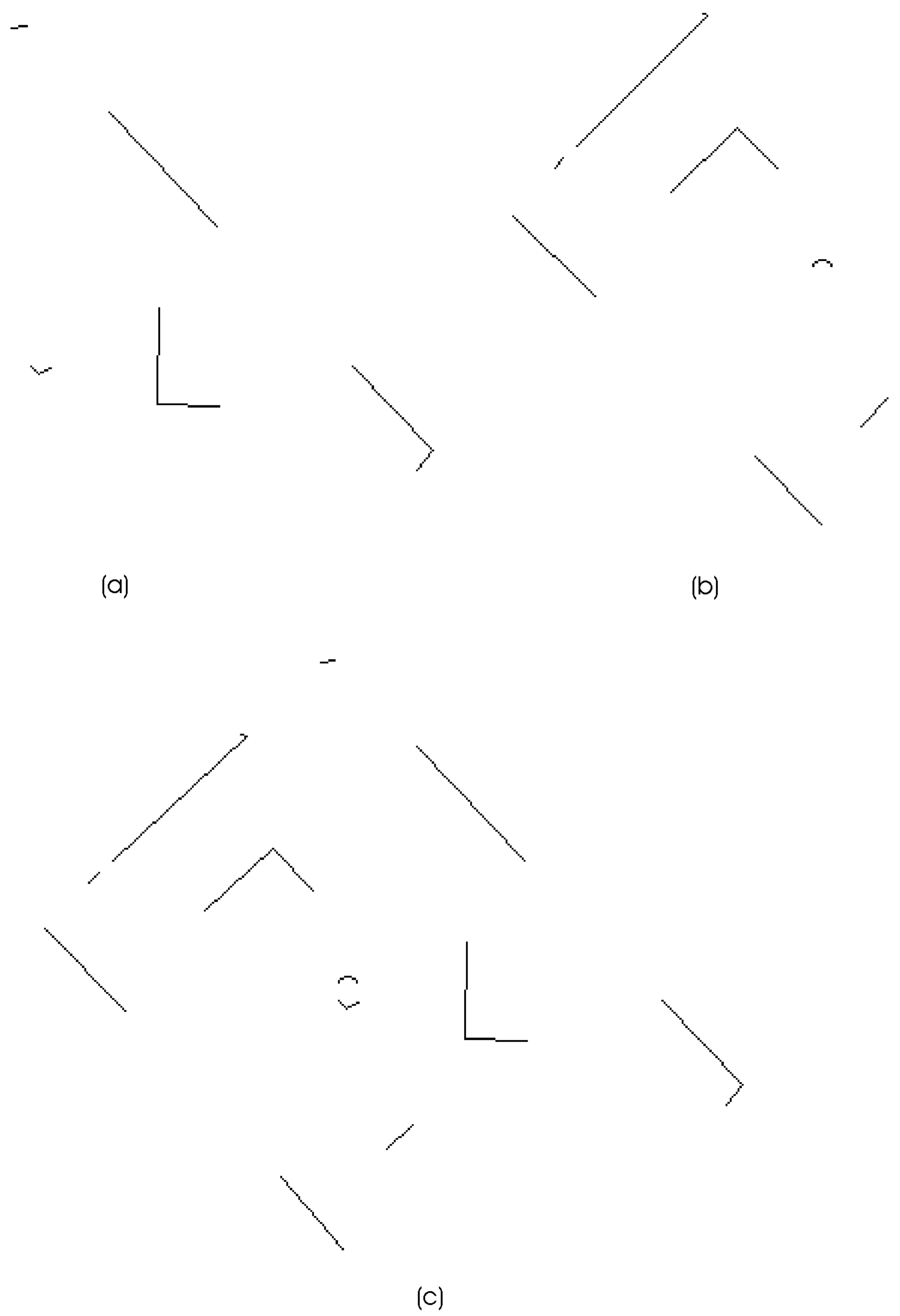


Figure 5-8. Structural redundancy in the distributed sensor fusion

Spatial redundancy is the first kind of redundancy being exploited to tolerate such faults. In a group of embedded systems observing some real-world entity in their environment, if a component on one of the systems fails to deliver an observation, the others may still provide the missing information in time. The distributed sensor fusion, for example, combines results from several laser scanners that observe the environment from different perspectives. The fusion automatically tolerates component failures that result in data from one of the sensors being missing as long as two or more sensors are observing the same part of the environment. Moreover, incomplete results from a faulty module may still represent a valuable contribution to the results of the fusion. Figure 5-8 shows an example: Figure

5-8 (a) and (b) depict two incomplete observations from two laser scanners. The scan depicted in Figure 5-8 (a) contains edges from only one of the rectangular objects in the scene, while the scan depicted in Figure 5-8 (b) only contains edges of two other rectangular objects. The fusion of both of the scans (Figure 5-8 (c)), however, comprises edges of all three rectangular objects so that they can all be detected in the fused scan.

Timing redundancy is the second kind of redundancy being exploited. In many control applications, control loops are executed well above their stability criteria. This means that the corresponding tasks are performed with a period significantly smaller than necessary to accomplish a stable control of the system. Such applications exhibit redundancy in the number of task instances that must provide sufficient results. This is because two considerations guide the selection of the period: It must be sufficiently small so that the controller (i) can react to changes in the environment before the controlled system is damaged (a safety constraint) and (ii) exhibits a smooth reaction to the changes in the environment (a quality goal). While being less critical regarding system safety, (ii) implies the more stringent timing requirements. For example, there is a minimum frequency at which motion planning, and hence the sensor fusion, must be executed to avoid collisions. It depends on the sensor range, the speed of the robot, and the speed of the surrounding objects. Usually, motion planning and sensor fusion are performed at a much higher frequency to achieve a smooth driving.

Thus, when scheduling controllers, frequently only m out of n scheduled task instances are really hard, while the remaining $n - m$ instances can be considered as timing redundancy, which can be exploited in overload situations. Systems with such constraints, called $\binom{m}{n}$ constraints, are also known as weakly-hard real-time systems (Hamdaoui and Ramanathan 1995,Koren and Shasha 1995,Bernat and Burns 2001,Bernat and Cayssials 2001,Wang et al. 2002).

Exploiting the kinds of application-inherent redundancy discussed so far, transient overloads can be tolerated without explicit fault treatment on part of the application. In the following subsection we consider what can be done if an overload situation is more persistent.

5.3.3 Signaling Persistent Overload

If the execution times of a main part persistently exceed its ECET such that it observers faults over some extended period of time, relying on the kinds of redundancy described above does no longer suffice and we are faced with a persistent overload situation. There are two ways to address the problem:

- One can adapt the specified resource demand of the main part — that is, its ECET — to its actual resource demand so that the scheduler allocates more resources for executing this task.

- One can adapt the resource demands of the application in order to reduce the system load.

(Gergeleit 2001) shows how the first approach can be realized. He combines the TAFT scheduler with an online-monitoring component, which provides execution time statistics to the scheduler. Using these statistics, the scheduler can adapt the resource allocation to changing execution times. This approach allows adapting the guarantee the scheduler provides to the demands of the tasks as long as the task set with the increased resource demands passes the acceptance test. If the latter is no longer the case, the second approach can be used. This approach has to prerequisites: First, the application must be able to adapt its resource demand by gracefully degrading its service; second, the middleware must signal the overload situation to the application. The exception handling of TAFT provides the means to detect and signal persistent overload situations. To exploit it, the ratio between the number of aborted instances of the main part and the number of all instances of the main part is computed in the exception part of a task pair. Actually, according to the kind of constraint presented in the preceding sub-section, this value is computed over an interval of n released instances, where n is specified by the application. The number k of aborted instances within this interval is counted. If k exceeds an application-specific threshold a persistent overload is detected. In this case, the exception part signals the overload and thus triggers the adaptation of the application. Thus, the exception handling TAFT guarantees per task pair instance can be used to trigger a next level of exception handling if an $\binom{m}{n}$ constraint is violated or about to be violated.

The distributed sensor fusion scenario allows for adaptation under overload. As explained in Section 3.2, performing the fusion on a higher level of abstraction allows adapting the resource demands of the distributed sensor fusion at the expense of its accuracy. In a persistent overload situation, the sensor fusion is switched to a higher level of abstraction. Thus, the system load is reduced and the ratio of aborted task instances is reduced also as a result.

5.4 Aperiodic Requests

So far, we explained how using the TAFT concept and exploiting several kinds of application inherent redundancy achieves a timely predictable and reliable execution of applications. In this section, we present a scheduling algorithm that implements TAFT for a task model fitting the requirements of our middleware.

The task model underlying the existing implementation of TAFT includes sets of independent periodic task pairs (Becker and Gergeleit 2001,Becker et al. 2003,Gergeleit et al. 2003). While periodic task pairs adequately model the application tasks in our application domain, they are not well suited to model the tasks that execute the communication protocols of the middleware. The latter — referred to as *communication tasks* — are characterized by varying inter-arrival times and response time requirements that are much smaller than the inter-arrival times. They are thus better modeled as aperiodic requests that arrive in the system at a priori unknown instants and should be served as soon as possible. Actually, as we assume a bounded scheduling delay in the system model presented in Chapter 4, aperiodic requests have deadlines in our task model. As the unknown inter-arrival times do not allow giving an a priori guarantee that all aperiodic requests will be completed by their deadline, a guarantee is given on a per instance basis. To this end, an acceptance test checks whether an aperiodic request can be completed before its deadline.

In this section, we present a scheduling algorithm that schedules task sets consisting of periodic independent task pairs and aperiodic tasks. For the aperiodic tasks, a guarantee is provided on a per instance basis. This means that an aperiodic request is either completed by its deadline or not executed at all. This section is structured as follows. First, we present the extended task model that the scheduling algorithm has to accommodate (Sub-Section 5.4.1), before considering existing algorithms for the scheduling of hybrid task sets consisting of periodic tasks and aperiodic requests (Sub-Section 5.4.2). We then present the scheduling algorithm, which is based on the idea of using an existing aperiodic server algorithm — the IPE server — to schedule the task pairs and the aperiodic request (Sub-Section 5.4.3). We present acceptance tests for both the set of periodic task pairs and the aperiodic requests (Sub-Section 5.4.4).

5.4.1 Model

The communication tasks or not adequately modeled as periodic tasks. The first reason is that they are not activated at constant intervals, even tough polling is performed in a round-based manner. For one thing, whenever the group size changes, so does the length of the polling list and hence the activation interval of the communication tasks. Furthermore, aspects like the following ones cause additional jitter:

1. Request frames and mc frames may include application messages or not;
2. Application messages may have varying lengths;
3. Both polling and request frame may be received, or the AP may have to wait for a timeout.

The second reason for the communication tasks' not fitting to the periodic task model is that their deadlines are much smaller than their periods. Consider, for example, the task of reacting to a poll. Although the period of this task corresponds to one round length, the deadline obviously does not, for the station must react to the polling frame much earlier than just before receiving the next poll. In fact, it should response as soon as possible to keep the idle time of the medium small. In the periodic task model, it is commonly assumed that the relative deadlines of the tasks are equal to their periods and that all possible completion times of a periodic task are equally well as long as they are not greater than the deadline.

The communication tasks can be modeled as sporadic tasks. Despite not being constant, the inter-arrival times of the communication tasks can be characterized by a lower bound. Using this lower bound for acceptance testing would allow guaranteeing that all instances of the tasks are completed by their deadline. However, we decided not to pursue this approach because we do not want to use worst-case inter-arrival times of the communication tasks for acceptance testing. This would be in contrast to our general approach, which is to avoid worst-case assumptions for parameters that are hard to predict.

Because of the above arguments, we decided to model the communication tasks as aperiodic, event-triggered tasks R_i, $i \geq 1$. An aperiodic task R_i is characterized by its execution time C_i and a relative deadline D_i, and is hence represented as a tuple $R_i = (C_i,D_i)$, $i \geq 1$.

We assume that the execution times of the communication tasks are small and predictable. Therefore, C_i is assumed to be a worst-case bound and significantly smaller than the ECETs of the application tasks. The k^{th} instance of an aperiodic task R_i is denoted as $R_{i,k}$ and called an aperiodic request. The release time of $R_{i,k}$ is denoted as $r_{i,k}$ and its deadline as $d_{i,k}$, where $d_{i,k} := r_{i,k} + D_i$. We assume that the relative deadlines are smaller than the inter-arrival times of the requests.

Without resorting to a worst-case arrival pattern, it is not possible to guarantee that all aperiodic requests are completed by their deadlines, the known-execution-time assumption notwithstanding. To achieve a predictable timing behavior in spite of this new kind of unpredictability in the system load, two things have to be done. First, similar to the concept of fault containment in TAFT, it must be ensured that the unpredictable load of aperiodic requests does not compromise the predictability of the periodic tasks. This means that the timely completion of correct jobs and exception handling guarantee must be preserved for the periodic tasks, whatever the aperiodic load may be. Second, timely predictable execution of the aperiodic requests must be accomplished. That is, the execution of a request should only be started if it will be completed by its deadline. This corresponds to the system model underlying our communication protocols (see Chapter 4), which assumes that the execution service of the stations is not subject to timing failures. Thus, we consider the aperiodic tasks to be firm tasks for which a guarantee is provided on a per request basis. This means that an acceptance test checks whether a newly arriving request can be completed in time. If so, the request is accepted; otherwise it is rejected.

Combining the existing model of periodic independent task pairs with the aperiodic task model introduced above, we come to the following task model: The task set is the union of a set of periodic task pairs $\{\tau_i = (T_i,C_i,E_i) \mid i \in 1..n\}$ and a set of aperiodic tasks $\{R_i = (C_i,D_i) \mid i \in 1..m\}$. The timing of the periodic task pairs is characterized by a three parameters C_i, E_i, and T_i, where the former two represent the excepted-case execution time of the main part and the worst-case execution time of the exception part respectively and the latter is the period. The relative deadlines of the instances are equal to the period. Analogously to its definition for a set of period tasks, we define the *utilization factor* for a set of periodic task pairs $\{\tau_i = (T_i,C_i,E_i) \mid i \in 1..n\}$ to be

$$U := \sum_{i=1}^{n} \frac{C_i + E_i}{T_i} \quad .$$

5.4.2 Scheduling Algorithms for Hybrid Task Sets

In this sub-section we consider existing algorithms for the scheduling of hybrid task sets, which consist of periodic tasks with hard deadlines and aperiodic requests. We will focus our discussion on works in the context of EDF scheduling for the following two reasons:

- The current implementation uses two earliest deadline scheduling algorithms: EDF and EDL (earliest deadline as late as possible);

- EDF, and EDL too, has a utilization-based schedulability criterion with an utilization bound of 1. Basing the implementation on such an algorithm bears the potential to achieve a similarly high utilization factor.

The task model underlying the following algorithms deviates from ours. It models aperiodic requests as soft, rather than firm tasks. They are not characterized by a common worst-case execution time or deadline either. Therefore, no acceptance test is performed for the aperiodic requests; they are served on a best-effort basis, the goal being to minimize their response times. Furthermore, the periodic jobs are instances of simple, hard tasks, not task pairs of course. The implication of this latter difference will be discussed in Sub-Section 5.4.3, where we introduce our approach. These differences notwithstanding, considering these works is still worthwhile because of the two major analogies with our model: aperiodic requests must not compromise the guarantees given for periodic tasks and should be completed as soon as possible. The presentation is based on (Stankovic et al. 1998), which gives a good survey of the field. We do not strive to give a complete overview here, but restrain ourselves to those algorithms relating to the scheduling algorithm we present in the following sub-section (5.4.3). In what follows, conventional periodic tasks are represented as $\tau_i = (T_i,C_i)$, where T_i is the period and C_i the worst-case execution time, and the utilization factor of a set of periodic tasks $\{\tau_i = (T_i,C_i) \mid i \in 1..n\}$ is defined as $U := \sum_{i=1}^{n} \frac{C_i}{T_i}$.

5.4.2.1 The Dynamic Priority Exchange (DPE) Server

The DPE server is a periodic server task that handles aperiodic requests; that is, aperiodic requests are executed during the times allocated for the server task (Spuri and Buttazzo 1996). Without additional measures, the time allocated for the server task is assigned to the other periodic tasks and is lost for the execution of aperiodic requests whenever no aperiodic request is waiting to be served when the server task becomes the highest priority task in the pending queue. The basic idea of the DPE server is to let the server task exchange its allocated executions times with lower priority periodic tasks if no aperiodic requests are pending. This means that the lower priority tasks run at the priority of the server task until the execution time allocated for the server has been consumed. When an aperiodic request arrives afterwards, the allocated execution time is exchanged back to the server so that the server can use it to execute the aperiodic request. Thus, the execution times allocated for the server task are not lost if no aperiodic request is pending, but they are preserved to be used later.

In more detail, the DPE server works as follows. So-called *aperiodic capacities* (or capacities, for short) are associated with all periodic tasks. Each capacity is assigned a priority according to the deadline of the last released instance of the corresponding periodic task, even if that instance has been completed already. In case a task and a capacity have the same deadline, ties are broken in favor of capacities. Whenever an instance of the server task is released, the corresponding capacity is set to the execution time allocated for the server task; all other capacities are initially 0. When an aperiodic capacity Γ is the highest priority entity — task or capacity — the scheduler chooses a task instance to execute in the following priority order:

1. A pending aperiodic request;
2. The periodic task instance with the shortest deadline;
3. The idle task.

The scheduler executes the selected task instance for at most Γ time units and subtracts the time the instance was running from the capacity Γ. When a periodic instance runs under a capacity for Δ*t* time units the capacity associated with that task increases by Δ*t*. So, if a periodic instance runs under a capacity, this means that the capacity is exchanged and preserved in the system.

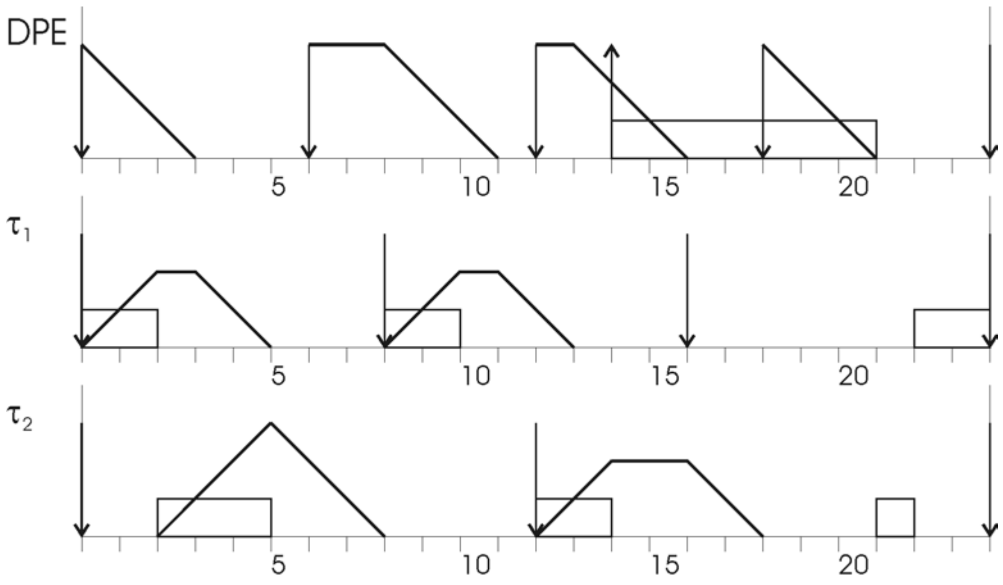


Figure 5-9. Example schedule of the DPE server (cf. (Stankovic et al. 1998 p. 171)).

Figure 5-9 shows an example schedule of the DPE server. In the depicted scenario, there are two periodic tasks, $\tau_1 = (8,2)$ and $\tau_2 = (12,3)$, and a DPE server, DPE = (6,3). The bold lines in the figure visualize the aperiodic capacities of the instances. At time 0, the highest priority entity is the capacity of the DPE server. The scheduler executes the first instance of τ_1 ($J_{1,1}$) and, for 1 time unit, the first instance of τ_2 ($J_{2,1}$) under that capacity. Accordingly, both instances accrue a capacity themselves. The remaining 2 time units of $J_{2,1}$ are executed under the capacity of $J_{1,1}$, which is now the highest priority entity. Thus, by time 5, $J_{2,1}$ has accrued a capacity of 3, which at that time becomes the highest priority entity. This capacity is exhausted by time 8 with executing the idle task, since there is neither an aperiodic request nor a periodic task instance pending. So, the initial capacity of 3 time units, which was preserved until time 5 is lost by time 8. At time 8, $J_{1,2}$ is released and is executed under the capacity of the DPE server; so $J_{1,2}$ accrues a capacity of 2 time units meanwhile. During times 10 to 12, the capacities of the server and of $J_{1,2}$ are spent for the idle task. When $J_{2,2}$ is released at time 12, it is first executed for one time unit under the capacity of $J_{1,2}$ and then, for another time unit, under the capacity of the server. So it has a total capacity of 2 time units when an aperiodic request is released at time 14. At that time the highest priority entity is the capacity of the server. Therefore, the aperiodic request is executed for 2 time units under that capacity and then, for another two time units, under the capacity of $J_{2,2}$. At that time (18), the capacity of the server is replenished so that there are 3 further time units available to execute the request, which is sufficient to complete it.

This example nicely shows, how the capacities initially allocated for the server task are preserved by exchanging them with the periodic tasks.

5.4.2.2 The Earliest Deadline Late (EDL) Server

The EDL server is an optimal algorithm w.r.t the response times of aperiodic requests (Spuri and Buttazzo 1996). It works as follows: As long as no aperiodic request is pending the periodic tasks are executed according to EDF. When an aperiodic request arrives, say at time t, the scheduler computes the idle times of an EDL schedule for the current job set $\mathcal{J}(t)$. Under EDL scheduling, jobs still have priorities according to their deadlines, a shorter deadline implying a higher priority, but tasks are executed as late as possible (Chetto and Chetto 1989). The set $\mathcal{J}(t)$ comprises all periodic task instances active at time t with their remaining allocated processing times — their WCET minus the processing time they already received — and all future periodic instances within the same hyper period with their WCET. The EDL server assigns the thus computed idle times to the aperiodic request. Once the request has been completed, the scheduler returns to normal operation; that is, it schedules the periodic instances according to EDF.

Figure 5-10 gives an example of how the EDL server works. Two periodic tasks have to be scheduled, τ_1 with a period of 6 and a WCET of 3, and τ_2 with a period of 8 and a WCET of 3. The instances of these tasks are scheduled with EDF until time 8 when an aperiodic request arrives (light gray in the figure). At this time, the scheduler computes an EDL schedule for the remaining periodic instances in the hyper period (represented by the dashed boxes in the figure). For the current instance of τ_1, only its remaining allocated processing time of 1 is considered. The lowest graph in Figure 5-10 (a) depicts the idle times of this schedule (denoted by the function $\omega(t)$). Figure 5-10 (b) shows how the scheduler proceeds after computing the EDL schedule. First, it executes the aperiodic request during the idle times of the EDL schedule. After two time units, at time 10, the aperiodic request is completed. Afterwards, it returns to EDF scheduling for the periodic instances.

The optimality of this algorithm w.r.t the response times of the aperiodic requests stems from a result presented in (Chetto and Chetto 1989). They show that for any time t, there is no other scheduling algorithm providing more idle times during $[0,t]$ than EDL. Thus, a maximum of idle times is made available whenever an aperiodic request arrives. The main problem with this algorithm is the time complexity of computing the idle times of the EDL schedule. As this computation has a complexity of $O(Nn)$, where n is the number of periodic tasks and N is the number of instances within a hyper period, it may not be practically feasible to use this approach (Stankovic et al. 1998). In particular, N may be quite large if the tasks do not have harmonic periods. Hence, it is worthwhile looking for a scheduling algorithm that comes close to the EDL server regarding response times of the aperiodic requests, but incurs less overhead. The IPE server presented in the following clause is such an algorithm.

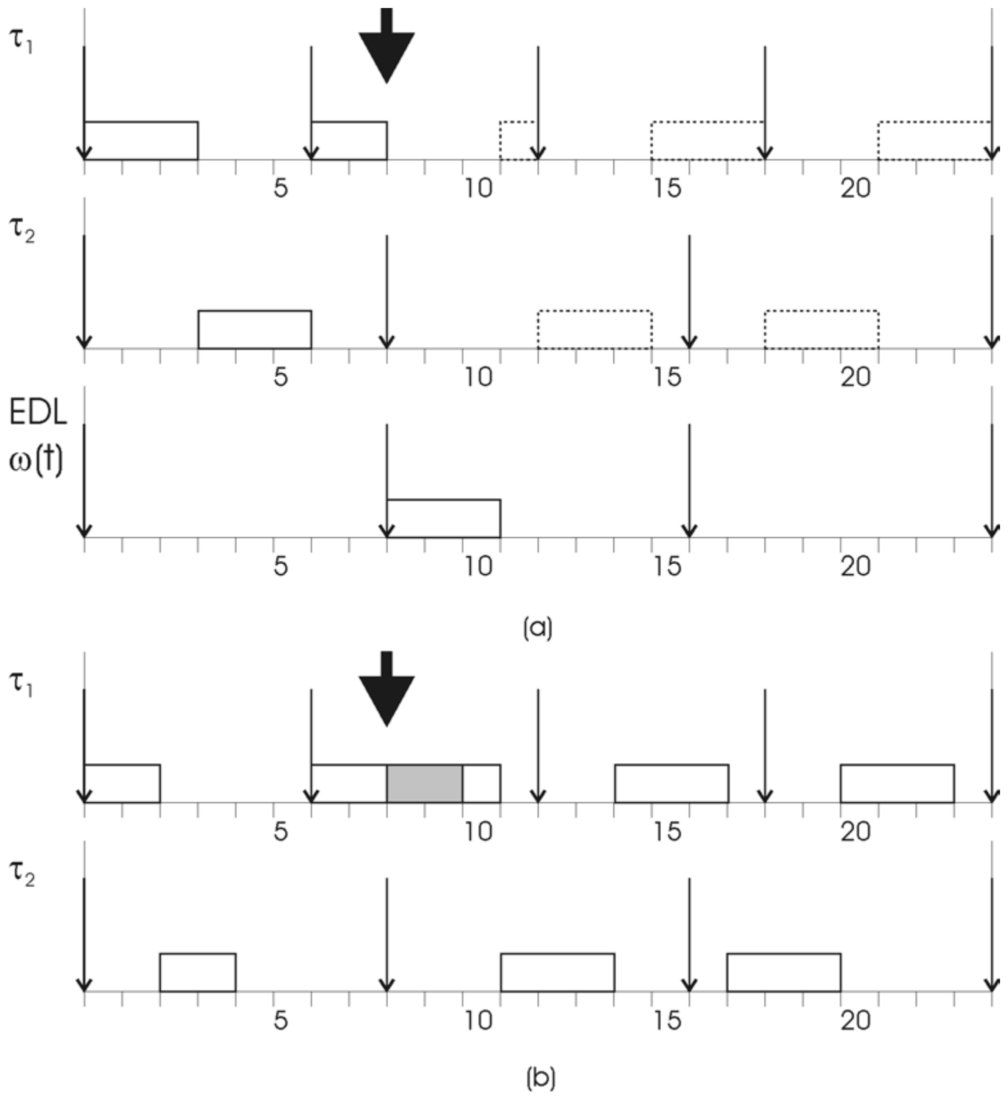


Figure 5-10. Example schedule for the EDL server

5.4.2.3 The Improved Priority Exchange (IPE) Server

The IPE server avoids the computational overhead of the EDL server, yet still achieves a near optimal performance w.r.t the response times of aperiodic requests (Spuri and Buttazzo 1996). It combines ideas from the DPE and the EDL server. In principle, it works nearly the same way as the DPE server. What changes is the way in which the capacities of the server task are replenished. While in the DPE server the initial capacities are replenished periodically by some fixed amount of time allocated for the server task, the IPE server uses the idle times of an EDL schedule of the periodic tasks to replenish the server capacities. This means that whenever an idle time interval of the EDL schedule starts, a capacity equal to the length of the idle time interval is added to the capacity of the server task. As EDL maximizes those idle times (Chetto and Chetto 1989), it maximizes the initial capacities of the server tasks.

Exchanging capacities works the same way as in the DPE server. Capacities get priorities according to their deadlines. The capacity of the server always has the highest priority. If a capacity is the highest priority entity, a task instance is selected for execution under the capacity in the following order:

1. A pending aperiodic request;
2. The periodic task instance with the shortest deadline;
3. The idle task.

So, if no aperiodic request is pending, the capacity is exchanged with a periodic instance so that it is preserved and can be used when an aperiodic request arrives later.

The IPE server has been shown to exhibit a performance, which is comparable to that of the EDL server and hence nearly optimal w.r.t the response times of aperiodic requests (Spuri and Buttazzo 1996). Furthermore, it feasibly schedules each periodic task set with a processor utilization not greater than one. Expressed more formally, this means: For each set $\mathcal{T} := \{\tau_j = (T_j, C_j) \mid j \in 1..n\}$ of periodic tasks, the IPE server feasibly schedules $\mathcal{T}$ if and only if

$$U := \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

The drawbacks of the IPE server as compared to the DPE server, or other periodic server approaches, is that it is based on the idle times of an EDL schedule of the periodic task set, which is computed offline. Storing the idle times incurs a certain memory overhead, in particular if the periods of the periodic tasks are not harmonic. The current implementation of TAFT (Becker and Gergeleit 2001,Becker et al. 2003), which is also based an EDL schedule for a periodic task set (for the exception parts in this case), shows that using an EDL schedule for the periodic tasks is a viable solution. We therefore decided to base our implementation on the IPE server due to the advantages stated above.

5.4.3 Realizing TAFT with the IPE Server

In our discussion of existing scheduling algorithms for hybrid task sets, the IPE server turned out to be a promising candidate for scheduling aperiodic requests together with periodic tasks. The main requirement for the scheduling algorithm to be used in the execution service, however, is that it implements the TAFT concept. This means it must be able to schedule task pairs and achieve a timely completion of correct jobs and exception handling. In this sub-section, we therefore consider how the IPE server can be extended to schedule periodic task pairs instead of conventional periodic tasks. We present the solution in two steps. In the first step, we modify the IPE server such that it guarantees timely completion of correct jobs and exception handling for a set of periodic task pairs and hence fulfills the basic requirement. In the second step, we extend this algorithm in order to allow the scheduler to keep faulty main parts in its pending queue and complete them if possible; that is, we enhance the performance of the algorithm w.r.t to the completion of faulty main parts. After explaining both steps, we present a formal description of the resulting algorithm, which is called TAFT-IPE. TAFT-IPE combines the following three advantages:

- It guarantees timely completion of correct jobs and exception handling for each set of periodic task pairs with a utilization factor not greater than one;
- It achieves a nearly optimal performance w.r.t to the response times of the communication tasks;
- It makes use of remaining processor idle times to complete faulty main parts.

5.4.3.1 The Basic Algorithm

We present here an extended version of the IPE server, called *TAFT-IPE(basic)*, which is able to schedule periodic task pairs. TAFT-IPE(basic) works as follows: Whenever an instance $J_{i,k}$ of task pair $\tau_i = (T_i,C_i,E_i)$ is released, the scheduler adds the main part $MP_{i,k}$ to the pending queue and schedules it the same way as the IPE server schedules a conventional periodic task. Scheduling the main part like a periodic task under the IPE server ensures that aperiodic requests cannot prevent it from getting up to the specified amount of resources (C_i) by its deadline. However, while a conventional task never exceeds its specified resource demand, the main part $MP_{i,k}$ may do so. Therefore, TAFT-IPE(basic) performs an accounting of the actual execution time of the main part. As soon as it is equal to the specified resource demand C_i, the scheduler aborts the main part and continues with the execution of the exception part $EP_{i,k}$ instead. The exception part too is scheduled in the same way as the IPE server would schedule a conventional periodic task. Again, scheduling the exception part like a periodic task under the IPE server ensures that aperiodic requests cannot prevent the exception part’s getting up to the specified amount of resources (E_i) by its deadline.

In this algorithm, the amount of resources assigned to any job $J_{i,k}$ is limited by $C_i' := C_i + E_i$. Therefore, the resources consumed by a task pair $\tau_i = (T_i,C_i,E_i)$ under this algorithm can be represented by a virtual periodic task τ'_i with a specified resource demand of C_i' and a period $T_i' = T_i$. If the IPE server is able to complete each instance of τ'_i by its deadline, this means TAFT-IPE(basic) is able to allocate up to $C_i + E_i$ time units to each instance of τ_i by the deadline of that instance. Therefore, TAFT-IPE(basic) is able to complete $MP_{i,k}$ as long as it has an actual execution time not greater than C_i. If this is not the case, TAFT-IPE(basic) is still able complete $EP_{i,k}$ by its deadline using the remaining specified resource demand. Therefore, TAFT-IPE(basic) guarantees a timely completion of correct jobs and exception handling for a set of periodic task pairs $\mathcal{T} := \{\tau_j \mid j \in 1..n\}$ whenever the IPE server feasibly schedules the corresponding set $\mathcal{T}' := \{\tau_j' \mid j \in 1..n\}$ of virtual periodic tasks. Together with the IPE schedulability criterion presented in Clause 5.4.2.3, this implies that TAFT-IPE(basic) achieves a timely completion of correct jobs and exception handling for a set $\mathcal{T} := \{\tau_j \mid j \in 1..n\}$ of periodic task pairs as long as

$$U = \sum_{i=1}^{n} \frac{C_i + E_i}{T_i} \leq 1 \ .$$

We do not prove this conclusion here; a prove will be provided for the complete TAFT-IPE algorithm in Sub-Section 5.4.4.

TAFT-IPE(basic) aborts main parts directly after they consumed their specified resource demand C_i although there may still be a chance to complete them in time. The current implementation of TAFT exploits remaining processor idle times to try to complete faulty main parts. Since this feature increases the reliability of task execution, we strive to preserve it while extending the supported task model. In the following clause, we show how the basic algorithm can be extended for this purpose.

5.4.3.2 Executing Faulty Main Parts

Our objective in extending TAFT-IPE(basic) is to use remaining processor idle times to complete faulty main parts instead of aborting them directly. According to the TAFT concept, faulty main parts are scheduled on a priority level lower than that of the correct main and exception parts, so as to ensure that the guarantees provided to the latter are not compromised by the faulty main parts; that is, to provide fault containment. Our approach to achieve this is inspired by the following observation: The way aperiodic requests are handled by aperiodic servers is analogous to the lower level of priority of faulty main parts in TAFT: Aperiodic servers try to complete the aperiodic requests, but only as long as the guarantees provided to periodic tasks are not compromised. So, the fact that faulty main parts are scheduled on a lower priority level than correct main parts and exception parts in TAFT can be mapped to the distinction between aperiodic requests and periodic instances in the server approaches. Therefore, the basic idea of the extension is mapping faulty main parts to aperiodic requests.

TAFT-IPE(basic) is extended in the following way: Each time a main part has been executed for C_i time units and is still not completed, the scheduler turns it into an aperiodic request instead of aborting it directly. At the same time, the scheduler adds the exception part to the pending queue. So, a main part's getting faulty corresponds to the arrival of an aperiodic request, while executing the exception part corresponds to continuing the execution of the virtual periodic task. Correct main parts and exception parts are conceptually scheduled on a higher priority level: For both, guarantees are provided, while the faulty main parts are served on a best-effort basis. In particular, this means that the exception part of the faulty task pair is guaranteed to be completed by its deadline unless the faulty main part is completed. Nevertheless, like the IPE server defers executing periodic instances to improve responsiveness of aperiodic requests, TAFT-IPE defers the execution of exception parts to serve the faulty main parts first. So, the scheduler first tries to complete the faulty main parts and executes the exception handling only if required. If the scheduler is able to complete the faulty main part, it removes the exception part from the pending queue. Deferring the execution of the exception parts does not compromise the guaranteed exception handling: As soon as there are no more aperiodic capacities available, the scheduler will stop executing faulty main parts. Therefore, as soon as further delaying the execution of the exception part would result in the exception part's missing its deadline, the scheduler starts executing it. In this case, the faulty main part is removed from the pending queue; that is, the main part is aborted.

In spite of the analogy between handling faulty main parts in TAFT and handling aperiodic requests in the IPE server, there are also the following differences:

1. In the IPE server, delaying the execution of a periodic instance does not impact its chance of being completed because its WCET is known and it is completed by its

deadline in any case. While this is also the case for the exception parts in TAFT-IPE, it is not for the main parts. Here, delaying the execution of a main part means reducing its chance of being completed if it should become faulty.

2. While responsiveness is the main issue for aperiodic requests in the IPE server, it is not per se an issue for the execution of faulty main parts. For them, early execution is only an advantage if it increases the probability of being completed.

Together, points 1. and 2. raise the question whether it is reasonable to defer the execution of a correct main part in order to execute aperiodic requests. Deferring the exception parts is not a problem since it does not affect their probability of being completed; rather, they are guaranteed to be completed whether deferred or not.

In answering this question, we must distinguish between communication tasks and faulty main parts, which are both scheduled as aperiodic requests. We deem it reasonable to defer the execution of correct main parts in order to execute communication tasks. For one thing, communication tasks have small execution times, so they do not delay the execution of the main parts too much. Furthermore, they have strong response time requirements, which warrant their early execution. Therefore, if a capacity is available, communication tasks have the highest priority in using it.

Things are not as clear regarding the execution of faulty main parts. Deferring correct main parts to execute a faulty one allows the first faulty main part to consume all available capacities. Thus, if one of the deferred main parts becomes faulty also, there may be no more capacities available so that it may have to be aborted. Therefore, the first faulty main part would have the best chance to be completed, and furthermore, it would reduce the chance of the following ones. For these reasons, it seems fairer to delay the execution of faulty main parts and execute the correct ones first. Besides increasing the fairness, this keeps the capacities longer in the system, which is of advantage when instance of a communication task arrive. On the other hand, faulty main parts have deadlines. Sooner or later, a faulty main part will be aborted by the corresponding exception part. Always favoring correct main parts may delay the execution of faulty ones until the moment at which the corresponding exception part is started.

Summarizing the discussion above, the following three points can be settled:

- Deferring exception parts is not a problem since they are completed at any rate if required. Therefore, they are never executed under a capacity.
- If a communication task is pending, it is run under the capacity.
- Amongst several faulty main parts, priorities are assigned according to deadlines.

The question remains whether a faulty or a correct main part should be executed under a capacity if both kinds of instances are pending. Two strategies appear to be plausible:

1. Allocate the capacities to the main part with the shortest deadline, whether it is faulty or not; ties are broken in favor of correct main parts. This means that deadlines are the first criterion and the distinction between correct and faulty main parts is the second.

2. Allocate the capacities to a correct main part; among the correct main parts, priorities are assigned based on deadlines. This means that the distinction between faulty and correct main parts is the first criterion and the deadlines are the second.

Both strategies are possible and easily implemented; the first gives focus on trying to complete faulty main parts and not wasting capacities, while the latter stresses fairness. We decided to apply the first approach, because it works better for task having precedence constraints. In a pair of jobs with a precedence constraint, the successor cannot be started before the predecessor is terminated. Therefore, if the predecessor becomes faulty, it is not possible to execute the successor first and then try to complete the faulty predecessor. Furthermore, using deadlines as the first criterion allows using a simple approach to enforce precedence amongst tasks, as we shall explain in Section 5.5.

5.4.3.3 Formal Description of TAFT-IPE

We now present a formal description of TAFT-IPE. For this purpose, we use an SDL-like syntax enhanced by some common mathematical notions to keep the description simple. At the heart of the description is the procedure *schedule*, which determines the task instance to be executed (*cur_task*). It is assumed that when *schedule* returns, the task referenced by *cur_task* is dispatched on the CPU. We assume that a set $\{\tau_i = (T_i, C_i, E_i) \mid i \in 1..n\}$ of periodic task pairs has been accepted and an EDL schedule for this set been computed offline. The acceptance criterion for the task pairs will be presented in Sub-Section 5.4.4. Two vectors describe the idle times of the EDL schedule for a single hyper period of length $H := \mathrm{lcm}(\{T_i \mid i \in 1..n\})$[6]: The vector $= (e_0, e_1, \ldots e_p)$ denotes the starting times of the idle time intervals, whereas the vector $\Delta = (\Delta_0, \Delta_1, \ldots, \Delta_p)$ denotes their lengths, where p is the number of idle times in the hyper period. As EDL schedules are cyclic, these two vectors are sufficient to describe all idle times in the whole schedule. We use the following notions in the formal description:

- Γ_S: the capacity of the server; for $i \in 1..n$, Γ_i is the capacity associated with τ_i; and $\mathcal{C} := \{\Gamma_S\} \cup \{\Gamma_i \mid i \in 1..n\}$ is the set of all capacities;

- *periodics, comms, aperiodics*: the set of ready instances of correct main and exception parts, communication tasks, and faulty main parts respectively waiting to be executed;

- *cur_ent*: a reference to the current entity (task or capacity);

- *cur_task:* a reference to the currently running task;

- *ed*(S): returns the entity with the shortest deadline in a set S of entities. Each capacity $\Gamma_i > 0$ is assigned a deadline $d_{i,k}$ during $[r_{i,k}, d_{i.k}]$, where $r_{i,k}$ and $d_{i.k}$ are the release time and deadline of the kth instance of τ_i. Ties are broken in the following order:

[6] "lcm" denotes the least common multiple of a set of numbers.

capacity, correct main part, faulty main part, and exception part. When *S* is empty, the function returns the idle task;

- *rem_i*: the remaining allocated execution time of the current instance $MP_{i,k}$ of τ_i's main part; it represents the difference between the specified execution time C_i and the processing time already assigned to $MP_{i,k}$. When $rem_i = 0$ and $MP_{i,k}$ is not completed it has become faulty;

- *cur_rem*: remaining execution time of the current entity *cur_ent*. If *cur_ent* is a capacity, *cur_rem* corresponds to the value of that capacity;

- *rnext*: the next time at which a periodic instance will be released;

- *request:* a signal arriving in the scheduler whenever an instance of a communication task is released;

- *completed:* a signal arriving in the scheduler whenever an instance completes;

- *accept*: the acceptance test for instances of the communication tasks, which will be presented in Sub-Section 5.4.4;

- *dispatch_time*: the time when the current task was dispatched to the CPU;

- *sched_timer*: the scheduling timer;

- *now*: the current clock time;

We organized the explanation of the formal description according the different themes of the algorithms:

Maintaining Capacities. During the start transition all capacities are initially set to zero (lines 2-3). The server capacity Γ_S is replenished whenever an idle time of the underlying EDL schedule starts (lines 13-14). Each such point of time coincides with the release time of a task instance. To replenish the server capacity, the length of the idle time is added to the server capacity. Whenever some instance has been executing under a capacity for Δt time units, the capacity is reduced by that amount of time during the accounting (line 62-63). If it was an instance of a main part, the capacity associated with that main part is increased accordingly (line 64-65). This means that the capacity is exchanged from *cur_ent* to Γ_i.

Monitoring the execution of main parts. Whenever a main part is released, the remaining execution time rem_i is initialized with the resource demand specified for that main part (line 10). The scheduler uses this variable to detect when a main part becomes faulty. All execution times of the main part are subtracted from the remaining execution time during the accounting (line 67-68). When rem_i reaches zero, the main part is converted into an aperiodic request and the exception part EP_i is added to the pending queue (lines 69-73). Before a main part is dispatched to the CPU, the scheduling timer is set to ensure that the execution of the main part is interrupted when the remaining execution time has been consumed (lines 42-43 and 48).

```
start;
    $\Gamma_s$ := 0; rnext := 0;
    for(i $\in$ 1..n) $\Gamma_i$ := 0;

input sched_timer;
  call accounting;
  if (now = rnext) {
    for (i $\in$ 1..n : now mod $T_i$ = 0) {
      periodics := add(instance_of($MP_i$),periodics);
      $rem_i$ := $C_i$;
    }
    rnext := min{($\lfloor now/T_i \rfloor$+ 1)$T_i$};
    if ($\exists$i $\in$ 0..p : now = $e_i$ + $\lfloor now/H \rfloor$H)
      $\Gamma_s$ := $\Gamma_s$ + $\Delta_i$;
  }
  call schedule;

input request(J);
  call accounting;
  if (accept(J)) comms := add(J,comms);
  call schedule;

input completed(J);
  remove J from the corresponding pending queue;
  if ($\exists$i : is_instance_of(J,$MP_i$)) {
    $\Gamma_i$ := $\Gamma_i$ + $rem_i$ + $E_i$; $rem_i$ := 0;
    if ($\exists$J' $\in$ periodics : is_instance_of(J',$EP_i$))
      periodics := remove(J',periodics));
  }
  call schedule;

procedure schedule;
start;
cur_ent := if $\Gamma_s$ > 0 then $\Gamma_s$
           else ed($\mathcal{C}$ $\cup$ periodics);
if (cur_ent $\in$ $\mathcal{C}$) {
  cur_task := if $\exists$J $\in$ comms then J
              else ed(periodics $\cup$ aperiodics);
  cur_rem := cur_ent; }
else {
  cur_task := cur_ent;
  cur_rem := if $\exists$i : is_instance_of(cur_task,$MP_i$) then $rem_i$
             else $\infty$ fi;
}
if ($\exists$i: is_instance_of(cur_task,$EP_i$))
  aperiodics := remove(instance_of($MP_i$),aperiodics);
dispatch_time := now
set(sched_timer,min{now + cur_rem,rnext});
return;
```

Figure 5-11. Formal description of the TAFT-IPE algorithm (part 1)

```
60 procedure accounting;
61 start;
62 if(cur_ent ∈ $\mathcal{C}$)
63   cur_ent := cur_ent - (now - dispatch_time);
64   if (∃i : is_instance_of(cur_task,$MP_i$))
65     $\Gamma_i$ := $\Gamma_i$ + (now - dispatch_time);
66 }
67 if (∃i:is_instance_of(cur_task,$MP_i$)){
68   $rem_i$ := max(0,$rem_i$ - (now - dispatch_time));
69   if ($rem_i$ = 0 and not cur_task ∈ aperiodics){
70     periodics := remove(cur_task,periodics);
71     aperiodics := add(cur_task,aperiodics);
72     periodics := add(instance_of($EP_i$),periodics);
73   }
74 }
75 return;
76 endprocedure accounting;
```

Figure 5-12. Formal description of the TAFT-IPE algorithm (part 2)

Scheduling. Deciding which task to execute is performed in two steps. First, the set of capacities and periodic instances is considered, and the highest priority entity is chosen as the current entity (lines 34-35). If this entity is a task instance, it is selected for execution (lines 40-44). Otherwise, it is a capacity and a task has to be selected to run under this capacity (lines 36-39). If a communication task is pending, it is selected. Else, the instance with the shortest deadline is chosen. This means that regarding execution under a capacity, the highest priority is given to the communication tasks. Among correct and faulty main parts with a common deadline, ties are broken in favor of correct ones for fairness reasons. If a main part and an exception part have the same deadline ties are broken in favor of the main part in order to increase the probability that the main part can be completed by its deadline. In fact, the exception part will not be executed under a capacity because each exception part is released together with an aperiodic request (the faulty main part) having the same deadline. So, the scheduler defers executing the exception part and tries to complete the main part first. If a faulty main part can be completed, the exception part is removed from the pending queue (lines 27-28). If, however, the scheduler starts executing the exception part, it aborts the main part (line 45-46).

Reclaiming Resources. To have even more capacities for executing aperiodic requests, we use a mechanism that converts resources that have been allocated for main parts and exception parts, but have not been used into aperiodic capacities. So, resources of main parts that do not need the full C_i and of exception parts that are not executed can be used to execute aperiodic requests. When a main part is completed, its remaining execution time and the time allocated for the corresponding exception part are transferred to its aperiodic capacity (line 26). This mechanism for reclaiming resources nearly adds no overhead to the scheduler.

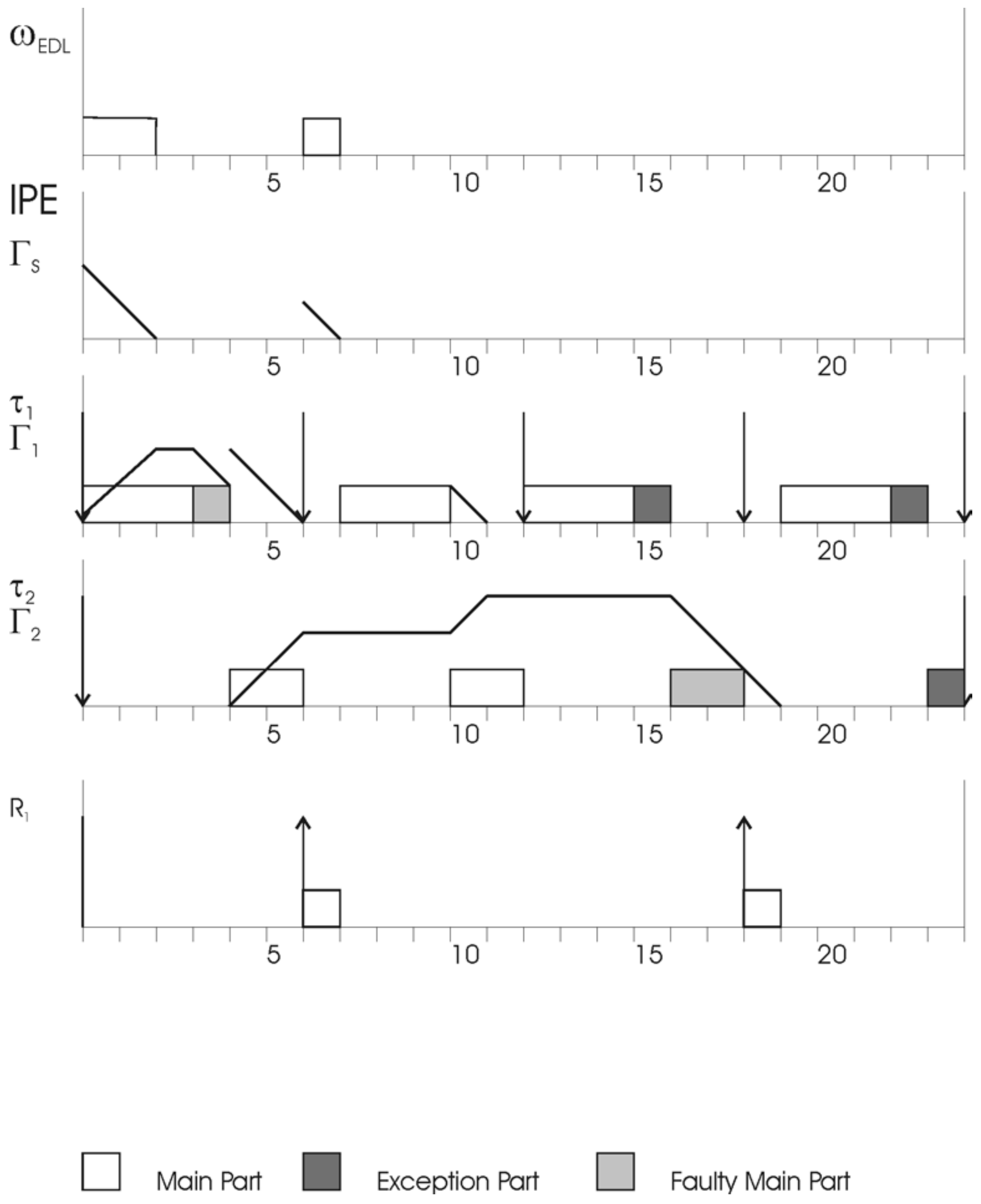


Figure 5-13 Example of a TAFT-IPE schedule

Figure 5-13 shows an example of a TAFT-IPE schedule. The example contains two periodic task pairs τ_1 := (6,3,1) and τ_2 := (24,4,1) and a communication task R_1 := (1,3). From top to bottom, the graphs depict the idle times of the underlying EDL schedule (ω_{EDL}), the capacity of the IPE server (Γ_S), and the states and capacities of both tasks. In the example, the execution times of the exception parts and of the communication task are not that much shorter than the execution time of the main parts as we suppose they would be in reality; yet, we decided not to make them to small so that the figure remains easy to comprehend. The actual required execution times of the periodic task instance are assumed to be $C_{1,1}$ = 4, $C_{1,2}$ = 3, $C_{1,3}$ = 4, $C_{1,4}$ = 5, and $C_{2,1}$ = 7. At the start, the initial capacity Γ_S = 2 is the highest priority entity and the task with the shortest deadline (τ_1) is selected to run under that capacity, meanwhile accruing a capacity Γ_1. At time 2, Γ_S is exhausted and Γ_1 becomes the highest priority entity. Task τ_1 runs under that capacity until time 3 when it becomes faulty and is turned into an aperiodic request. This request is selected for execution under the capacity, for it is the task instance with the shortest deadline. At time 4, it is completed and the time allocated for its exception part is added to Γ_1, which is still the highest priority entity. Task τ_2 runs under that capacity until time 6 and accrues a capacity $\Gamma_2 = 2$. At time 6, the server capacity Γ_S is replenished, another instance of τ_1 is released, and an instance of the communication task arrives. Γ_S is the highest priority entity at that time, and according to our policy, the communication task is selected to run under that capacity. So, by time 7, Γ_S is exhausted and the communication task completed. Now, the highest priority entity is τ_1, which is executed consequently. When τ_1 is completed at time 10, the time allocated for its exception part is added to capacity Γ_1, which at once becomes the highest priority entity. The execution of τ_2 is continued for 1 time unit under Γ_1 so that Γ_2 increases to 3. From time 11 to 12, Γ_2 is the highest priority entity and τ_2 is executed under it until it becomes faulty. At the same time, the next instance of τ_1 is released. This instance is now the highest priority entity and is executed until time 15, when it becomes faulty. Its exception part, which is added to the pending queue at that moment, immediately becomes the highest priority entity and is executed accordingly. After the exception part is completed, Γ_2 becomes the highest priority entity again. The only pending task instance is the faulty main part of τ_2, which is executed under the capacity until time 17. At that time, the next instance of τ_1 is released and another aperiodic request arrives. As Γ_2 is still the highest priority entity, the aperiodic request is executed under it and completed by time 19. After Γ_2's being exhausted, τ_1 is selected for execution. At time 22, when it becomes faulty, its exception part is added to the pending and immediately executed. After the exception part is completed, the exception part of τ_2 is run.

This example illustrates how the mechanism of priority exchange allows preserving the capacities in the system: The last time a new capacity is added is at time10, when τ_1 is completed and the time allocated for the execution of its exception part is turned into a capacity. The capacities are still available from time 16 to 18 to execute the faulty main part of τ_1 and at time 18 to execute the communication task. This instance of the communication task is served immediately, even though the last capacity was added at time 10. Actually, the last replenishment of the server capacity occurred even earlier at time 6.

5.4.4 Acceptance Test

In this sub-section we provide acceptance tests for both periodic task pairs and aperiodic requests. For periodic task pairs, acceptance testing is performed offline for the whole task set. For aperiodic requests, acceptance testing is performed online on a per instance basis.

5.4.4.1 Acceptance Test for Periodic Task Pairs

Lemma 5-1 *TAFT-IPE guarantees timely completion of correct jobs and exception handling for a set* $\mathcal{T} := \{\tau_i = (T_i,C_i,E_i) \mid i \in 1..n\}$ *of periodic task pairs if the set* $\mathcal{T}' := \{\tau'_i := (T_i,C_i + E_i) \mid i \in 1..n\}$ *is schedulable with the IPE server.*

Proof. Let $\mathcal{T} := \{\tau_i = (T_i,C_i,E_i) \mid i \in 1..n\}$ be an arbitrary set of periodic task pairs for which the IPE server feasibly schedules $\mathcal{T}' := \{\tau'_i := (T_i,C_i + E_i) \mid i \in 1..n\}$, and let

$\{J_{i,k} \mid i \in 1..n, k \geq 1\}$ be an arbitrary set of instances of the task pairs in $\mathcal{T}$. We define for each $i \in 1..n$ and $k \geq 1$ a job $J'_{i,k}$ that has the same release time and deadline as $J_{i,k}$ and an actual execution time $C'_{i,k}$ defined as follows:

$$C'_{i,k} := \begin{cases} C_{i,k}, C_{i,k} \leq C_i \\ C_i + E_{i,k}, C_{i,k} > C_i \end{cases},$$

where $C_{i,k} > 0$ and $E_{i,k} > 0$ are the actual execution times of the main part and exception part of $J_{i,k}$ respectively. Since for all $J'_{i,k}$, $C'_{i,k} \leq C_i + E_i$, they can be considered as the instances of the tasks $\tau'_i \in \mathcal{T}'$. Hence, the IPE server feasibly schedules the job set $\{J'_{i,k} \mid i \in 1..n, k \geq 1\}$ for any additional load of aperiodic requests. Let σ' denote the schedule that the IPE server produces for that job set and σ be the schedule produced by TAFT-IPE, both for the same aperiodic load. We show that σ meets each of the three conditions of Definition 5-1.

First, note that at the same time when $EP_{i,k}$ is added to the pending queue, $MP_{i,k}$ is turned into an aperiodic request (lines 70-72) so that throughout the ready time of $J_{i,k}$ exactly one of both is part of the pending queue as a periodic instance. As furthermore $J_{i,k}$ and $J'_{i,k}$ have the same release time and deadline by construction, it follows that $MP^P_{i,k}$ (denoting $MP_{i,k}$ as long as it is periodic) and $EP_{i,k}$ get as much resources in σ as $J'_{i,k}$ in σ', and get them by the same time as well. Actually, since we do not generally give highest priority to aperiodic requests under capacities, periodic instances may be served even earlier in σ than in σ'. Therefore we state that for each time t prior to the completion of either $EP_{i,k}$ or $MP_{i,k}$

$$c_\sigma(MP^P_{i,k},t) + c_\sigma(EP_{i,k},t) \geq c_{\sigma'}(J'_{i,k},t) \quad (1)$$

where for a schedule σ an entity e and a time t, $c_\sigma(e,t)$ denotes the time allocated to e in σ by time t. Furthermore, since $EP_{i,k}$ is inserted into the pending queue and $MP_{i,k}$ turned into an aperiodic request immediately after running $MP_{i,k}$ for C_i, we observe that

$$\forall t\ c_\sigma(MP^P_{i,k},t) + c_\sigma(EP_{i,k},t) \geq C_i \Rightarrow c_\sigma(MP^P_{i,k},t) = C_i \qquad (2)$$

For each instance $J_{i,k}$, we distinguish two cases:

1. $C_{i,k} \leq C_i$. It follows that $C'_{i,k} = C_{i,k}$. Due to the feasibility of σ', $\exists t \leq d_{i,k} : c_{\sigma'}(J'_{i,k},t) = C'_{i,k}$. Together with (1) and (2), this implies that $MP_{i,k}$ completes no later than t in σ. Furthermore, this implies that $EP_{i,k}$ is never put into the pending queue. So, the conditions (1) – (3) of Definition 5-1 are fulfilled.

2. $C_{i,k} > C_i$. It follows that $C'_{i,k} > C_i$. Due to the feasibility of σ', $\exists t \leq d_{i,k}$: $c_{\sigma'}(J'_{i,k},t) = C_i$. Together with (1) and (2), this implies that $\exists t' \leq t$: $c_\sigma(MP_{i,k},t') = C_i$. At t', $MP_{i,k}$ is turned into an aperiodic request and $EP_{i,k}$ is added to the pending queue. Due to the feasibility of σ', there is a first time $t'' < d_{i,k}$: $c_{\sigma'}(J'_{i,k},t) > C_i$. Due to (1) and (2), $\exists t^+ \leq t''$ such that either $MP_{i,k}$ is completed before t^+ or $c_\sigma(EP_{i,k},t^+) > 0$.

    a. In the first case, $MP_{i,k}$ is completed before $t^+ < t'' < d_{i,k}$. Furthermore, at that time, $EP_{i,k}$ is removed from the pending queue, so that $\forall x > t^+$: *not running*$(EP_{i,k},x)$.

b. In the second case, t^+ is the start time of $EP_{i,k}$. Since $MP_{i,k}$ is aborted when $EP_{i,k}$ starts running, it follows that *not completes*($MP_{i,k}$) and $\forall x > t^+$: *not running*($MP_{i,k}$, x). Furthermore, due to the feasibility of σ', $\exists t^* \leq d_{i,k} : c_{\sigma'}(J'_{i,k},t^*) = C'_{i,k} = C_i + E_{i,k}$, which together with (1) and (2) implies that $EP_{i,k}$ is completed by t^*.

In both cases, a. and b., the condition (1) – (3) of Definition 5-1 are fulfilled.

■

Theorem 5-2 *TAFT-IPE guarantees timely completion of correct jobs and exception handling for a set* $\mathcal{T} := \{\tau_i = (T_i,C_i,E_i) \mid i \in 1..n\}$ *of periodic task pairs if the utilization factor*

$$U := \sum_{i=1}^{n} \frac{C_i + E_i}{T_i} \leq 1 \ .$$

Proof The theorem is a direct implication of Lemma 5-1 and the schedulability condition of the IPE server (Stankovic et al. 1998 p. 189).

■

5.4.4.2 Acceptance Test for Aperiodic Requests

The acceptance test is performed for each instance of a communication task to provide predictability on a per request basis. After passing the acceptance test, a request is guaranteed to be completed by its deadline; requests that do not pass the acceptance test are not executed. This ensures that the execution service does not exhibit timing failures, thus meeting the assumptions of our system model (cf. Section 4.2).

As the acceptance test is performed for every instance of a communication task it must be efficient. Therefore, instead of devising a complex exact test, we provide a sequence of simple sufficient criteria. These criteria are evaluated in order of increasing complexity. As soon as the first sufficient criterion is true, the request is accepted. If none of the criteria evaluates to true, the request must be rejected, since its timely completion cannot be guaranteed based on the acceptance test.

In the presentation of the criteria, C and D are the worst-case execution time and relative deadline of the request respectively, and $d := t + D$ is its absolute deadline, where t is the time at which the test is performed. Each of the following tests decrements the value of C. They require that C after being decrementing is less than or equal to zero, which means that there are sufficient capacities in the system to complete the request by its deadline.

1. *If the current entity 'cur_ent' is a capacity* Γ *and* $C - \min(\Gamma, rnext - t) \leq 0$;

   If the current entity is a capacity, it is allocated to the request, at least until either the next task instance is released ($rnext - t$) or the capacity is exhausted (Γ).

2. $C'_1 := C - \Gamma_S \leq 0$;

   If the server has a capacity at time *t*, it can be allocated to the request immediately.

3. $C'_2 := C'_1 - \sum_{d_i \leq d} \Gamma_i \leq 0$, *where d_i is the deadline associated with Γ_i;*

   All capacities with deadlines not greater than *d* can be allocated to the request.

4. $C'_3 := C'_2 - \sum_{t < H' + e_i < d} \min(\Delta_i, d - H' - e_i) \leq 0$, *where $H' := H \lfloor t / H \rfloor$ is the starting time of the current hyper period;*

   The request can use all server capacity replenishments that take place during [*t*,*d*], but only during [$H'+e_i$,*d*], where $H'+e_i$ is the time at which the replenishment occurs.

In the actual implementation, rather than computing the sum first and then subtracting it from *C*, it is better to iteratively decrement *C* and stop as soon as the criterion is verified. As we assume that *C* is small as compared to the execution times and periods of the periodic tasks, we suppose that one of the first criteria will yield a decision in many cases.

A practical approach to increase the probability that a request can be accepted is to give the communication tasks a higher priority than the main parts, but a lower priority than the exception parts. This means that the communication tasks can run not only under capacities, but also during the times allocated for the main parts. The price to be paid for this improvement is losing the guarantee that main parts are completed as long as they do not exceed their specified resource demand. This may be acceptable for certain applications for the following reasons:

- The fact that main parts may not be completed does not raise a new systematic problem, but is already part of the underlying concept. So, the suggested approach has mainly a quantitative impact in that it reduces the probability that main parts are completed. In particular, a timely completion of the exception parts would still be guaranteed.

- The execution times of the communication tasks are small as compared to those of the application tasks. Hence, even if the same main part is preempted several times, this will not lead to a significant reduction of its allocated execution time. Therefore, the mentioned quantitative impact appears to be limited.

5.5 Precedence Constraints

In our sensor fusion scenario (Section 3.2), the processing of the sensor data is structured as a pipeline of filtering stages. We believe this will be frequently the case for applications dealing with complex sensor data, like scan or images. The tasks executing the stages of such a pipeline are subject to *precedence constraints*, which means that a successor stage must not start executing before its predecessor has delivered its results. For example, in the sensor fusion, the element filter should not start executing before the object filter has delivered its results. The whole pipeline has a common period and deadline by which the final stage of the pipeline must have been finished and deliver its results.

One may question why not realizing all the stages in a single task, thus eliminating the dependencies from the task set. To understand why this is not appropriate for the kind of processing we consider, imagine that one of the first processing stages required significantly more execution time than was expected. It would consume the resources actually meant to execute its successor stages, and the scheduler would have no chance to prevent it from doing so. In the end, when the execution time allocated to the whole pipeline is consumed, processing may not have proceeded to the final stage so that no result may be available. Structuring the pipeline as a sequence of task pair's, the expected-case execution time of the pipeline is distributed among the stages. TAFT ensures that each stage, from the first to the last, gets at least its allocated ECET. Thus, an intermediate stage requiring more than the allocated resources cannot "steal" the execution times allocated for the other stages in the pipeline.

In what follows, precedence constraints can be accommodated in TAFT-IPE. At first, we explain how precedence constraints are modeled and then show how task pairs with precedence constraints are scheduled.

5.5.1 Model

To model sets of task pairs with precedence constraints, we adapt a model presented in (Stankovic et al. 1998). We model the precedence constraints between instances of task pairs as an ordering relation $\prec$ on the set of instances, where for two task pair instances I and J, $I \prec J$ means that I must be finished before J is started. The conditions that a schedule must fulfill to be compliant with a precedence constraint are formalized in the following definition.

Definition 5-2. *A schedule σ for a set $\mathcal{I}$ of task pair instances is compliant with the precedence constraint $\prec \subseteq \mathcal{I} \times \mathcal{I}$ if and only if for any two instances $J_i, J_j \in \mathcal{I}$, $J_i \prec J_j \Rightarrow f_\sigma(J_i) < s_\sigma(J_j)$, where $f_\sigma(J_i)$ is the last point of time t at which $running_\sigma(MP_i,t)$ or $running_\sigma(EP_i,t)$, and $s_\sigma(J_j)$ is the first point of time t at which $running_\sigma(MP_j,t)$ or $running_\sigma(EP_j,t)$*

A pipeline P_i is modeled as a 3-tuple $P_i = (\mathcal{T}_i, \prec_i, T_i)$, where $\mathcal{T}_i$ is a set of periodic task pairs $\{\tau_{i,j} = (T_i, C_{i,j}, E_{i,j})\}$, $\prec_i$ a total order on $\mathcal{T}_i$, and T_i the period of P_i. This means that

- At each time $t = (k-1)T_i$, $k \geq 1$, and for each task $\tau_{i,j}$ in $\mathcal{T}_i$, an instance $\tau_{i,j,k}$ is released;
- The instances must be executed according to $\prec'_i := \{(J_{i,l,k}, J_{i,m,k}) \mid k \geq 1, \tau_{i,l} \prec_i \tau_{i,m}\}$;
- All instances must finish by kT_i.

Definition 5-3. *A schedule σ for a set of n pipelines $\{P_i = (\mathcal{T}_i, \prec_i, T_i) \mid i \in 1..n\}$ is compliant with $\prec_i$, $i \in 1..n$, if it is compliant with the precedence constraint*

$$\prec' := \{(J_{i,l,k}, J_{i,m,k}) \mid i \in 1..n,\ \tau_{i,l} \prec_i \tau_{i,m},\ k \geq 1\}.$$

5.5.2 Extending TAFT-IPE

To ensure that TAFT-IPE produces schedules that are compliant with the precedence constraint of a pipeline P_i, we assign priorities to the tasks of that pipeline according to the relation $\prec_i$; that is, the highest priority is assigned to the first and the lowest priority to the last task w.r.t $\prec_i$. These priorities are pipeline internal in that they are used only to discriminate between tasks belonging to the same pipeline. In TAFT-IPE, the pipeline-internal priorities impact how the highest priority entity and the task instance to run under a capacity are determined. For both, the pipeline-internal priorities of the instances are added as last criterion. As all pending task instances belonging to the same pipeline P_i have the same deadline, this ensures that among these instances, the first instance w.r.t $\prec_i$ takes precedence over all its successors. The schedulability for the resulting, modified TAFT-IPE is stated in the following theorem:

Theorem 5-3. *For a given set $\{P_i = (\mathcal{T}_i, \prec_i, T_i) \mid i \in 1..n\}$ of pipelines, TAFT-IPE produces a schedule σ that is compliant with $\prec_i$, $i \in 1..n$, and achieves timely completion of correct jobs and exception handling if*

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1, \text{ where } C_i = \sum_{\tau_j \in \mathcal{T}_i} C_{i,j} + E_{i,j}$$

Proof Neglecting the precedence constraints, TAFT-IPE produces a schedule that achieves timely completion of correct jobs and exception handling if the above condition holds because

$$U = \sum_{i=1}^{n} \sum_{\tau_{i,j} \in \mathcal{T}_i} \frac{C_{i,j} + E_{i,j}}{T_i} = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \ .$$

Let σ' be this schedule. The schedule σ' can be transformed into the schedule σ produced by TAFT-IPE with precedence constraints by just exchanging the execution times of released instances having the same deadline. Note furthermore, that the suggested priority assignment does not effect the ordering of the main and exception part of a single task pair. Hence, the resulting schedule still guarantees completion of correct jobs and exception handling. It remains to show that σ is compliant with the precedence constraint. Consider two instances $J_{i,l,k} \prec' J_{i,m,k}$. At their release time, the main parts $MP_{i,l,k}$ and $MP_{i,m,k}$ of both instances are added to the pending queue for periodic instances. Since both have the same deadline, and $MP_{i,l,k}$ has the higher pipeline-internal priority, $MP_{i,m,k}$ is not started as long as $MP_{i,l,k}$ is in the pending queue for periodic instances. If the scheduler removes $MP_{i,l,k}$ from this queue, this has one of the following reasons:

1. $MP_{i,l,k}$ completed. In this case, the finishing time of $J_{i,l,k}$ is smaller than the starting time of $J_{i,m,k}$, and the constraint is fulfilled.

2. $MP_{i,l,k}$ became faulty. In this case, the exception part $EP_{i,l,k}$ of $J_{i,l,k}$ is added to the pending queue for periodic instances. Following the same reasoning as above, $MP_{i,m,k}$ is not executed as long as $EP_{i,l,k}$ is pending. When $EP_{i,l,k}$ is removed from the pending queue, it follows that either $EP_{i,l,k}$ or the faulty main part $MP_{i,l,k}$ completed. In both cases, the finishing time of $J_{i,l,k}$ is smaller than the starting time of $J_{i,m,k}$ and the constraint is fulfilled.

■

Actually, the algorithm can handle more general precedence relations than just total orders, but for the time being, we focus on those kinds of precedence constraints which are required to model the kind of pipelined processing we have in mind. Furthermore, it is possible to transform the combined deadline- and priority-based scheduling back to a pure deadline-based scheduling (Stankovic et al. 1998).

6 Prototypes and Implementations

In this chapter, we present the prototypes of our two application scenarios. The first is a prototype of the shared spatial resource scenario and is intended to show the viability of our approach to the coordination of mobile embedded system. Furthermore, it allows gaining some first indications about the response times that can be achieved using the communication services of the middleware. The second is a prototype of the distributed sensor fusion scenario. As compared to the first prototype, it has the more demanding CPU load and hence was used to show the viability of our approach to the provision QoS in dynamic environments. It allows applying and evaluating TAFT in a real application, which exhibits unpredictable execution times and several kinds of application-inherent redundancy. Finally, we present a modular implementation of the communication hardcore, which allows configuring protocol stacks at compile time. With this implementation, we want to show that the concept of modularity, which guided our architectural design and formal modeling, can be transformed into a modular, yet highly performant implementation.

6.1 Prototype of the Shared Spatial Resources Scenario

In order to demonstrate the viability of our approach to the coordination of mobile systems in a real application, we build up an application prototype of the shared spatial resources scenario (Nett and Schemmer 2003b). In this prototype, a group of mobile robots coordinate their speeds at a shared resource based on the common views the middleware provides and on a scheduling function that each robot computes locally. Furthermore, the prototype allows measuring the response times of a cooperative application that uses the communication services of the middleware under realistic traffic patterns and assessing the results against the timing constraints the prototype imposes.

6.1.1 Description of the Prototype

The application prototype was realized according the application architecture presented in Chapter 3. In the prototype, a group of mobile robots uses the communication services of the middleware to coordinate their access to a shared spatial resource. The robots are driving along traces that form two overlapping loops (see Figure 6-1). The front part of the intersection of the two loops represents the hot spot. The length of the hot spot was chosen in such a way that a given safety distance was maintained between the robots. This setup corresponds to the model described in Section 3.1; we only placed one of the loops inside the other to use the space available in our laboratory more efficiently. On each loop, there is an approaching zone starting at a certain distance in front of the hot spot. Robots detect their entering an approaching zone through markings. When a robot passes a marking, it calls the Event Service to trigger a global rescheduling of the shared resource. As parameters of the call, it provides its current position and velocity to the Event Service. The Event Service propagates the event to all robots that are approaching the hot spot plus the single robot possibly located within it. Associated with the event, the Event Service delivers the global state, which consists of the positions and velocities of all the robots w.r.t the same point of time. Upon delivery of the event, the robots compute the scheduling function based on the global state.

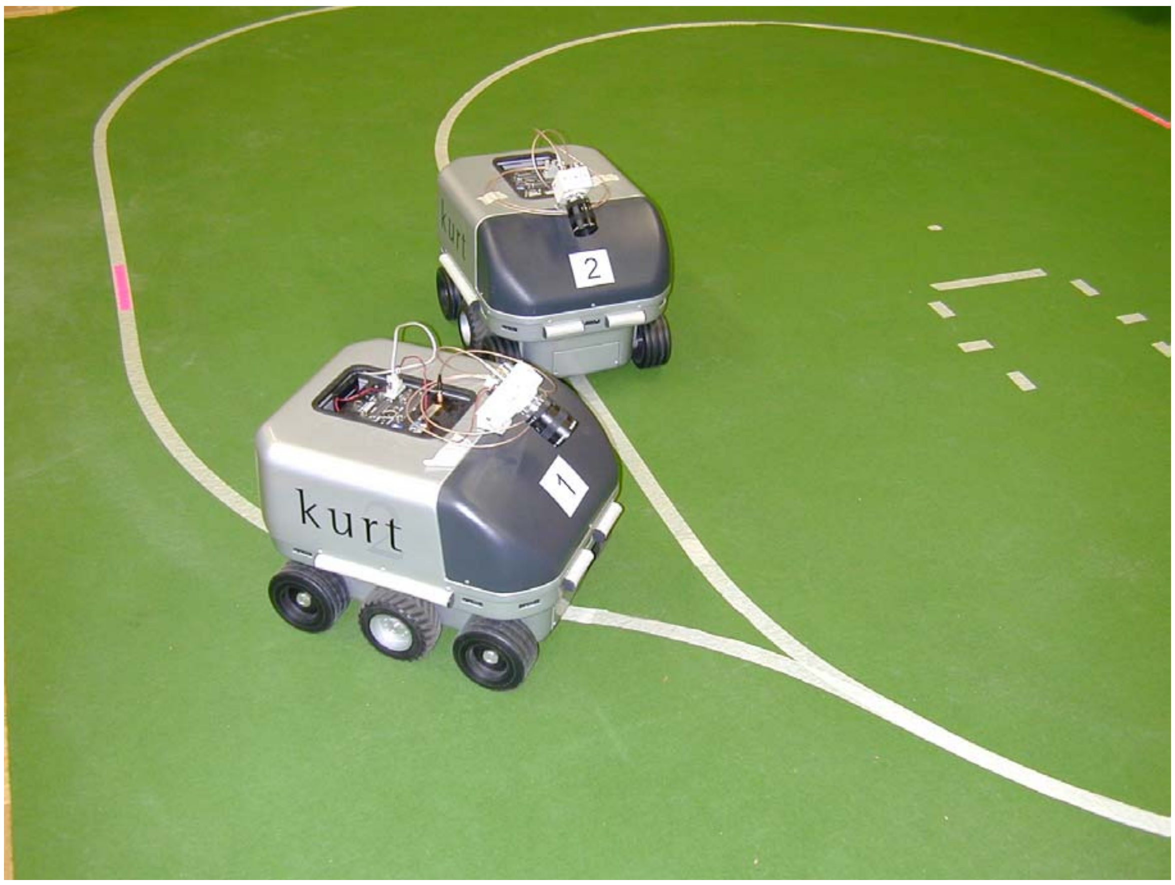


Figure 6-1. Picture of the application prototype

The prototype demonstrates the feasibility of our approach to the coordination of mobile systems. There are no interactions between the robots on the application level. Rather, the schedule for the hot spot is computed locally at each robot as a function of the global state. In designing this scheduling function, we were able to concentrate on the application-specific optimization objectives, such as achieving a smooth driving of the robots and high utilization of the hot spot. The concrete scheduling function we used was presented in (Schemmer and Nett 2001). Still, the robots demonstrate a coordinated behavior at the hot spot. During extended experiments with the prototype, the mobile systems passed the hot spot without collisions by adapting their speeds in the approaching zones.

6.1.2 Measurements

We measured the end-to-end response times of the application architecture; this is to say, the time that elapses from the moment in which a marking has been detected to the moment in which the new schedule has been computed. To measure the response times, a robot takes a timestamp when it detects a marking and a second timestamp when it has computed the new schedule. The difference of the two timestamps yields the response time. Figure 6-2 shows the results of our measurements. The response times have been measured in a group of three mobile robots equipped with PC104 computers (AMD K6-266, Windows NT) and connected by an IEEE 802.11 Standard wireless LAN (2 Mbit/s for broadcast transmission). The parameters of the communication protocols were set as follows: *OD* was set to 15, the resiliency equal to *OD*, and the poll timeout to_{Poll} to 20ms. Because of the limited physical extend of the prototype, a static group has been considered.

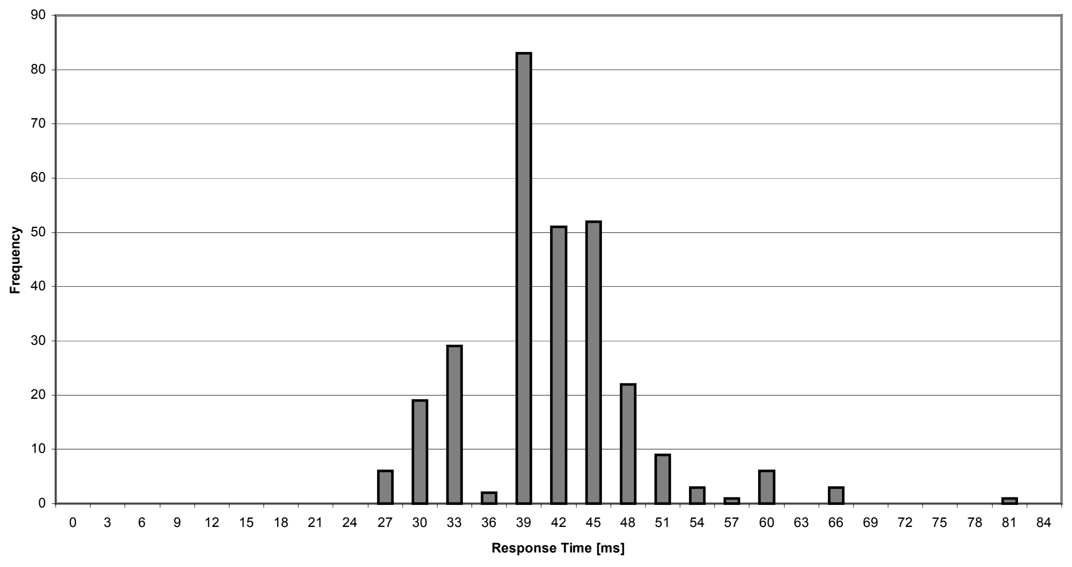


Figure 6-2. Measured delay distribution of the architecture.

The measured response times are distributed around a mean value of 42.76ms with a maximum of 81.90ms. It appears that the measured response times are acceptably small. For example, assume that a robot is driving at 3m/s and that the approaching zones have a length of 3m. In this setting, the robot must start braking no later than 0.6s after entering the approaching zone to be able to stop in front of the hot spot (considering a brake retardation of 3.79m/s^2). The measured response times are much smaller than this latest reaction time. How the response times scale with an increasing number of mobile systems in the approaching zones can be approximated as follows. The delay of the atomic multicast service is approximately proportional to the duration of a communication round and hence to the number of polled stations (cf. Sub-Section 4.3.6). This also applies to the response times of the whole application, which are mainly determined by the delay of that service.

To give an idea, for a number of 5, 10, or 20 robots scaling the maximum (mean) measured response time yields an approximated response time of 136.5ms (71.25ms), 273ms (142ms), or 546ms (285ms) respectively. All these values are still under the above stated bound of 0.6s. Additionally, it should be taken into account that with a number of 20 robots, the approaching zones would be a good deal larger. This indicates that the communication services of the middleware are able to meet the performance requirements of the prototype, even with their first, WindowsNT-based implementation. It can be expected that with the current, RTLinux-based implementation (see Section 6.3) an even better performance can be achieved in the next version of the prototype.

6.2 Prototype of the Distributed Sensor Fusion Scenario

In order to evaluate our approach to the provision of QoS in a dynamically changing environment, we build up an application prototype of the distributed sensor fusion scenario (Feist 2002,Nett and Schemmer 2003a,Schemmer and Nett 2003a). The prototype comprises several sensor data processing tasks, each with environment-dependent execution times. So, in this prototype, we can apply and evaluate the TAFT concept in an application with unpredictable resource demands of the tasks. Furthermore, the application prototype allows examining how application-inherent redundancy can be exploited to tolerate task abortions (Section 5.3).

6.2.1 Description the Prototype

The application prototype was implemented according to the application architecture presented in Section 3.2. In the prototype, three laser scanners observe a dynamically changing scene from different angles. A RoboCup-like scenario was chosen as an example for the prototype. It consists of a rectangular field enclosed by four boards. Within the field, there is a ball and rectangular objects representing the robots. The data delivered by the laser scanners are filtered to a certain level of abstraction, exchanged within the group using the atomic multicast service, fused, and then possibly filtered again until they reach the element level where they are represented as real world elements; that is, robots, boards, and balls with their positions. The rectangular objects and the ball are being moved within the boards so that the observed scene is dynamically changing. A visualization tool has been developed that allows monitoring the results of the distributed fusion online on different levels of abstraction (see Figure 6-3). As well, the results can be stored for purpose of offline analysis. Whereas the non-time-critical visualization tool is running as a Linux user-space application, the time-critical parts — that is to say, the filter and fusion stages — have been implemented in an RTLinux kernel module.

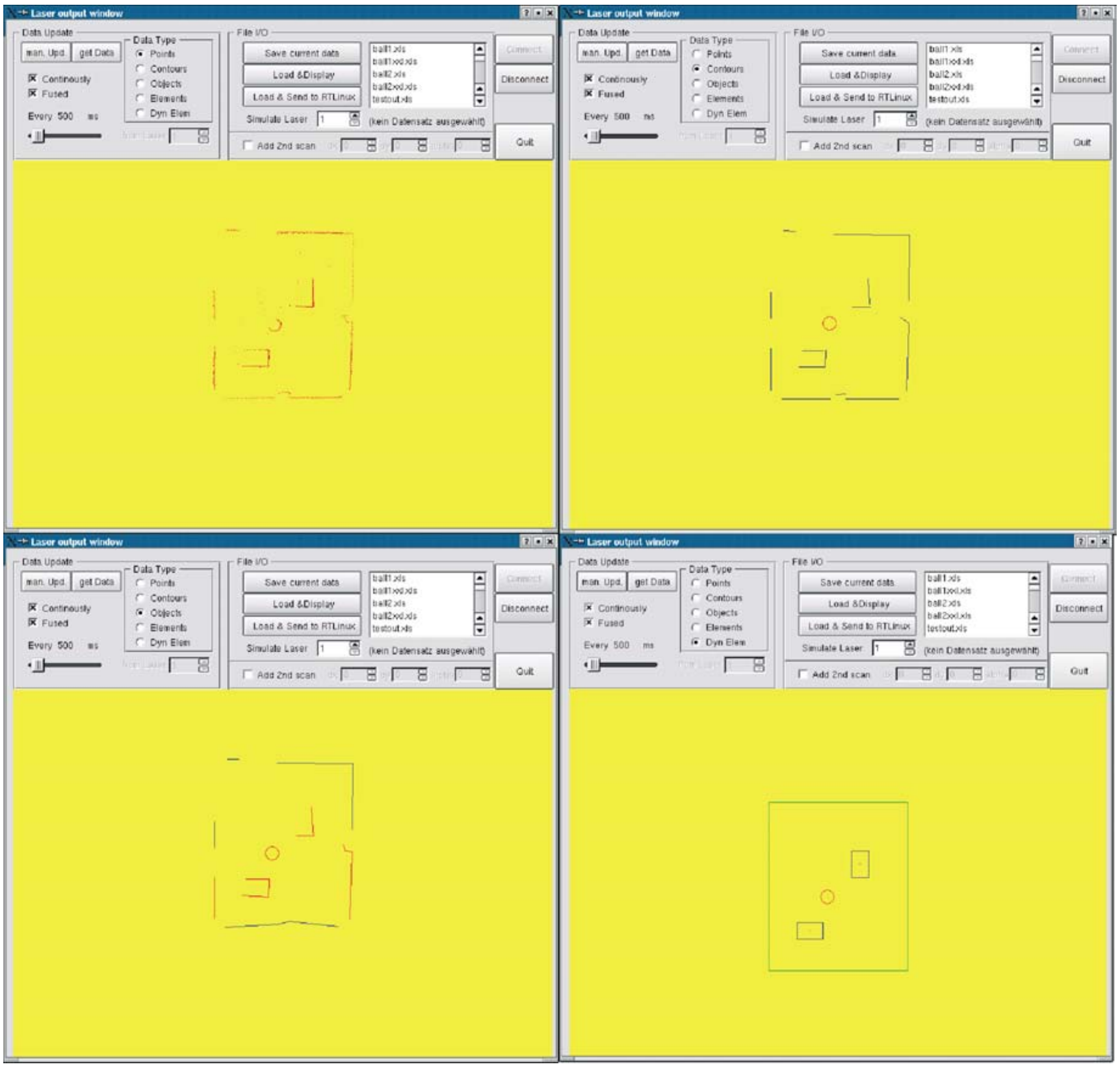


Figure 6-3. Screenshots from the distributed sensor fusion application

Figure 6-3 shows screenshots from the visualization tool. Each screenshot shows the same scene represented on a different levels of abstraction. Proceeding from the top left to the bottom right screenshot, the levels of abstraction are raw data, contours, objects, and elements. In its current status, the prototype allows fusion on any level of abstraction. So, execution time measurements and fusion results could be obtained for sensor fusion on different levels of abstraction. Dynamically changing the level at runtime has not yet been implemented.

Each filtering and fusion stage has been implemented as a separate task pair. The task pairs exchange data locally through a common buffer, which can be accessed by all task pairs. To communicate the data over the network, an additional stage was added to the pipeline. It reads the data from the buffer, encapsulates them in a message, and multicasts the message using the atomic multicast service. Since an implementation of TAFT-IPE was not yet available when the first version of the prototype was built up, we used the existing TAFT implementation to schedule the task pairs. To accommodate the precedence constraints in this implementation, executing the pipeline has been divided into a sequence of phases with fixed lengths (see Figure 6-4). Each task pair realizing a stage of the pipeline is executed during the corresponding phase. Its release time is equal to the starting time of the phase and its deadline is equal to the finishing time of the phase. Thus, the release time of the successor stage is equal to the deadline of its predecessor. To accomplish this setting of the release times, all task pairs have the same period but a different initial starting time; that is, the task pairs are started successively, each task pair at the deadline of the first instance of its predecessor. Once the task pairs have been phased this way, they can be released periodically.

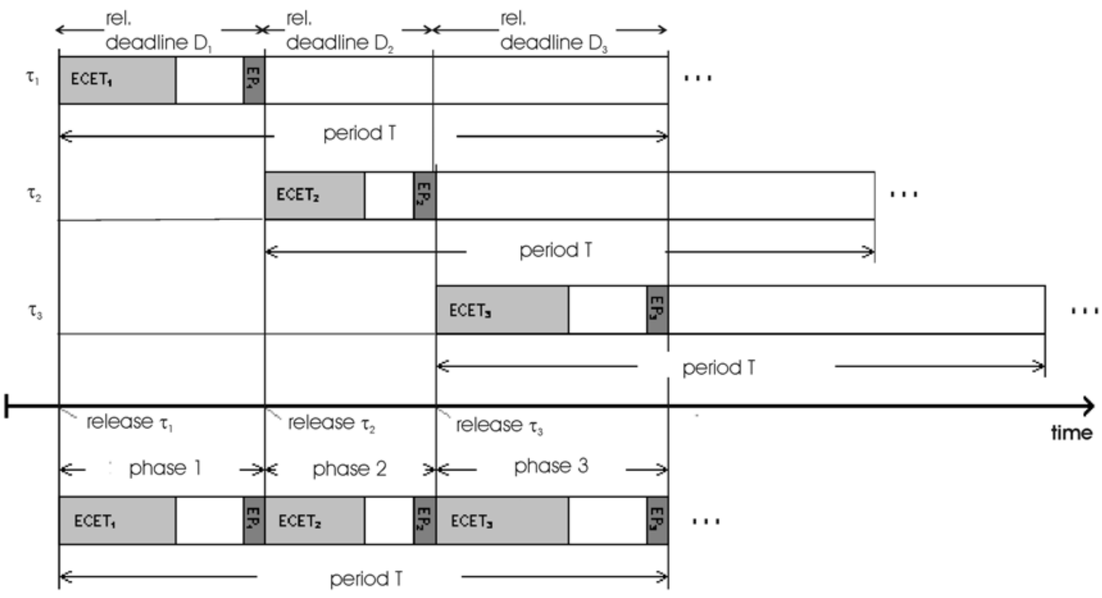


Figure 6-4. Phase structure of the distributed sensor fusion

Two approaches to achieve time coherence of the data to be fused have been implemented. The first approach is using a Kalman filter (Bar-Shalom and Fortmann 1988) to fuse data observed at different points of time. To give an idea how this works, let us consider fusion on the element level. Assume a mobile system already has a position estimate of some element in its environment, say the ball, which was determined at time *t*. When receiving a scan with timestamp *t′* that contains a new position estimate for the ball, the system fuses the received data with the existing position estimate in two steps. First, the filter uses a model to predict the position of the ball at time *t′* based on the old position estimate. A Gaussian distribution models the uncertainty of this prediction. Since after the prediction step both position estimates relate to the same time *t′*, they can be fused into a single, more accurate position estimate. As in our scenario, a system receives scans from several other systems, it sorts the scans it receives in order of increasing timestamps and successively fuses them with its current world model. This approach does not require all the scans to be recorded at the same point of time. However, it cannot be used on all levels of abstraction. Using the Kalman filter requires finding for each element in the current scan the corresponding element in the current world model. For example, if a mobile system receives a scan including position estimates for five robots and there are five robots in its current world model, each robot in the scan must be associated with a robot in the world model so that the position estimates of the associated robots can be fused as described above. While finding these associations is typically possible on the element level, it surely is not on the point level. In the second approach, the starting times of the periods of the above-mentioned phase structure (Figure 6-4) are synchronized at all systems using the clock synchronization service. This ensures that the data to be fused are time coherent right from their observation. Furthermore, it ensures that all scans a fusion stage receives are from the same period. This approach is used for fusion on low levels of abstraction — points and contours — where Kalman filters cannot be applied.

6.2.2 Measurements

First, we conducted measurements to analyze the execution times of the filtering and fusion stages (we use the common term module in the following) and how they depend on the environment. We already presented these measurements in Chapter 5, where they served to corroborate our problem exposition. We then analyzed the timing behavior TAFT achieves for such loads (Clause 6.2.2.1) and how task abortions can be tolerated by exploiting the application-inherent redundancy (Clause 6.2.2.2). Finally, we conducted measurements to illustrate the concept of application-level adaptation in case of persistent overloads (Clause 6.2.2.3).

6.2.2.1 Timely Predictable Execution

Figure 6-5 shows the response times of the arc filter under TAFT scheduling. The response times were measured in the same setting that was used for measuring the execution times presented in Section 5.1; that is, the same sequence of scans and the same hardware was used (AMD Athlon 700MHz CPU with 128MB memory). The arc filter was executed at a frequency which was two times the frequency used during the execution time measurements, so each scan was processed twice.

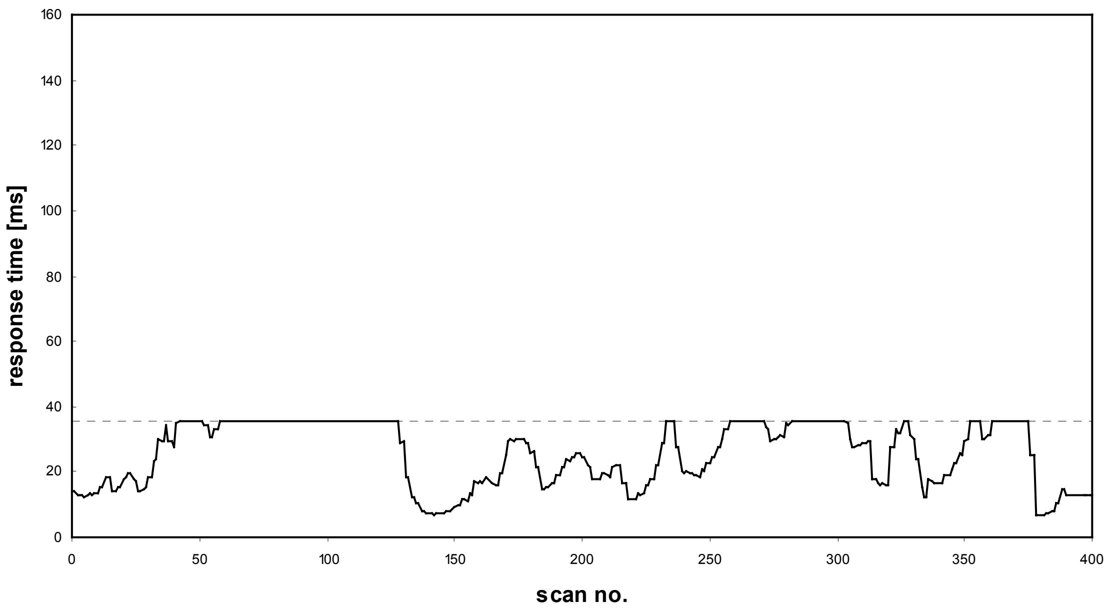


Figure 6-5. Response times of the arc filter

As can be seen in Figure 6-5, TAFT ensured that no instance of the arc filter missed its deadline (depicted by the dashed, straight line) during the measurements. The relative deadline of the task pair set to 35,82ms, the ECET of the main part to 30ms, and the WCET of the exception part to 0,2ms. The values were intentionally chosen to be quite small as compared to the measured execution times, so as to have many situations in which the main part cannot be completed before its deadline and the TAFT scheduler has to take action to prevent timing failures. The measured maximum response time of 35,70ms is smaller than the task pair's deadline. So, during the whole run no timing failures occurred although the specified resource demand of the main part and the relative deadline were significantly shorter than the measured maximum execution time of the arc filter.

6.2.2.2 Exploiting Functional Redundancy

We conducted measurements to assess the impact of using anytime algorithms on the reliability of the arc filter. The arc filter is in charge of detecting arcs in the raw data delivered by a laser scanner (cf. Section 3.2). To this end, the arc filter successively evaluates tentative arc positions and, if stopped, delivers those positions with an evaluation above a given threshold. Since, at this point, we are particularly interested in situations in which the main part becomes faulty, we considered a scenario with large and increasing execution times (cf. the execution times plotted in Figure 6-6). Furthermore, we chose the scenario such that there actually was an arc in the field of view of the laser scanner so that the arc filter was expected to deliver the position of that arc. Thus, we were able to analyze the results of the arc filter in cases in which its execution time exceeded the allocated ECET. The ECET was set to 58.36ms (cf. Figure 6-6) corresponding to the 95%-quantile of the measurements presented in Section 5.1. To assess the impact of using an anytime algorithm for the arc filter, we compare for each set of input points the results delivered by a completed instance to the results delivered by an aborted instance. The most important part of the result is the estimated arc position with the highest evaluation because in a RoboCup-like context, where a single round object is to be detected, this would be considered the estimated position of the ball. Thus, our criterion in comparing the results was whether the estimated position with the highest evaluation in the completed instance was also present in the output of the aborted instance. As long as this was case, the output was considered to be sufficient.

Figure 6-6 presents the results. It shows that in the considered situation, there are a large number of instances that exceed their ECET; that is, a large number of faulty main parts. The detection time plotted in the figure is the time at which the arc filter found the tentative arc position with the highest evaluation. All the detection times are smaller than the ECET. This makes clear that each instance of the arc filter that delivered a position estimate at all, detected the best position estimate before the ECET. This means that all instances, even if aborted at their ECET, would have had the best position estimate in their output and therefore would have delivered sufficient results. Thus, in the measurement, the fact that task instances had to be aborted did not affect the reliability of the task pair. An example situation from the application prototype that shows how spatial redundancy can be exploited has been presented in Sub-Section 5.3.2.

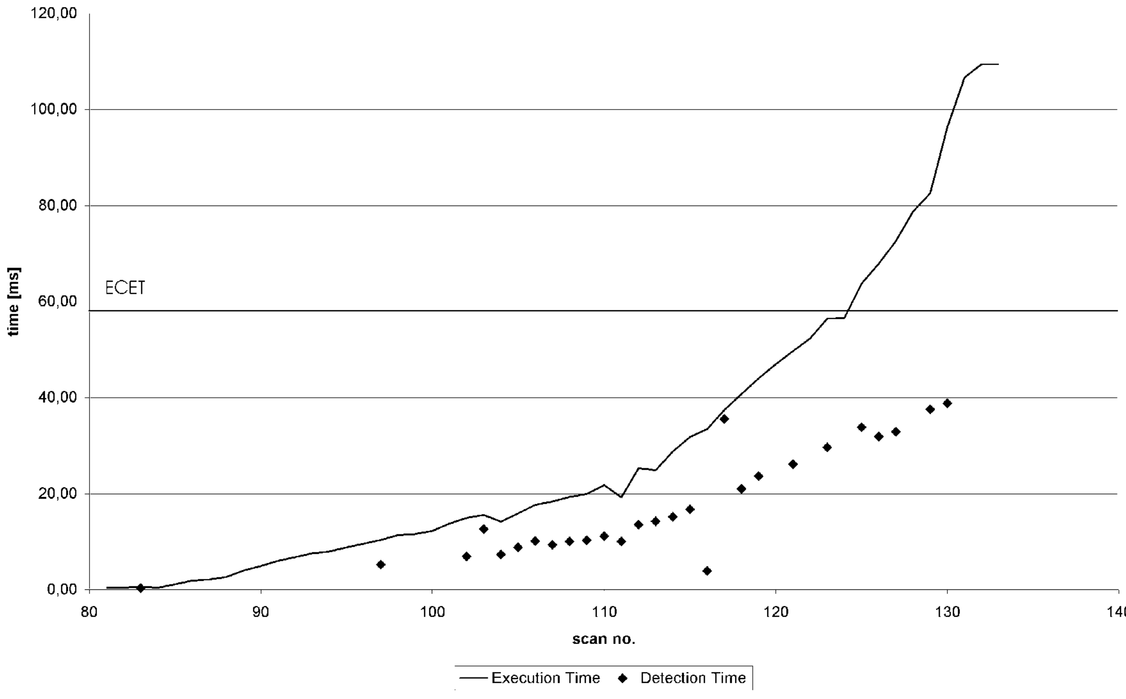


Figure 6-6. Execution times and detection times of the arc filter

6.2.2.3 Application-Level Adaptation

In this clause, we present measurements illustrating the idea of application-level adaptation in the distributed sensor fusion scenario. On the one hand, we show that performing the fusion on a higher level of abstraction reduces the load on the CPU. On the other hand, we present an example situation where fusion on a higher level of abstraction produced less complete results than fusion on a lower level.

Figure 6-7 shows the measured execution times of the modules for sensor fusion on different levels, each column corresponding to one of the levels. The measurements have been conducted on an AMD Athlon 700, 128MB RAM, with three rectangular objects and a ball situated within the field (Figure 6-8). Filtering stages before the fusion are executed for both scans so that two times appear in the table.

	Point level	Contour level	Object level	Element level
Point fusion	181.5	-	-	-
Contour filter	714.3	180.0 / 180.1	177.3 / 180.7	180.0 / 180.1
Contour fusion	-	1.1	-	-
Object filter	0.9	0.6	0.2 / 0.2	0.3 / 0.2
Object fusion	-	-	2.8	-
Element filter	0.1	0.1	0.1	0.06 / 0.04
Element fusion	-	-	-	0.02
Sum	896.8	181.9	183.8	180.38

Figure 6-7. Comparison of execution times [ms] for different fusion levels

The results show that performing the sensor fusion on the point level has by far the longest delay. Most of the time is spent for the point level fusion and for filtering the fused data, because both modules have two complete sets of points (or the union thereof) as input. If fusion is performed on higher levels, filtering the raw data consumes the largest part of the overall time as well. Therefore, even if fusing data on the object or element level saves some time in the corresponding modules, this effect is not significant compared to the execution time of the contour filter. Hence, the contour-, object-, and element-level fusion all have about the same overall execution time.

Figure 6-8 presents an example scenario from the prototype where three rectangular objects and a ball are located in the field. The pictures (a) and (b) show the raw data of two laser scanners observing the scene. Picture (c) shows the results obtained by fusing the data of the two scanners on the element level. The boxes represent the bounding boxes computed by the element filter. The fused set of elements contains only two of the three robots. This is because none of the scanners has sufficient data on its own to recognize the missing robot. This shows that the local view of a single sensor might not suffice to recognize all objects in the environment and that fusion on the element level does not always solve this problem. Element level fusion yields complete results only if for each element at least one scanner has sufficient data to classify it. Figure 6-8 (d) depicts the results (on the element level) for contour level fusion. Here, all elements have been detected. Both scanners together perceive a sufficient fraction of the robot missing in Figure 6-8 (c) to classify it.

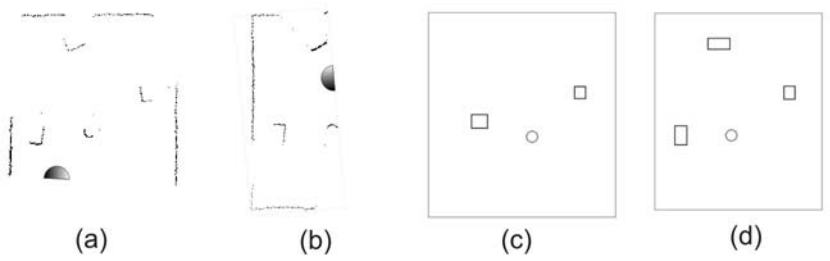


Figure 6-8. Raw data (a,b) and results on the element level for fusion on the element (c) and the contour level (d).

Taking both results presented above together, we see that switching to a higher level of abstraction under persistent overload can reduce the system load significantly and thus is a promising approach to prevent task abortions. On the other hand, the price to be paid is a possibly reduced accuracy or completeness of the results; that is, a degradation of the delivered service.

6.3 Modular Implementation of the Communication Hardcore

An implementation of the protocol stack was conducted under RTLinux (Vandersee 2004). The goal of this implementation was twofold: First, the implementation has to observe small and predictable delays to be applicable in real-time applications with tight time constraints; second, we want the implementation to maintain the modular structure of the formal specification so that application-specific protocol stacks can be configured at compile time (the formal specification is presented in the Appendix). The particular problem in implementing the protocols lies in the relation of the goals. In modular implementations the communication between modules soon becomes a performance bottleneck. If it is not handled efficiently, the implementation will exhibit a poor performance. Thus, a crucial aspect of this implementation is to find concepts for an efficient communication among the micro protocols within the protocol stack as well as between the protocol stack and both the drive below it and the application above it.

6.3.1 Description of the Implementation

The implementation was conducted under RTlinux versions 3.1 and 3.2pre. It is based on Orinoco 802.11b Standard wireless cards for which an RTLinux driver had to be written. The driver was implemented as a separate kernel module. The protocol stack itself was implemented as a kernel module too. Most of the source code was written in C++; only some particularly time-critical parts were programmed in C.

6.3.1.1 Object Structure of the Implementation

Each process of the formal specification has been implement as a separate class. All these classes are derived from a common base class *SDLProcess* (cf. Figure 6-9). This base class has two virtual methods *Init* and *DispatchSignal*, which each derived process class must implement. Furthermore, it has two attributes *State* and *PId*, the former being the explicit state of the process' state machine and the later its process ID. Each derived process class adds its own attributes corresponding the data in the SDL process it implements. The *Init* method mainly implements the initial transition of the process's state machine and is called at startup of the protocol stack. The *DispatchSignal* method is the core of the class. It implants the transitions of the state machine. Whenever a process consumes a signal in the formal model, this means the *DispatchSignal* method of the corresponding class is called in the implementation with the signal being provided as a parameter. Within this method, the transition to be performed is selected based on the actual parameters of the method and the *State* attribute of the object.

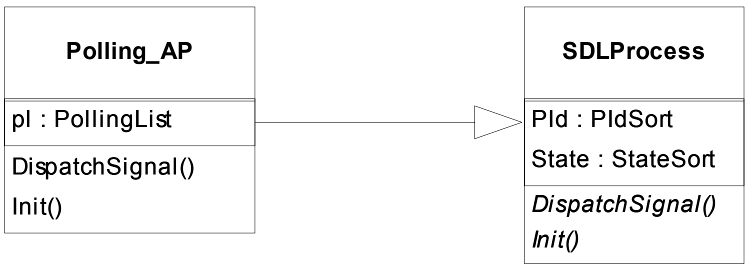


Figure 6-9. The implementation base class and an example of a derived class

Figure 6-9 shows the base class and the class *Polling_AP* as an example of a derived class. The class *Polling_AP* overwrites both abstract methods and adds the polling list, which is a data item in the formal specification, as an attribute.

The classes communicate via a global signal queue. When a class outputs a signal, it adds the signal to the global queue. The global signal queue is an active object. It reads signals from the queue, determines the receiving process, and calls the dispatch method of the corresponding object. Using a centrally maintained signal queue rather than a separate queue for each process is more efficient because the central signal scheduler calls the dispatch method only if there is actually a signal to process for the object. Figure 6-10 illustrates the signal exchanging between the global queue and the process objects. It shows an object of the class *QScheduler* that is connected to several process objects. In the figure, object *Polling_dyn_AP* outputs a signal to the global queue. The scheduler dispatches this signal to the object *DynMedAcc_AP* by calling the method *DispatchSignal* of *DynMedAcc_AP*. During the execution of the transition the object outputs another signal by adding it to the global signal queue.

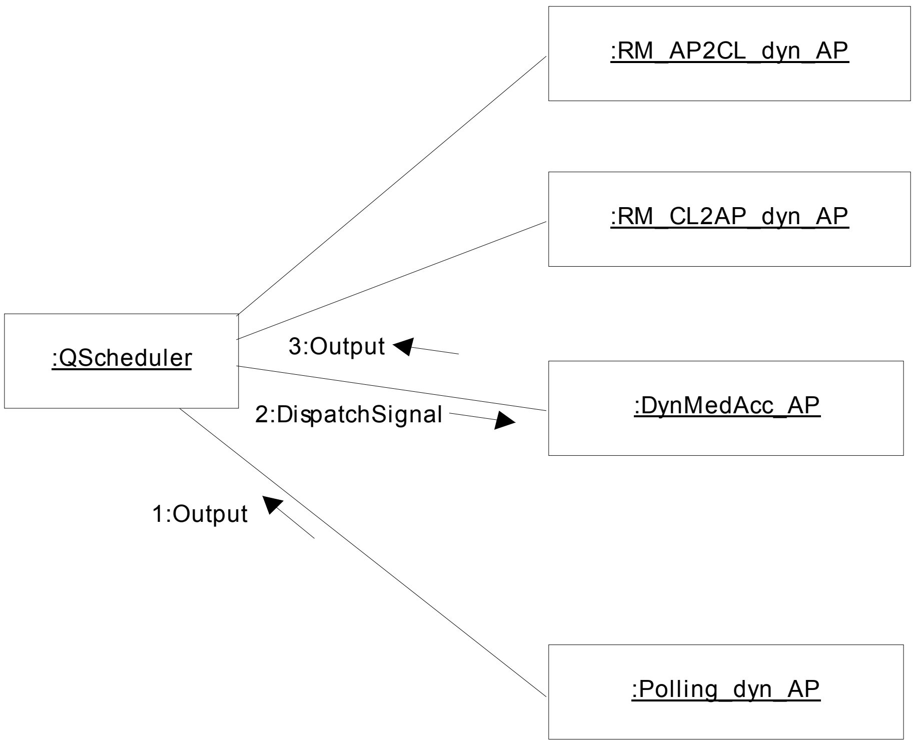


Figure 6-10. Signal exchange between the process objects and the central signal queue

6.3.1.2 Achieving Efficiency

As explained above, it is crucial to keep the time overhead of the communication between the processes small. To accomplish this, two problems have to be tackled:

1. Avoid copying of the application messages, which usually constitute the largest part of the PDUs the protocols send. The protocols themselves typically add some header information only.

2. Avoid scheduling delays between executing the sender and the receiver of a signal.

To address the first point, a shared memory module was developed. When the application wants to transmit a message, it requests a buffer from the shared memory module. All buffers have a fixed size, which is sufficient for a maximum application message plus all headers possibly added by the micro protocols. The application copies its message at the end of the buffer and passes a handle to that buffer to the protocol stack. Within the protocol stack the content of the buffer is never copied. Rather, each micro protocol adds its header in front of the headers added by the higher layers. This means that only the pointer to the start of the frame is moved. Thus, the frame is successively extended from the end of the buffer towards its beginning. When the buffer has been passed through the whole stack, all headers have been added and the complete frame has been constructed within the buffer. The protocol stack then passes the handle to the driver, which copies the frame into the network interface card. In this approach, the user data are never copied; only the handle is passed from protocol to protocol. Note that this holds also for the communication between the application and the protocol stack as well as between the protocol stack and the driver.

Communication in the other direction works very much the same way. In this case the driver requests the buffer. It reads the frame from the network interface card into the buffer and passes the handle to the protocol stack. In the protocol stack, the micro protocols successively remove their headers; that is, they move the pointer indicating the start of the frame towards the end of the buffer. Thus, the frame is successively shrunk until only the application message remains in the buffer. At the end, the protocol stack passes the handle to the application, which can then read the message.

To avoid scheduling delays between executing successive micro protocols, a single thread executes all the protocols. This thread waits for messages to arrive from either the driver or the application. Whenever this happens, it puts a corresponding signal into the signal queue and then starts processing the signal as described above. When the queue is empty, the thread again waits for the next message to arrive.

The thread waits for message arrivals passively. A single semaphore is used to signal message arrivals from both the application and the driver. If one of both passes a buffer to the protocol stack, it increments the semaphore. The thread in the protocol stack, in its turn, decrements the semaphore whenever it receives a buffer and hence blocks when it has processed all buffers passed by the application or the driver.

6.3.1.3 Configuration and API

Configuring stacks is done at compile time. To change the configuration of the protocol stack, a single header file has to be edited. The user decides which protocols are part of the stack in which order by defining their process IDs in this header filer. Additionally, there is a single flag determining whether dynamic network scheduling is used or not. From this header file, a script generates all the configuration specific code to initialize the protocols and establish a matching between the protocol IDs and the corresponding objects. This script also modifies the makefiles controlling the compilation of the protocols. Figure 6-11 shows how the module size varies with the chosen configuration. Obviously, the fewer services are part of the configuration, the smaller is the memory footprint of the resulting module.

Configuration	Size of the AP module [KB]	Size of the client module [KB]
All services	62	50
Polling, Reliable Multicast, Atomic Multicast	52	41
Polling, Reliable Multicast	49	35
Polling	42	31

Figure 6-11. Sizes of the client and AP modules depending on configuration

Applications can access the services of the protocol stack through a simple API. To send a message, an application program first requests a buffer from the shared memory module (*shm_alloc*). It copies the message it wants to transmit into the buffer, putting parameters, such as the resiliency, in front of it. It then notifies the shared memory module of the buffer's being ready for transmission (*shm_done*). After this call, the shared memory module wakes-up the thread in the protocol stack so that it can process the message. The application program has two possibilities to receive messages. For a non-blocking receive, the program just checks if there has been any buffer delivered for it (*shm_getnextbuf*). As long as no message is waiting to be received, this call returns a null handle. The second method blocks the program while waiting for messages. To use this method, the program calls *sem_wait* on a semaphore the shared memory module provides. When the semaphore is open, the next call to *shm_getnextbuf* is ensured to deliver a handle to a buffer containing a message for the application.

6.3.2 Measurements

To evaluate the performance of the modular implementation we measured the delays of the communication services under varying configurations. The measurements were conducted on three machines, namely a Pentium P266 mobile, 128 MB RAM, acting as the AP, a Pentium III 450MHz, 256MB RAM, acting as the first client, and an AMD Duron 850, 512 MB RAM, acting as the second client. All three machines had RTLinux running as the operating system with the standard scheduler. The protocol parameters were set as follows: omission degree *OD* = 3, resiliency *res* = 0, and the poll timeout 30ms. We chose a resiliency of zero to reduce the impact of message losses on the delays. So, the results for different configurations can be better compared against one another. The measurements were carried out for groups with one and two clients; in both cases, one client sent 100Byte messages at a rate of 1/15ms.

	Polling		Reliable Multicast		Atomic Multicast	
Group Size	1	2	1	2	1	2
Min	2273	2268	2377	2392	4386	7078
Max	5916	8065	5779	8721	8570	14616
Avg.	3412	4441	3518	4644	5806	10074
Std. Dev.	659	1296	700	1337	734	1531

Figure 6-12. Delays of the communication services in µs

Figure 6-12 displays the results of the measurement. For one thing, they reveal the performance costs associated with choosing a stronger communication service. The lowest delays are obtained if only polling is used. It is remarkable that adding reliable multicast only slightly increases the delays. This shows that the reliable multicast protocol only adds a very little fixed overhead. The difference between the delays of the two services will grow as soon as a resiliency greater than zero is chosen and an increasing number of message losses are considered. In this case, however, the larger delays are the price of higher reliability. The difference between the delays of the reliable and the atomic multicast service is more significant. This has two reasons: First, after the stations received a message, it takes at least one round until the APs decides whether to deliver it or not. Second, the AP has to transfer its decision to the stations, which takes at least another slot if no message losses occur. The difference between these two services, however, will not grow with the resiliency and only gradually with the number of message losses. Rather, it will reduce as compared to the overall delay. These measurements show that services with stronger semantics incur a performance cost so that it is better to include only those services in the hardcore that are actually needed.

To give an idea of how the presented results are to be rated, we compare them with the mean delays of an earlier, RTLinux-based monolithic implementation, which exhibits the best performance among the earlier implementations. The delays were measured in a similar setting: Intel Celeron 400 MHz, 128MB RAM, acting as AP, two AMD Athlon 700, 128MB RAM, acting as clients, and the following protocol parameters: omission degree *OD* = 15, resiliency *res* = 0, and the poll timeout 20ms. Note that the difference in the omission degree and the poll timeout are not very significant for the average values because the measurements took place under good link conditions. The results presented in Figure 6-13 show that the modular implementation achieves about the same performance as the monolithic one. Obviously, we can compare the delays for the atomic multicast service only, since in the monolithic implementation only this service can be accessed.

	Atomic Multicast	
Group Size	1	2
Monolithic Implementation	6716	11441
Modular Implementation	5806	10074

Figure 6-13. Mean delays of the modular and a monolithic implementation in µs.

7 Conclusion and Future Work

The cooperation of mobile embedded systems gives rise to a lot of new and fascinating applications. The ever increasing number of embedded computer devices pervading our every day live and the rapid deployment of wireless communication technology provide a technological context which strongly fosters the trend to develop and deploy cooperative applications for mobile embedded systems. Such applications already gained a significant attention in today's industrial research and development. In this thesis, a middleware was presented that supports the development of cooperative applications for mobile embedded systems beyond the services that today's COTS operating systems offer. In particular, the middleware contributes to the solution of the following two challenges:

- The real-time requirements stemming from the locomotion of the mobile systems and their interactions with the physical environment require achieving a timely predictable behavior. A timely predictable behavior, however, is hard to achieve because the execution times of the tasks depend on a dynamically changing environment, as does the number of message losses on the links.

- Achieving efficient cooperation requires managing the interactions between the cooperating systems. If this, however, results in complex runtime interactions within the application, a predictable timing behavior is hard to achieve. Therefore, cooperation should be supported in such a way that autonomy of the cooperating systems is maintained on the application level.

As a prerequisite for the design of the middleware we developed a formal system model that captures the characteristics of a dynamically changing group of mobile systems connected by a wireless LAN. The key aspect to be modeled is the varying quality of the communication links in terms of the number of message losses. As the basic underlying model we adopted a synchronous system model with an unknown number of omission failures of the network. Following an idea introduced by (Cristian 1996), we defined time-dependent predicates to describe the varying quality of the links between the mobile systems. Such a model allows the design of protocols that provide safety properties under a wide range of environmental conditions, in particular without resorting to worst-case assumptions about the number of message losses. Furthermore, the protocols can be designed such that they provide reliable and timely services whenever the link quality is sufficiently good.

The two bottom layers of the middleware deal with the first of the above-mentioned challenges. They provide a timely predictable execution of tasks with unpredictable execution times and a timely predictable transmission of reliable multicast messages. For the task execution service we adopt the TAFT approach. This thesis adds the following two contributions to this approach:

1. To use TAFT in the task execution service of the middleware, the task model underlying the current implementation of TAFT had to be extended. First, the communication protocols, which are part of the middleware, require CPU resources and hence have to be scheduled in addition to the application tasks. The corresponding tasks are not adequately modeled as periodic tasks so that the model was extended to include aperiodic requests also. Furthermore, as turned out in our application scenario, application tasks in our intended application context typically exhibit precedence constraints, especially if they process sensor data. So, precedence constraints between periodic task pairs were included in the task model too. We presented a scheduling algorithm, called TAFT-IPE, which schedules aperiodic tasks and periodic task pairs in the extended model. It schedules each set of periodic task pairs with a processor utilization not greater than one. For aperiodic requests a guarantee is given on a per instance basis; that is, an aperiodic request is either finished in time or not executed at all. This fits to the underlying system model, where the execution service exhibits omission failure semantics. A set of simple sufficient and necessary conditions has been developed to allow for a fast acceptance test for the aperiodic requests.

2. The TAFT concept proposes to use redundancy to tolerate task abortions caused by resource faults. In the distributed sensor fusion scenario, we examined how application-inherent redundancy can be provided and how it can be exploited to tolerate task abortions. Experiments in the prototype show that exploiting application-inherent redundancy allows achieving a reliable task execution even if worst-case bounds on task execution times are not provided.

Regarding communication, the IEEE 802.11 Standard already provides basic support for predictable timing behavior through the polling mechanism. To add support for dynamic groups we suggested a solution to provide predictable access to joining stations, which are not yet part of the polling list. Since there is no general solution to that problem, we exploit application semantics here. We believe that the solution is applicable in many applications nevertheless. In particular, all those applications in the area of traffic control and logistics that motivated our first application scenario can adopt this solution. The key problem in achieving predictability for the communication services stems from unpredictable loss rates of the wireless medium. We presented a reliable multicast protocol, which adopts a dynamic timing redundancy approach to tolerate message losses and comprises an efficient acknowledgement mechanism. Instead of working with worst-case bounds, the protocol allows the user to specify for each message a resiliency, which is a bound on the number of retransmissions of that message. The protocol guarantees that the transmission delays of the messages are bounded, with the delay bound depending on the resiliency of the message. Thus, the protocol allows trading reliability for smaller time-bounds. Without using worst-case bounds, it may happen that not all stations are able to deliver a multicast message. In this case, the atomic multicast protocol on the next layer achieves agreement among the valid members.

In this thesis, we proposed an approach to achieve a coordinated behavior while preserving the autonomy of the mobile systems. Thus, complex runtime interactions on the application level are avoided. According to this approach, the two upper layers provide common views to the application. Based on these common views, the mobile systems make decisions locally at runtime. Thus, rather than explicitly coordinating the actions of the mobile systems on the application level, the middleware coordinates their worldviews. Since both the communication services of the middleware as well as the execution of the local application tasks are timely predictable, this approach achieves a timely predictable cooperation.

The two upper layers of the middleware implement this approach. The lower of these two layers provides common views on application-independent aspects of the control system. Namely, it provides a common global time base, group membership, and atomic multicasts. The protocols implementing membership and atomic multicasts have been presented as part of this thesis. The atomic multicast protocol ensures that all stations, valid or not, perceive infixes of the same totally ordered sequence of messages. Valid stations agree on the messages they deliver in bounded time. In particular, this is ensured if the underlying reliable multicast protocol does not deliver a message at all valid stations. In conjunction with the atomic multicast protocol, the membership protocol ensures that all stations deliver infixes of the same totally ordered sequence of membership changes and multicast messages. Every change in the group membership is reflected in a membership change message and delivered to all valid stations in bounded time. Both the atomic multicast and the membership protocol provide fail-awareness. This means that these protocols indicate to their users whether they are currently providing a valid service or not. When the protocol entity at some station is no longer able to guarantee agreement with the valid stations or an up-to-date view on the membership, it indicates this fact to its user. Thus, the user is enabled to react to this situation in bounded time.

The Event Service, on the highest layer, provides common views on the global state of the controlled system. The global state comprises the local states of the mobile system w.r.t to the same point of time on the global time base. The Event Service associates global states to the events, which are delivered totally ordered and with bounded delay based on the services of the underlying group communication protocols. Determination of the global states is based on a model of the local states, which describes how the local states evolve between the deliveries of two events. Using such a model, we obtained the following advantages:

- Time coherence of the global states is achieved without synchronizing the observation of the local states.
- The temporal consistency of the local states is improved, since the error caused by communication delays can be compensated.
- The message overhead required to determine the global states is significantly reduced.

The modular design of the middleware allows adapting it to the requirements of a variety of different applications. In particular, the modular design of the communication protocols in the hardcore was a challenging task. These protocols have been designed as a family of micro protocols, each with a well-defined service and interface. This involved resolving or balancing the tradeoff between modularity and efficiency, both during the design and the implementation of the protocols. The implementation allows configuring protocol stacks at runtime. Measurement show that the modular implementation achieves a high performance and it allowed assessing the performance costs associated with choosing stronger communication semantics.

Two prototypes were built up. The first shows the feasibility of our approach to the timely predictable coordination of mobile systems. In this prototype, a group of robots coordinate their speeds while approaching a shared spatial resource. According to our approach, the Event Service is used to determine the global state of the group whenever a rescheduling of the shared resource is required. At the application level, a locally executed function, the so-called scheduling function, determines the schedule for the shared resource based on the global state. Since both the determination of the global state in the middleware, as well as the computation of the scheduling function is timely predictable, a timely predictable behavior can be achieved. The second prototype illustrates and validates our approach to enforcing QoS in dynamically changing environments. It is a concrete example of a distributed fusion of sensor data within a group of mobile robots. Our analysis of the execution times revealed that the execution times of the sensor processing tasks not only depend on the amount of input but also on its content. So, the execution times of these tasks are in fact environment dependent and widely varying. TAFT has been applied to achieve a timely predictable behavior for these tasks, and our measurements show that no task instance missed its deadline. Furthermore, the prototype served to show that application-inherent redundancy allows tolerating task abortions and increasing the reliability of task execution.

The work presented in this thesis will be carried on along different lines of development. One line is to consider how other kinds of message traffic can be integrated with the presented communication protocols on the same wireless LAN. Other traffic classes may differ in the kinds of protocols they use, for example, point-to-point instead of multicast protocols, in their resource demands, and in the kinds of deadlines they have. Integrating different kinds of traffic on a single LAN will particularly affect the MAC and the dynamic network-scheduling layer. Here, it will be interesting to consider how the new and improved features of the supplement 802.11e can be best exploited and integrated into the middleware. We believe that the priority-based access during the DCF (EDCA in 802.11e), the more flexible polling mechanism (HCCA in 802.11e), and the extensions that support demand specification and negotiation, will afford interesting opportunities to accommodate different traffic classes in an integrated and flexible manner.

Another line of development is considering scalability in terms of both the spatial extent of the network and the number of stations it comprises. This thesis focused on the cooperation in local groups, but there are interesting cooperative applications for larger scale wireless networks as well. For example, consider an extended sensor network of small sensing devices that interacts with mobile systems like robots, or humans carrying portable computing equipment. One may imagine that this kind of structure will be used to help fire fighters in fulfilling their tasks. Works considering such kinds of larger networks are ongoing in our working group in the context of a project supported by the DFG[7] (Trikaliotis 2004). We consider large-scale networks to consist of clusters, each of which corresponds to a local group. While cooperation under tight real-time constraints is performed within the clusters, the clusters are connected to a large-scale network more loosely. For one thing, this requires adding new inter-cluster communication services to the intra-cluster services presented herein. In particular, routing messages through a network of clusters becomes an important issue. Furthermore, new services have to be added on the highest layer of the architecture, the CADI, which still follow the common view paradigm but are more tailored to the kind of cooperation and interaction found in large-scale networks. In particular, anonymous communication with content-based addressing fosters scalability and allows the mobile systems to access a distributed service infrastructure, like the sensor network in the example above, transparently without explicitly searching and contacting the nodes providing the services. Still, common views on the events this transparent service infrastructure delivers allow achieving a coordinated behavior of the mobile systems without explicit coordination on the application level.

[7] Deutsche Forschungsgemeinschaft (German Research Foundation)

References

Abdelzaher, T. F., A. Shaikh, et al. (1996). RTCAST: Lightweight Multicast for Real-Time Process Groups. In *Proc. 2nd IEEE Real-Time Technology and Applications Symposium*, Boston, Mass.

Almeida, C. and P. Verissimo (1995). An Adaptive Real-Time Group Communication Protocol. In *Proc. 1st IEEE Workshop on Factory Communication Systems*, Lausanne, Switzerland.

Almeida, C. and P. Verissimo (1996). Timing Failure Detection and Real-Time Group Communication in Quasi-Synchronous Systems. In *Proc. 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy.

Amir, Y., D. Dolev, et al. (1992). Transis. A communication subsystem for high availability. In *Proc. 22nd Int'l Symp. on Fault-Tolerant Compting*, Boston, Mass.

Amir, Y., L. E. Moser, et al. (1995). The Totem Single-Ring Ordering and Membership Protocol. *ACM Transactions on Computer Systems* **13**(4): 311-342.

Bade, R. (2003). Modifikation einer Stereobildverarbeitungsmethode für die Anwendung in einer Echtzeitumgebung. Diploma Thesis, Institute for Distributed Systems, Otto-von-Guericke-University**:** Magdeburg, Germany.

Bakre, A. and B. R. Badrinath. M-RPC: A Remote Procedure Call Service for Mobile Clients. Technical Report WINLAB TR-98, Department of Computer Science, Rutger University, 1995.

Baldwin, R. O., N. J. I. Davis, et al. (1999). A Real-Time Medium Access Control Protocol for Ad Hoc Wireless Local Area Networks. *Mobile Computing and Communications Review* **3**(2): 20-27.

Bar-Joseph, Z., I. Keidar, et al. (2000). QoS Preserving Totally Ordered Multicast. In *Proc. 5th Int'l Conf. on Principles of Distributed Systems*.

Bar-Shalom, Y. and T. E. Fortmann (1988). *Tracking and Data Association*. Boston, Academic Press.

Becker, L. B. and M. Gergeleit (2001). Execution Environment for Dynamically Scheduling Real-Time Tasks. In *Proc. 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, London.

Becker, L. B., M. Gergeleit, et al. (2001). An Approach for Implementing Object-Orientet Real-Time Models on Top of Embedded Targets. In *Proc. OMER-2 - Workshop on Object-oriented Modelling of Embedded Real-time Systems*, Herrsching am Ammersee, Germany.

Becker, L. B., M. Gergeleit, et al. (2003). Using a Flexible Real-Time Scheduling Strategy in a Distributed Embedded Applications. In *Proc. 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Lisbon, Portugal.

Becker, L. B., E. Nett, et al. (to appear). Robust Scheduling in Team-Robotics. *Journal of Systems and Software*.

Bernat, G. and A. Burns (2001). Weakly-Hard Real-Time Systems. *IEEE Transactions on Computers* **50**(4): 308-321.

Bernat, G. and R. Cayssials (2001). Guaranteed On-Line Weakly-Hard Real-Time Systems. In *Proc. Proc. of 22nd IEEE Real-Time Systems Symposium*, London.

Birman, K. P., A. Schiper, et al. (1991). Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems* **9**(3): 272-314.

Birman, K. P. and R. van Renesse (1994). *Reliable Distributed Computing with the ISIS Toolkit*. Los Alamitos, Calif., IEEE Computer Society Press.

Boddy, M. and T. Dean (1989). Solving Time-Dependent Planning Problems. In *Proc. IJCAI*.

Braek, R. and O. Haugen (1993). *Engineering Real Time Systems - An object-oriented methodology using SDL*. New York, London, Prentice Hall.

Casimiro, A. and P. Verissimo (2001). Using the Timely Computing Base for Dependable QoS Adaption. In *Proc. Symposium on Reliable Distributed Systems*, New Orleans, USA.

Cavalieri, S. and D. Panno (1997). On the Integration of Fieldbus Traffic within IEEE 802.11 Wireless LAN. In *Proc. IEEE International Workshop on Factory Communication Systems*, Barcelona, Spain.

Chandra, T. D., V. Hadzilacos, et al. (1992). The Weakest Failure Detector for Solving Consensus. In *Proc. ACM Symp. on Principles of Distributed Computing*, Vancouver, Canada.

Chandra, T. D. and S. Toueg (1991). Unreliable Failure Detectors for Asynchronous Systems. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, Montreal, Canada.

Chang, J.-M. and N. F. Maxemchuck (1984). Reliable Broadcast Protocols. *ACM Transactions on Computer Systems* **2**(3): 251-273.

Chetto, H. and M. Chetto (1989). Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transactions on Software Engineering* **15**(10): 1261-1269.

Chockler, G. V., N. Huleihel, et al. (1998). An Adaptive Total Ordering Multicast Protocol That Tolerates Partitions. In *Proc. 17th ACM Symp. on Principles of Distributed Computing*, Puerto Vallarta, Mexica.

Chockler, G. V., I. Keidar, et al. (2001). Group Communication Sepcifications: A Comprehensive Study. *ACM Computing Surveys* **33**(4): 1-43.

Cirrus Logic. Whitecap 2 Wireless Network Protocol. Whitepaper, Cirrus Logic.

Coutras, C., S. Gupta, et al. (2000). Scheduling of Real-Time Traffic in 802.11 Wireless LANs. *Wireless Networks* **6**: 457-466.

Cristian, F. (1989). Probabilistic Clock Synchronization. *Distributed Computing* **3**: 146-156.

Cristian, F. (1991). Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems. *Distributed Computing* **4**: 175-187.

Cristian, F. (1996). Synchronous and Asynchronous Group Communication. *Communications of the ACM* **39**(4): 88-97.

Cristian, F., H. Aghili, et al. (1985). Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *Proc. 15th Int'l Symp. on Fault-Tolerant Computing*, Ann Arbor, Mich.

Cristian, F. and C. Fetzer (1999). The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems* **10**(6): 642ff.

Cristian, F. and S. Mishra (1995). The Pinwheel Asynchronous Atomic Broadcast Protocols. In *Proc. 2nd Int'l Symp. on Autonomous Decentralized Systems*, Phoenix, Arizona.

Cristian, F. and F. Schmuck. Agreeing on Processor Group Membership in Timed Asynchronous Distributed Systems. Technical Report CSE95-428, UCSD: San Diego, 1995.

Dean, T. and M. Boddy (1988). An Analysis of Time-dependent Planning. In *Proc. AAAI-88*.

Defago, X., A. Schiper, et al. Totally Ordered Broadcast and Multicast Algorithms: A Comprehensive Survey. Technichal Report DSC/2000/036, Swiss Federal Institute of Technology: Lausanne, Switzerland, 2000.

Dietl, M., J.-S. Gutmann, et al. (2001). Cooperative Sensing in Dynamic Environemts. In *Proc. IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems (IROS'01)*, Maui, Hawaii.

Dolev, D. and C. Dwork (1987). On the Minimal Synchronism for Distributed Consensus. *Journal of the ACM* **34**(1): 77-97.

Ergen, M., D. Lee, et al. (2002). Wireless Token Ring Protocol. In *Proc. 6th World Multi-conference on Systemics, Cybernatics and Informatics (SCI)*, Orlando, Fla.

Ezhilchelvan, P. D. and R. de Lemos (1990). A Robust Group Membership Algorithm for Distributed Real-Time Systems. In *Proc. Real-Time Systems Symposium*, Lake Buena Vista, Fla.

Ezhilchelvan, P. D., R. A. Macêdo, et al. (1995). Newtop: A Fault-Tolerant Group Communication Protocol. In *Proc. 15. Int. Conf. on Distributed Computing Systems*, Vancouver, Canada.

Feist, M. (2002). Mehrstufige, verteilte Sensorfusion von Laserscandaten. Diploma Thesis, Institute for Distributed Systems, Otto-von-Guericke-University**:** Magdeburg, Germany.

Fetzer, C. and F. Cristian (1996). Fail-Awareness in Time Asynchronous Systems. In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, Philadelphia.

Fischer, M. J., N. Lynch, et al. (1985). Impossibility of Distributed Consensus with on Faulty Process. *Journal of the ACM* **32**(2): 374-382.

Franz, W. J., H. Hartenstein, et al. (2001). Internet on the Road via Inter-Vehicle Communications. In *Proc. Workshop on Mobile Communication over Wireless LAN: Research and Applications*, Vienna, Austria.

Galleni, A. and D. Powell. Consensus and Membership in Synchronous and Asynchronous Distributed Systems. Technical Report LAAS No96104, LAAS-CNRS, 1996.

Gergeleit, M. (2001). A Monitoring-based Approach to Object-Oriented Real-Time Computing. PhD Thesis, Institute for Distributed Systems, Otto-von-Guericke-Universität Magdeburg**:** Magdeburg, Germany.

Gergeleit, M., L. Buss Becker, et al. (2003). Robust Scheduling in Team-Robotics. In *Proc. WPDRTS'03*, Nice, France.

Grünsteidl, G. and H. Kopetz (1991). A Reliable Multicast Protocol for Distributed Real-Time Systems. In *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, Georgia, USA.

Haardt, M., A. Klein, et al. (2000). The TD-CDMA Based UTRA TDD Mode. *IEEE Journal on Selected Areas in Communications* **18**(8): 1375-1385.

Hamdaoui, M. and P. Ramanathan (1995). A Dynamic Priority Assignment Technique for Streams with (m,k)-firm Deadlines. *IEEE Transactions on Computers*.

Harrison, T. H., D. L. Levine, et al. (1997). The Design and Performance of a Real-time CORBA Event Service. In *Proc. OOPSLA '97*, Atlanta, GA.

Herms, A. (2004). Entwurf eines verteilten Laufplaners basierend auf heuristischen Optimierungsverfahren. Diploma Thesis, Institute for Distributed Systems, Otto-von-Guericke-University**:** Magdeburg, Germany.

Hiltunen, M. A. and R. D. Schlichting (1995a). Properties of Membership. In *Proc. 2nd IEEE Symposium on Autonomous Decentralized Systems*.

Hiltunen, M. A. and R. D. Schlichting. Understanding Membership. Technical Report 95-07, Department of Computer Science, University of Arizona: Tuscon, Ariz., 1995b.

IEEE. IEEE Std. 802.11b, Higher-Speed Physical Layer Extension in the 2.4 GHz Band. Standard, IEEE: New York, 1999.

IEEE. IEEE 802.11F - IEEE Trial-Use Recommended Practice for Multi-Vendor Access Point Interoperability via an Inter-Access Point Protocol Across Distribution Systems Supporting IEEE 802.11™ Operation. Standard, IEEE: New York, 2003.

Inoue, Y., M. Iizuka, et al. (1998). A Reliable Multicast Protocol for Wireless Systems with Representative Acknowledgement Scheme. In *Proc. 5th International Workshop on Mobile Multimedia Communication*, Berlin, Germany.

ITU-T. CCITT Specification and Description Language (SDL). ITU-T Recommendation Z.100 (03/93), ITU-T, 1993.

ITU-T. Specification and Description Language (SDL). ITU-T Recommendation Z.100 (11/99), ITU-T, 1999.

Jia, W., J. Kaiser, et al. (1996). RMP: Fault-Tolerant Group Communication. *IEEE Micro* **16**(2): 59-67.

Kaashoek, M. F. and A. S. Tanenbaum (1991). Group Communication in the Amoeba Distributed Operating System. In *Proc. 11th International Conference on Distributed Computing Systems*, Arlington, Texas, USA.

Kermarrec, A.-M., L. Massoulié, et al. (2003). Probabilistic Reliable Dissemination in Large-Scale Systems. *IEEE Transactions on Parallel and Distributed Systems* **14**(3): 248-258.

Killijian, M., R. Cunningham, et al. (2001). Towards Group Communication for Mobile Participants. In *Proc. Principles of Mobile Computing*, Newport, Rhode Island.

Kitano, H., M. Asada, et al. (1997). RoboCup: The Robot World Cup. In *Proc. Agents 1997.*

Knieriemen, T. (1991). *Autonome mobile Roboter – Sensordateninterpretation und Weltmodellierung zur Navigation in unbekannter Umgebung*, B. I. Wissenschaftsverlag.

Kopetz, H. (1997). *Real-Time Systems*. Boston, Kluwer Academic Press.

Kopetz, H. and G. Grünsteidl (1993). TTP - A Time Triggered Protocol for Fault-Tolerant Real-Time Systems. In *Proc. 23rd International Symposium on Fault-Tolerant Computing*, Toulouse, France.

Koren, G. and D. Shasha (1995). Skip-over: Algorithms and Complexity for Overloaded Real-Times Systems. In *Proc. IEEE Real-Time Systems Symp.*

Kümmel, S., A. Schill, et al. (1996). RPC over Advance Network Technologies: Evaluation and Experiences. In *Proc. 3rd Int'l Workshop on Services in Distributed Networked Environments*, Macua, China.

Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* **21**(7): 558-565.

Laprie, J. C., Ed. (1992). *Dependability: Basic Concepts and Terminology*, Springer.

Lin, K.-J., S. Natarajan, et al. (1987). Imprecise Results: Utilizing Partial Computations in Real-Time Systems. In *Proc. IEEE 8th Real-Time Systems Symposium*, San Jose.

Liu, J. W.-S., W.-K. Shih, et al. (1994). Imprecise Computations. *Proceedings of the IEEE* **82**(1): 83-94.

Malone, T. W. and K. Crowston (1994). The Interdisciplinary Study of Coordination. *ACM Computing Surveys* **26**(1): 87-119.

Masum, A. (2000). Non-Cooperative Byzantine Failures - A New Framework for the Design of Efficient Fault Tolerant Protocols. Dissertation, Fachbereich Mathematik und Informatik, Universität-GH Essen**:** Essen.

Melliar-Smith, P. M., L. E. Moser, et al. (1990). Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems* **1**(1): 17-25.

Microsoft Corporation. DCOM Technical Overview. Mirosoft Windows NT Server White Paper, 1996.

Mishra, S., C. Fetzer, et al. (1997). The Timewheel Asynchronous Atomic Broadcast Protocol. In *Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nev.

Mishra, S., C. Fetzer, et al. (1998). The Timewheel Group Communication Protocol. In *Proc. Workshop on Fault Tolerant Parallel and Distributed Systems*, Orlando, Flor.

Mishra, S., C. Fetzer, et al. (2002). The Timewheel Group Communication System. *IEEE Transactions on Computers* **51**(8).

Mitschele-Thiel, A. (2001). *Systems Engineering with SDL - Developing Performance Critical Communication Systems*. Chichester, New York, John Wiley & Sons, Ltd.

Mock, M. (2003). On the Coordination of Autonomous Systems. habilitation thesis, Faculty of Computer Science, Otto-von-Guericke-University**:** Magdeburg.

Mock, M., R. Frings, et al. (2000a). Clock Synchronization for Wireless Local Area Networks. In *Proc. 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden.

Mock, M., R. Frings, et al. (2000b). Continuous Clock Synchronization in Wireless Real-Time Applications. In *Proc. 19th IEEE Symposium on Reliable Distributed Systems*, Nuremberg, Germany.

Mock, M., E. Nett, et al. (1999). Efficient Reliable Real-Time Group Communication for Wireless Local Area Networks. In *Proc. 3rd European Dependable Computing Conference*, Prague, Czech Republic, Springer.

Moser, L. E., P. M. Melliar-Smith, et al. (1994). Processor Mebership in Asynchronous Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems* **5**(5): 459-473.

Nett, E. (1991). Supporting Fault Tolerant Distributed Computations. Habilitationsschrift, Universität Bonn.

Nett, E. and M. Gergeleit (1997). Preserving Real-Time Behavior in Dynamic Distributed Systems. In *Proc. IASTED International Conference on Intelligent Information Systems*, The Bahamas.

Nett, E., M. Gergeleit, et al. (1997). Flexible Resource Scheduling and Control in an Adaptive Real-Time Environment. In *Proc. IASTED International Conference on Artificial Intelligence and Soft Computing*, Banff, Canada.

Nett, E., M. Mock, et al. (2001). *Das drahtlose Ethernet - Der IEEE 802.11-Standard: Grundlagen und Anwendung*, Addison-Wesley.

Nett, E. and S. Schemmer (2003a). Realizing Virtual Sensors by Distributed Multi-Level Sensor Fusion. In *Proc. 2003 International Workshop on Multi-Robot Systems*, Washington, D.C, Kluwever Academic Publishers.

Nett, E. and S. Schemmer (2003b). Reliable Real-Time Communication in Cooperative Mobile Applications. *IEEE Transactions on Computers* **52**(2): 166-180.

Nett, E. and S. Schemmer (2004). An Architecture to Support Cooperating Mobile Embedded Systems. In *Proc. 2004 ACM Computing Frontiers Conference*, Ischia, Italy.

Object Management Group. The Common Object Request Broker: Architecture and Specification, 2.6.1. Object Management Group, 2002.

Oki, B., M. Pfluegel, et al. (1993). The Information Bus - An Architecture for Extensible Distributed Systems. In *Proc. ACM Symposium on Operating System Principles*.

Peterson, L. L., N. C. Buchholz, et al. (1989). Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems* **7**(3): 217-246.

Piaggio, M. and A. Sgorbissa (2000). Exploiting ethnos for communication and coordination of heterogenous soccer robots in the art team. In *Proc. European Workshop on RoboCup*, Amsterdam, Netherlands.

Piaggio, M., A. Sgorbissa, et al. (1999). Programing Real Time Distrubuted Multiple Robotic Systems. In *Proc. RoboCup-99 Workshop*, Stockholm, Sweden.

Plenge, C. (1995). The Performance of Medium Access Protocols for Inter-Vehicle Communication Systems. In *Proc. Mobile Kommunikation*, Neu-Ulm, Germany.

Powell, D. (1992). Failure Mode Assumptions and Assumptions Coverage. In *Proc. 22nd Int'l Symp. on Fault-Tolerant Computing*, Boston, Mass., IEEE.

Rajkumar, R., M. Gagliardi, et al. (1995). The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. In *Proc. First IEEE Real-Time Technology and Applications Symposium*.

Rappaport, T. S. (1996). *Wireless Communications - Principles & Practice*. Upper Saddle River, New Jersey, Prentice Hall PTR.

Rodrigues, L., H. Fonseca, et al. (1995). Reliable Computing over Mobile Networks. In *Proc. 5th International Workshop on Future Trends of Distributed Computing Systems*, Ceju Island, Korea.

Rodrigues, L. and P. Verissimo (1992). xAMP: a Multi-Primitive Group Commuication Service. In *Proc. 11th Symposium on Reliable Distributed Systems*, Houson, Texas, USA.

Russel, S. J. and S. Zilberstein (1991). Composing Real-Time Systems. In *Proc. 12th Int'l Joint Conf. on Artificial Intelligence*, Sydney, Australia.

Schemmer, S. (2000). Zuverlässige Echtzeit-Gruppenkommunikation auf einem lokalen Funknetz. Diploma, Rheinische Friedrich-Wilhelms-Universität**:** Bonn.

Schemmer, S. and E. Nett (2003a). Achieving Reliable and Timely Task Execution in Mobile Embedded Applications. In *Proc. 9th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003F)*, Capri Island, Italy.

Schemmer, S. and E. Nett (2003b). Managing Dynamic Groups of Mobile Systems. In *Proc. 6th International Symposium on Autonomous Decentralized Systems*, Pisa, Italy.

Schemmer, S., E. Nett, et al. (2001). Reliable Real-Time Cooperation of Mobile Autonomous Systems. In *Proc. 20th Symp. on Reliable Distributed Systems*, New Orleans, La.

Schmitt, T., S. Buck, et al. (2001). Cooperative Probabilistic State Estimation for Vision-based Autonomous Mobile Robots. In *Proc. IEEE Int'l Conf. on Intelligent Robots and Systems*.

Schmitt, T., R. Hanek, et al. (2002). Watch their Moves: Applying Probabilistic Multiple Object Tracking to Autonomous Robot Soccer. In *Proc. The Eighteenth National Conference on Artificial Intelligence*, Edmonton, Canada.

Sharon, O. and E. Altman (2001). An Effficient Polling MAC for Wireless LANs. *IEEE/ACM Transactions on Networking* **9**(4).

Sobrinho, J. L. and A. S. Krishnakumar (1996a). Distributed multiple access procedures to provide voice communication over IEEE 802.11 wireless networks. In *Proc. IEEE Global Telecommunications Conference (GLOBECOM): Communications: The key to global prosperty*, London, England.

Sobrinho, J. L. and A. S. Krishnakumar (1996b). Real-Time Traffic over the IEEE 802.11 Medium Access Control Layer. *Bell Labs Technical Journal* **1**(2): 172-187.

Spuri, M. and G. Buttazzo (1996). Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Real-Time Systems* **10**(2): 179-210.

Stankovic, J. A., M. Spuri, et al. (1998). *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Boston, Kluwer Academic Publishers.

Stroupe, M. M. and T. Balch (2001). Distributed sensor fusion for object position estimation by multi-robot systems. In *Proc. Int. Conf. on Robotics and Automation (ICRA'01)*.

Sun, Q. and D. Sturman (2000). A Gossip-Based Reliable Multicast for Large-Scale High-Throughput Applications. In *Proc. Int'l Conf. on Dependable Systems and Networks*, New York, N.Y.

Tanenbaum, A. S. (2003). *Computer Networks*. Upper Saddle River, Pearson Education International.

Trikaliotis, S. (2004). Utilizing Fault-Tolerance for Achieving QoS in Ad-hoc Networks. In *Proc. Int'l Conf. on Architecture of Computing Systems*, Augsburg, Germany.

van Hoesel, L. F. W., L. Dal Pont, et al. (2003). Design of an autonomous decentralized MAC protocol for wireless sensor networks (fast abstract). In *Proc. Sixth Int'l Symp. on Autonomous Decentralized Systems*, Pisa, Italy.

van Renesse, R., K. P. Birman, et al. (1996). HORUS: A Flexible Group Communication System. *Communications of the ACM* **39**(4): 76-83.

Vandersee, S. (2004). Effiziente Realisierung in SDL spezifizierter Mikroprotokoll-Architekturen. Diploma Thesis, Institute for Distributed Systems, Otto-von-Guericke-University**:** Magdeburg, Germany.

Verissimo, P., V. Cahill, et al. (2003). CORTEX: Towards Supporting Autonomous and Cooperating Sentient Entities. In *Proc. European Research on Middleware and Architectures for Complex and Embedded Cooperative Systems (held in conjunction with ISADS'03)*, Pisa, Italy.

Verissimo, P., A. Casimiro, et al. (2000). The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proc. Int'l Conf. on Dependable Systems and Networks*, New York City, USA.

Verissimo, P., J. Rufino, et al. (1991). Enforcing Real-Time Behavior on LAN-Based Protocols. In *Proc. 10th IFAC Workshop on Distributed Computer Control Systems*, Semmering, Austria, IFAC.

Wang, Z., Y.-Q. Song, et al. (2002). Survey of Weakly-Hard Real Time Schedule Theory and Its Application. In *Proc. Int'l Symp. on Distributed Computing and Applications to Business, Engineering and Science*, Wuxi, China.

Weber, J., K.-W. Jörg, et al. (2000). APR – Global Scan Matching Using An-chor Point Relationships. In *Proc. 6th Int'l Conf. on Intelligent Autonomous Systems*.

Appendix A — Formal Description of the Communication Hardcore

We present here a formal specification of the communication hardcore. We use SDL (Specification Description Language) as the formal language for the specification. SDL is a standard of the International Telecommunication Union (ITU) (ITU-T 1993,ITU-T 1999) and widely employed for the specification of communication systems, for which purpose it was originally developed. For example, the IEEE 802.11 Standard includes a formal specification in SDL. For descriptions of the SDL the reader is referred to (Braek and Haugen 1993,Mitschele-Thiel 2001).

In an introductory sub-section, the structure of the specification is explained. Subsequently, the specifications of the protocols are presented bottom-up. For each protocol, specifications of the services and interfaces, the PDUs, the AP entity, and the client entity are provided. The presentation of each protocol is split into two parts. The first part presents a version working with static groups, whereas the second part presents the extensions required to accommodate dynamic groups. The dynamic network scheduling and the membership protocol are presented in a single step, as they are not intended to be used in static groups.

A.1 Structure of the Formal Model

In this section, we introduce the common structure of the following sections. Each section comprises the same sequence of sub-sections. Following the approach we adopted in Chapter 4, we present the protocols in two steps: First, a version for static groups, and subsequently, the extensions required to support dynamic groups will be presented. Hence, there will be two sections for most of the protocols. Protocols that are only used in dynamic groups — that is to say, the dynamic network scheduling and the membership protocol — will be presented in a single section.

As explained above, there are two kinds of stations: the AP and the clients. The AP and the clients exhibit asymmetric behaviors during the execution of the protocol. Accordingly, each protocol has two roles; that is, each layer consists of two kinds of layer entities, each with its own behavioral specification. Therefore, we specify an AP and a client role for each protocol.

All specifications belonging to the same layer are combined in an SDL package. Such a package contains:

1. The specifications of the signals that are the service primitives of that layer. This includes also those data types that are used to declare the parameters of the signals;

2. The specification of the PDUs;

3. The specification of the AP role;

4. The specification of the Client role.

Correspondingly, each of the following sections has four sub-sections, named “Services and Interfaces”, “PDUs”, Specification of the AP Role”, and “Specification of the Client Role”. Some of the sub-sections may be omitted in sections presenting the dynamic group extensions of some protocol if there are no extensions to be presented in that sub-section. For example, if no new PDUs need to be introduced to support dynamic groups, sub-section “PDUs” will be omitted.

In the “Services and Interfaces” sub-section, the signals exchanged between the user and the provider of the service are specified. The signal names observe the following naming convention: <service_name>_<kind_of_signal>. We distinguish five kinds of signals *poll, rqu, ind, rsp, cnf.* To request a service *X* the user sends a signal *X_rqu* to the service provider. There are services, called polled services, where the user is not free to send the *X_rqu* at any time but only after receiving a polling signal *X_poll* from the service provider. For example, the *AP_DATA* service of the polling protocol is a polled service, where the AP role sends an *AP_DATA_poll* signal to its user to solicit an *AP_DATA_rqu*. The provider of a service *X* sends the signal *X_ind* to its user to indicate that a relevant event w.r.t to that service has occured; for example, the client role of the polling protocol sends an *AP_DATA_ind* to its user when it receives a data frame from the AP. We call a service a responded service if the provider expects a signal *X_rsp* in response to the indication. Consider as an example the service *FAIL* of the polling protocol: After indicating that a client became invalid, the AP role waits until it receives a *FAIL_rsp* signal from the user in order to give the user the chance to change the polling list in reaction to the indication. Finally, a service is called a confirmed service if the provider sends a signal *X_cnf* to indicate whether or not the requested services has been executed successfully. For example, a user that initiates the *JOIN* service at the client role of the polling protocol gets a *JOIN_cnf* signal from the client role, when the client has been added to the polling list at the AP.

In the architecture, there is the concept of service “forwarding” or “inheriting” between successive layers. A layer inherits a service from the lower layer if it uses that service and also provides it to the following layer. This allows using a service initially realized on a low layer of the architecture on all the following layers. Consider, for example, the *EXT_POLL* service of the polling protocol. Not only the layer above the polling layer may be interested in piggybacking data on the polling frames but also the layers above. This kind of “forwarding” or “inheriting” a service allows stacking layers above each other that all depend on the same service. If a layer inherits a service from a lower layer, the corresponding signals need not be defined a second time in the specification of the higher layer.

Following the external, service-oriented view, the following sub-sections turn to the internals of the layer. Sub-section "PDUs" presents the definition of PDUs the protocol entities exchange. There are typically several types of PDUs each with its own internal structure. Each PDU type is defined as a SDL data type. For example, in the polling protocol the peers exchange *poll*, *data*, and *null* PDUs.

The next two sub-sections present the specifications of the protocol roles. Each protocol role is modeled as a SDL block type. The block type definitions of the protocol roles typically have a structure as depicted in Figure A-1.

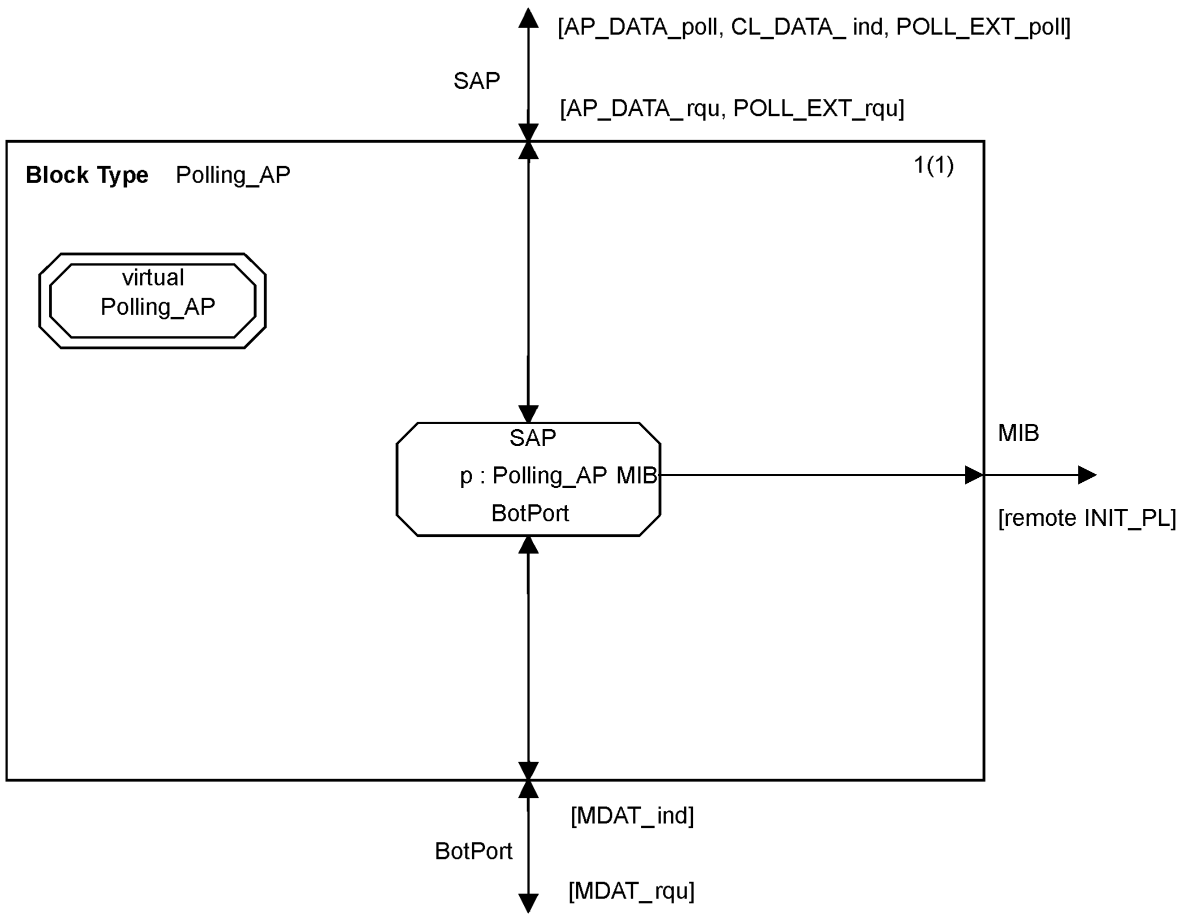


Figure A-1. Example of the block type definition of a protocol role

The block type definition diagram reveals the internal structure of the protocol role. It depicts the SDL processes running in each instance of the protocol role (the single-lined six-cornered box in Figure A-1), the signal flows between these processes, and the signal flows between the processes and the environment of the protocol role. Most of the protocol roles encompass a single process, like in the example above, but there are also roles that comprise two processes — like, for example, the reliable multicast protocol. For sake of brevity, we omit block type diagrams that include only a single process and exhibit the same structure as the example in Figure A-1.

Each process in a protocol role is the instance of a process type. For example, in Figure A-1, the process *p* is an instance of the process type *Polling_AP*; the douple-lined six-cornered box in the diagram represents a reference to the definition of the process type. The specification of the behavior of a process is provided in the definition of its process types. For each process type used in a protocol role its definition will be presented in the SDL Textual Phrase Representation (SDL/PR). The definitions of the process types therefore represent the definition of the protocol's behavior. The SDL/PR representation style is somewhat more space efficient than the graphical representation, and since the single process type definitions are not too complex, it should be similarly comprehendible as the graphical representation. Each process type definition describes an *extended finite state machine* (EFSM). An EFSM has a finite number of explicit, discrete states. Additionally, EFSMs are allowed to contain variables as well, so that the values of these variables are part of the state of the machine too. EFSMs communicate via signals. When an EFSM receives a signal it performs a state transition. During a transition, the EFSM can output signals or change the values of its variables.

The points where the instances of a process type connect with their environment are called gates. For example, process *p* in Figure A-1 has three gates: *SAP*, *BotPort*, and *MIB*. If a protocol role contains a single process, the gates of that process are connected to the corresponding gates of the block type representing the protocol role (In Figure A-1, the arrows outside of the box denote the gates of the block type). Thus, each instance of the protocol role has the same three connection points *SAP*, *BotPort*, and *MIB*. The gate *SAP* represents the service access point (SAP) of the protocol where it exchanges signals with its user. At the gate *BotPort*, the protocol role accesses the services of the lower layer. Thus, in a stack, the *BotPort* of a layer *n* entity is connected to *SAP* of the layer *n*-1 entity. Additionally, a protocol role may have a *MIB* gate where it accesses the services of the *Management Information Base (MIB)*, which is a common information repository for the stack. For each process type it is specified what signals it sends or receives across the gates.

In the presentation of the dynamic group extensions of a protocol, the first two subsections introduce new services and signals and new PDU types respectively, which have been added to support dynamic groups. To add dynamic group support in the specifications of the protocol roles, we use inheritance. To extend a protocol role, we define a sub-type of the block type representing that role. The new block type may add new structural elements such as processes and gates to its super-type or redefine the properties of existing elements. The most important kind of specialization usually performed is to redefine the process types within the protocol roles. Consider, for example, the polling protocol. In the extended version of the AP role, the process type *Polling_AP*, which describes the behavior of the AP role, is redefined to change the behavior of the role. The redefined process type inherits the specification of the process type it refines. When redefining a process type, one can add new states, new variables, and new transitions, or change transitions of the refined type. Using the inheritance/redefinition facility of the SDL, we are able focus on those aspects that need to be added to or changed in the protocol to support dynamic groups.

A.2 Polling

Services and Interfaces

```
/* ----------------------------------------------------------------------------

  SERVICE: AP_DATA, AP initiated


  Transmit data from the AP to the clients. Data is transmited on behalf of the
  owner of the current polling list entry. Polled service

  SIGNALS

  AP_DATA_poll(s), AP
  's' station on behalf of which frame transmission may be requested

  AP_DATA_rqu(a,s), AP
  AP requests transmission of SDU 's' to destination 'a'. Source address is
  the station mentioned in the correspoding AP_DATA_poll

  AP_DATA_ind(SA,s), Client
  Indicates SDU 's' from the AP. 'SA' is the source address

  ---------------------------------------------------------------------------*/

signal /* SERVICE AP_DATA */
  AP_DATA_ind(AddressType,SDU),
  AP_DATA_rqu(AddressType,SDU),
  AP_DATA_poll(StationId);


/* -----------------------------------------------------------------------------

  SERVICE: CL_DATA, Client initiated

  SIGNALS:

  CL_DATA_poll(s), Client
  Poll for transmission of a frame to the AP. 's' is an SDU that was
  provided at the AP over the POLL_EXT service (see SERVICE POLL_EXT).

  CL_DATA_rqu(DA,s), Client
  Request to transmit SDU 's' to the AP. 'DA' is the destination address.

  CL_DATA_ind(sid,s), AP
  Indicates that SDU 's' was received. 'sid' is the StationID of the source
  station.

  ----------------------------------------------------------------------------*/

signal /* SERVICE CL_DATA */
  CL_DATA_rqu(AddressType,SDU),
  CL_DATA_poll(SDU),
  CL_DATA_ind(StationId,SDU);


/* ----------------------------------------------------------------------------

  SERVICE POLL_EXT, AP initiated

  Allows the user to transmit SDUs in the polling frames. At the client, the
  SDU will be indicated in the CL_DATA_poll primitiv.

  SIGNALS:

  POLL_EXT_poll(sid)
  Poll for an SDU to be transmitted in the following poll to station 'sid'.

  POLL_EXT_rqu(s)
  Request transmission of SDU 's' in the following poll.

  ---------------------------------------------------------------------------*/

signal /* SERVICE POLL_EXT */
  POLL_EXT_poll(StationId),
  POLL_EXT_rqu(SDU);


/* -----------------------------------------------------------------------------

  SERVICE FAIL, AP only

  Notify that station has became invalid

  SIGNALS:
  FAIL_ind(sid)
  'sid' ID of the failed station

  FAIL_rsp.

  ---------------------------------------------------------------------------*/

signal /* SERVICE FAIL */
  FAIL_ind(StationId),
  FAIL_rsp;


/* -----------------------------------------------------------------------------

  SERVICE PL_SET, AP only

  Set the polling list

  SIGNALS:

  PL_SET_rqu(pl)
  Request to set the polling list to 'pl'.

  ---------------------------------------------------------------------------*/

signal /* SERVICE PL_SET */
  PL_SET_rqu(PollingList);


/*----------------------------------------------------------------------------

   PollingListEntry

   The polling list consists of polling list entries. Each entry corresponds to
   a reservation of the medium.  Attributes:

   owner: station ID of the client who owns the reservation
   entry_type:
   - poll: poll a client
   - relay: AP is allowed to transmit a message on behalf of a certain
     client
   - jpoll: Broadcast a polling message including a station ID
   stack: PID of the protocol stack that this reservation belongs to.
   pePeriodic: Slot is periodic/sporadic

  --------------------------------------------------------------------------- */
```

```
value type PollingListEntryType;
  literals poll,relay,jpoll;
endvalue type;

value type PollingListEntry
  struct
  peType  PollingListEntryType,
  peOwner StationId,
  peStack PId,
  pePeriodic Boolean
endvalue type;

value type PollingString inherits MyString<PollingListEntry>;

value type PollingList;
  struct
  list PollingString;
  cur Integer;

  operators

    /* get current polling list entry */
    cur(this PollingList) -> PollingListEntry;

    /* get address of the current slot owner */
    cur(this PollingList) -> AddressType;

    /* remove all entries with owner station id */
    remove(this PollingList,StationId) -> this PollingList;

    /* remove the current entry from the polling list */
    remove_cur(this PollingList) -> this PollingList;

    /* add entry of given type and owner after the current position
       increment current position */
    add(this PollingList,PollingListEntryType,StationId,PId) -> this PollingList;

    /* add polling list entry after given position */
    add(this PollingList,Natural,PollingListEntry) -> this PollingList;

    /* next entry becomes current entry */
    iterate(this PollingList) -> this PollingList;

    /* true if StationId has an entry in the polling list */
    has_rgcp_slot(this PollingList, StationId) -> Boolean;

endvalue type PollingList;
```

PDUs

```
value type Poll_PDUTypes;
  literals null,poll,data,jpoll,jrqu;
endvalue type;

value type Poll_PDU inherits PDU;
  struct
  pduType Poll_PDUTypes;
endvalue type Poll_PDU;

value type Poll_PDU_poll inherits Poll_PDU;
  struct
  sdu SDU;
endvalue type Poll_PDU_poll;

value type Poll_PDU_data inherits Poll_PDU ;
  struct
  sdu SDU;
endvalue type Poll_PDU_data;
```

Specification of the AP Role

```
virtual process type Polling_AP ;

  gate SAP
    in with AP_DATA_rqu, POLL_EXT_rqu;
    out with CL_DATA_ind,
     AP_DATA_poll,
     POLL_EXT_poll;

  gate BotPort
    in with MDAT_ind, MDAT_STATUS_ind;
    out with MDAT_rqu;

  gate MIB
    out with remote INIT_PL;

  /* constants */
  synonym
  toValPoll Duration = 2*messageDelay;

  /* state */
  dcl
  exported pl PollingList :=
   (. emptystring,0 .);
  timer toPoll := toValPoll;

  /* temp vars */
  dcl
  s AddressType,
  SA,DA,RA AddressType,
  msg SDU,
  p Poll_PDU;

  virtual procedure pollNext;

    dcl
    sdu SDU;

    start virtual;
      decision cur(pl).pePeriodic;
      (false):
        task pl := remove_cur(pl);
      enddecision;

      task pl := iterate(pl);
      decision cur(pl).peType;
      (relay):
        output AP_DATA_poll(
          cur(pl).peOwner)
          to cur(pl).peStack;
        return;
      (poll):
        output POLL_EXT_poll(
          cur(pl).peOwner)
          to cur(pl).peStack;
        nextstate wait_rqu_ext;
      enddecision;

    state wait_rqu_ext;
      input POLL_EXT_rqu(sdu);
        output MDAT_rqu(
          APAddress,cur(pl),
          SDU(mk_poll(msg)) );
        set(toPoll);
        return;
    endstate wait_rqu_ext;

  endprocedure pollNext;

  start;
    task pl := import(INIT_PL),
    output SET_PL_rqu(pl);
    call pollNext ;
    nextstate  normal ;

  state normal ;
    input AP_DATA_rqu(DA,msg);
      output MDAT_rqu(DA,cur(pl),msg);
      nextstate wait_ind;

    input virtual MDAT_ind(SA,DA,msg) ;
      l1 : reset(toPoll),
      p := Poll_PDU(msg);
      decision p.pduType;
      (null) : join go_on;
      else:
        output CL_DATA_ind(
        cur(pl).peOwner,msg);
        join go_on;
      enddecision;
      go_on:
      call pollNext;
      nextstate normal ;

      input virtual toPoll ;
        call pollNext;
        nextstate  normal ;
  endstate;

  state wait_ind;
    input MDAT_STATUS_ind;
      call pollNext;
      nextstate normal ;
  endstate wait_ind;

endprocess type;
```

Specification of the Client Role

```
process type Polling_CL;

  gate SAP
  in with CL_DATA_rqu ;
  out with AP_DATA_ind,CL_DATA_poll ;

  gate BotPort
  in with MDAT_ind ;
  out with MDAT_rqu ;

  gate MIB
  out with remote OWNADDR ;

  /* input vars */
  dcl
  SA,DA AddressType,
  OA AddressType, /* own address */
  sdu SDU;

  /* tmp vars */
  dcl
  p Poll_PDU,
  p_data Poll_PDU_data;

  start;
    task OA := import(OWNADDR) ;
    nextstate  WaitPoll;

  state WaitRqu ;
    input CL_DATA_rqu(DA,sdu);
      decision sdu;
        (''B) :
        output MDAT_rqu(
          OA,DA,SDU(mk_null));
        nextstate - ;
      else:
        output MDAT_rqu(OA,DA,
          SDU(mk_data(sdu)));
        nextstate - ;
      enddecision;
  endstate;

  state WaitPoll ;
    input MDAT_ind(SA,DA,sdu);
      task p := Poll_PDU(sdu);
      decision p.pduType ;
      (data) :
        data :
        output AP_DATA_ind(
          SA,Poll_PDU_data(sdu).sdu);
        nextstate - ;
      (poll) :
        poll :
        output CL_DATA_poll(sdu);
        nextstate  WaitRqu ;
      else:
        nextstate - ;
      enddecision;
  endstate;
endprocess type;
```

A.3 Polling – Dynamic Group Extensions

Services and Interfaces

```
/* ---------------------------------------------------------------------------

  SERVICE JOIN

  Allows a station that is not yet part of the APs polling list to transmit
  data.

  SIGNALS:

  JOIN_init(sid), Client
  Initiate the service. 'sid' is the identifier (track identifier, e.g.) to
  react to.

  JOIN_poll, Client
  Poll for SDU

  JOIN_rqu(s), Client
  Request SDU 's' be transmitted to the AP.

  JOIN_ind(SA,s), AP
  Indicates that a SDU 's' was received after a jpoll. 'SA' is the
  address of the source station.

  JOIN_rsp, AP
  User signals it has reacted to JOIN_ind.

  JOIN_cnf, Client
  Confirms that the station's being added to APs polling list.

  JOIN_START_ind, Client
  Indicate that joining started

  ---------------------------------------------------------------------------*/

signal /* SERVICE JOIN */
  JOIN_init(StationId),
  JOIN_poll,
  JOIN_rqu(SDU),
  JOIN_cnf,
  JOIN_ind(AddressType,SDU),
  JOIN_rsp,
  JOIN_START_ind;
```

PDUs

```
value type Poll_PDU_jpoll inherits Poll_PDU ;
  struct
  st StationId;
  sdu SDU;
endvalue type Poll_PDU_jpoll;

value type Poll_PDU_jrqu inherits Poll_PDU;
  struct
  sdu SDU;
```

```
endvalue type Poll_PDU_jrqu;
```

Specification of the AP Role

```
redefined process type Polling_AP ;

  gate SAP adding
  in with JOIN_rsp, FAIL_rsp,
    PL_SET_rqu ;
  out with JOIN_ind,
    FAIL_ind,SET_PL_rqu ;

  redefined procedure pollNext;

    start redefined;
      task
      pl := iterate(pl);
      output SET_PL_rqu(pl);
      decision cur(pl).peType;
      (relay):
        output AP_DATA_poll(
          cur(pl).peOwner)
          to cur(pl).peStack;
        return;
      (poll):
        output POLL_EXT_poll(
          cur(pl).peOwner)
          to cur(pl).peStack;
        nextstate wait_rqu_ext;
      (jpoll):
        output MDAT_rqu(
          APAddress,bcAddress,
          SDU(
          mk_jpoll(cur(pl).peOwner)) );
        set(toPoll);
        return;
      enddecision;

  endprocedure pollNext;


  value type StationType
    struct
    nl Natural = 0;
  endvalue type StationType;
  synonym StationNull StationType =
   (. 0 .);

  value type StationString
    inherits MyString<StationType>;
  endvalue type;

  dcl
  station StationString;

  state wait_new_rsp ;
      input JOIN_rsp ;
        join_rsp:
        call pollNext ;
        nextstate normal ;
  endstate;

  state wait_PF_rsp ;
     input FAIL_rsp ;
       join join_rsp:
  endstate;

  state normal ;
     input redefined MDAT_ind(
        SA,DA,msg);
       decision cur(pl).peType;
         (jpoll) :
         reset(toPoll) ;
         output JOIN_ind(SA,msg);
         nextstate wait_new_rsp;
       else:
         task station(
           cur(pl).peOwner).nl := 0 ;
         join l1 ;
       enddecision;

     input redefined toPoll ;
       decision cur(pl).peType;
         (jpoll) : join join_rsp;
       else:
         task station(
          cur(pl).peOwner).nl :=
          station(cur(pl).peOwner).nl
          +1;
         decision station(
          cur(pl).peOwner).nl ;
           (> OD) :
           output FAIL_ind(
             cur(pl).peOwner);
           nextstate  wait_PF_rsp ;
         else:
           join join_rsp;
         enddecision;
       enddecision;

  endstate;

  state*;
     input PL_SET_rqu(pl) ;
       SET_PL_rqu(pl) ;
       nextstate - ;
  endstate;

endprocess type Polling_AP;
```

Specification of the Client Role

```
redefined
process type Polling_CL ;

  gate SAP adding
    in with JOIN_init,JOIN_rqu ;
    out with JOIN_cnf,JOIN_poll ;
      JOIN_START_ind;
  /* state vars */
  dcl
  st StationId := 0;

  /* temp vars */
  dcl
  jp Poll_PDU;

  state wait_JOIN_rqu;
    input JOIN_rqu(sdu);
      output MDAT_rqu(OA,
        APAddress,SDU((. jrqu,sdu .)));
      nextstate  joining ;
  endstate;

  state WaitPoll ;
    input JOIN_init(st) ;
      output JOIN_START_ind;
      nextstate  joining ;
  endstate;

  state joining ;
    input MDAT_ind(SA,DA,sdu);
      task p := Poll_PDU(sdu);
      decision p.pduType;
        (jpoll) :
          decision Poll_PDU_jpoll(
            sdu).st = st;
            (true) :
              output JOIN_poll ;
              nextstate  wait_JOIN_rqu;
            (false) :
              nextstate joining ;
          enddecision;
        (data) :
          join data ;
        (poll) :
          output JOIN_cnf ;
          join poll ;
      enddecision;
  endstate;

endprocess type Polling_CL;
```

A.4 Dynamic Network Scheduling

Services and Interfaces

```
/* ---------------------------------------------------------------------------

   SERIVCE: EXCL, AP only, responded service

   Exclude clients invalid clients.

   SIGNALS:

   EXCL_ind(s):
   Indicates that client with StationId 's' has been excluded.

   EXCL_rsp:
  --------------------------------------------------------------------------- */

signal /* SERVICE EXCL */
  EXCL_ind(StationId),
  EXCL_rsp


/* ---------------------------------------------------------------------------

   SERVICE: RESMED, Client initiated, confirmed service

   Reserve bandwidth by requesting to be added to the polling list. For the time
   being only reservation for the RGCP are supported.

   SIGNALS:

   RESMED_rqu(id, type), Client
   request to be added to the polling list. 'id' is the track id used for
   the 'JOIN' service of the polling protocol. 'type' indicates the type
   of reservation requested.

   RESMED_cnf, Client

  --------------------------------------------------------------------------- */

signal /* SERVICE RESMED */
  RESMED_rqu(StationId,ResType),
  RESMED_cnf;

value type ResType;
  literals rgcp;
endvalue type;


/* ---------------------------------------------------------------------------

   SERVICE: NEW, AP only, responded service

   Notification that new member has been added.

   SIGNALS

   NEW_ind(s)
   's': StationID of the client.

   NEW_rsp
  ----------------------------------------------------------------------- */

signal /* SERVICE NEW */
  NEW_ind(StationId),
  NEW_rsp;
```

PDUs

```
value type DMA_PDU;
  struct
  res_type ResType;
  sdu SDU;
endvalue type;
```

Specification of the AP Role

```
process type DynMedAcc_AP;

  /* signature */

  gate BotPort
    in with FAIL_ind,JOIN_ind;
    out with FAIL_rsp,JOIN_rsp,
      PL_SET_rqu, remote PL;

  gate MIB
    out with procedure IDADDR_SET,
      remote IDADDR,
      remote STACKPID,
        remote EXCL_NOT;

  gate SAP
    in with EXCL_rsp, NEW_rsp;
    out with EXCL_ind, NEW_ind,
      CL_DATA_ind;

  /* tmp vars */
  dcl
  s StationId,
  a AddressType,
  sdu SDU,
  p DMA_PDU,
  id2addr AddrIdArray,
  allocated Boolean;

  procedure allocate_rgcp;
    returns Boolean;

   dcl
    sp PId ,
    pl PollingList;

    start;
      decision s = no_id;
      (true): task
        id2addr := add(id2addr,a),
        s := id(id2addr,a);
        call IDADDR_SET(id2addr);
      (false):
      enddecision;
      task
      sp := import(STACKPID)(rgcp),
      pl := import(PL);
      decision has_rgcp_slot(pl,s);
      (true): return false;
      (false):
        output PL_SET_rqu(
          add(add(pl,poll,s,sp),
          relay,s,sp));
        return true;
      enddecision;

endprocedure allocate_rgcp;

procedure free_rgcp;

  dcl
  sp PId ,
  pl PollingList,
  j Natural;

  start;
    task
    pl := import(PL),
    sp := import(STACKPID)(rgcp);
    output EXCL_ind(s) to sp;
    nextstate wait_EXCL_rsp;

  state wait_EXCL_rsp;

    input EXCL_rsp;
      task 'let j be the position of
        the rgcp polling slot of s';
      decision import(EXCL_NOT);
      (true):
        task pl :=
        add(pl,j,
        (. relay,s,sp,false .));
      (false):
      enddecision;
      task pl :=
          remove(remove(pl,j+1),j+1),
          pl.cur := if j = pl.cur
          then (pl.cur - 1) mod
            length(pl.list)
          else pl.cur fi;
        output PL_SET_rqu(pl);
        return;

    endstate wait_EXCL_rsp;

  endprocedure free_rgcp;

  start;
    nextstate normal;

  state normal;

    input JOIN_ind(a,sdu);
      task
      p := DMA_PDU(sdu),
      id2addr := import(IDADDR),
      s := id(id2addr,a);
      decision p.res_type;
      (rgcp):
        task allocated :=
          call allocate_rgcp;
      else:
      /* other types are not yet
         handled */
      enddecision;

      decision allocated;
      (true): output NEW_ind(s);
      (false): output JOIN_rsp;
      enddecision;
      nextstate -;

    input NEW_rsp;
      output CL_DATA_ind(s,p.sdu);
      output JOIN_rsp;
      nextstate normal;

    input FAIL_ind(s);
      call free_rgcp;
      output FAIL_rsp;
      nextstate normal;

  endstate normal;

endprocess type DynMedAcc_AP;
```

Specification of the Client Role

```
process type DynMedAcc_CL;

  /* signature */

  gate SAP
    in with RESMED_rqu;
    out with RESMED_cnf;

  gate BotPort
    in with JOIN_poll, JOIN_cnf;
    out with JOIN_rqu, JOIN_init;

  /* state */

  dcl
  t ResType;   /* reservation type */


  /* tmp vars */
  dcl
  rid StationId;

  start;
    nextstate idle;

  state idle;

    input RESMED_rqu(rid,t);
      output JOIN_init(rid);
      nextstate reserving;

  endstate idle;

  state reserving;

    input JOIN_poll;
      output JOIN_rqu(SDU(
        (. t,''B .)));
      nextstate reserving;

    input JOIN_cnf;
      output RESMED_cnf;
      nextstate idle;

  endstate reserving;

endprocess type DynMedAcc_CL;
```

A.5 Reliable Multicast

Services and Interfaces

```
/* -------------------------------------------------------------------------

PROTOCOL: RelMul

SERVICE RELMC, Client only

Transmission of reliable multicasts.


SIGNALS

RELMC_poll: poll for RELMC_rqu.
RELMC_rqu(r,sdu): Request transmission of sdu. 'r' is the resiliency of the
  message.
RELMC_ind(a,sdu): Deliver reliable multicast 'sdu'. Source station's address
  is a.

  ------------------------------------------------------------------------ */

signal /* SERVICE RELMC, client only */
    RELMC_poll,
    RELMC_rqu(Natural, SDU),
    RELMC_ind(AddressType,SDU);


/* -------------------------------------------------------------------------

SERVICE PB_BC, AP initiated

Allows a user at the AP to transmit data via piggy-backing in the mc frames the
AP sends for reliable multicast transmission. The service deliver a global
sequence number at the receiver user together with the piggy-backed SDU.

SIGNALS

PB_BC_poll(sg): AP. Poll for PB_BC_rqu. The SDU provided in the rqu will be
transmitted on the mc frame with global sequence no. 'sg'.

PB_BC_rqu(sdu): AP. Request transmission of SDU via piggy-backing. sdu should
be small.

PB_BC_ind(sdu,sg): Client. Piggy-bagged data received in a mc frame. 'sg' global
seq. no. of the mc frame

  ------------------------------------------------------------------------ */

signal /* SERVICE PB_BC */
    PB_BC_poll(StationId,Natural),
    PB_BC_rqu(SDU),
    PB_BC_ind(SDU,Natural);


/* -------------------------------------------------------------------------

SERVICE RELMC_STAT, AP only

Indicates the status of multicast messages when the protocol terminates their
transmission.
```

```
SIGNALS

RELMC_STAT_ind(s,stat) indicates the status of a multicast message send on
behalf of client with station ID s, when the protocol terminates the
transmission.

RELMC_STAT_rsp(b)

  ---------------------------------------------------------------------------- */

/* SERVICE RELMC_STAT, AP only */

value type StatusEnum;
  literals acked, ODretries, resretries;
endvalue type;

signal
    RELMC_STAT_ind(StationId,StatusEnum),
    RELMC_STAT_rsp(Boolean);

/* ---------------------------------------------------------------------------

SERVICE ACKS, AP only

Delivers the acks of clients.

SIGNALS

ACKS_ind(s,a): Indicates that client with station ID 's' has sent acknowledgments
for all stations in StationIDSet 'a'.

  ---------------------------------------------------------------------------- */

signal /* SERVICE ACKS, AP only */
    ACKS_ind(StationId,StationIdSet);
```

PDUs

```
/* ---------------------------------------------------------------------------

   PDUs

   ----------------------------------------------------------------------------*/

value type BooleanString inherits MyString<Boolean>;
  operators
  Octetstring : this BooleanString -> Octetstring;
endvalue type BooleanString;

value type RM_Poll_PDU inherits PDU;
    struct
    r   Natural;
    sg  Natural;
    sdu SDU;
endvalue type RM_Poll_PDU;

value type RM_Rqu_PDU inherits PDU;
    struct
    sl Natural;
    res Natural;
    acks BooleanString;
    sdu SDU;
endvalue type RM_Rqu_PDU;

value type RM_Bc_PDU inherits PDU;
    struct
    sl Natural;
    sg Natural;
    piggyLoad SDU;
    sdu SDU;
endvalue type RM_Bc_PDU;

/* ----------------------------------------------------------------------------

  INTERNAL SERVICES

  INT SERVICE RELMC_RQU_ind, AP only

  indicates that the client has transmitted a request to multicast a sdu.

  SIGNALS

  RELMC_RQU_ind(s,sl,r,sdu): 's' the station ID of the source station, 'sl' the
  local seq. no. of the SDU, 'r' the resiliency of the SDU, 'sdu' the SDU.

   ---------------------------------------------------------------------------*/

signal /* SERVICE RELMC_RQU_ind */
    RELMC_RQU_ind(StationId, Natural, Natural, SDU);

/* -----------------------------------------------------------------------------

   package global classes

   ----------------------------------------------------------------------------*/

value type Tuple;
  struct
  ad AddressType;
  sl Natural;
endvalue type Tuple;

value type TupleCache inherits MyString<Tuple>;
  operators
    search : this TupleCache, AddressType -> Natural;
    update : this TupleCache, Natural, Tuple -> this TupleCache;
    insert : this TupleCache, Tuple -> this TupleCache;
endvalue type TupleCache;
```

Specification of the AP Role

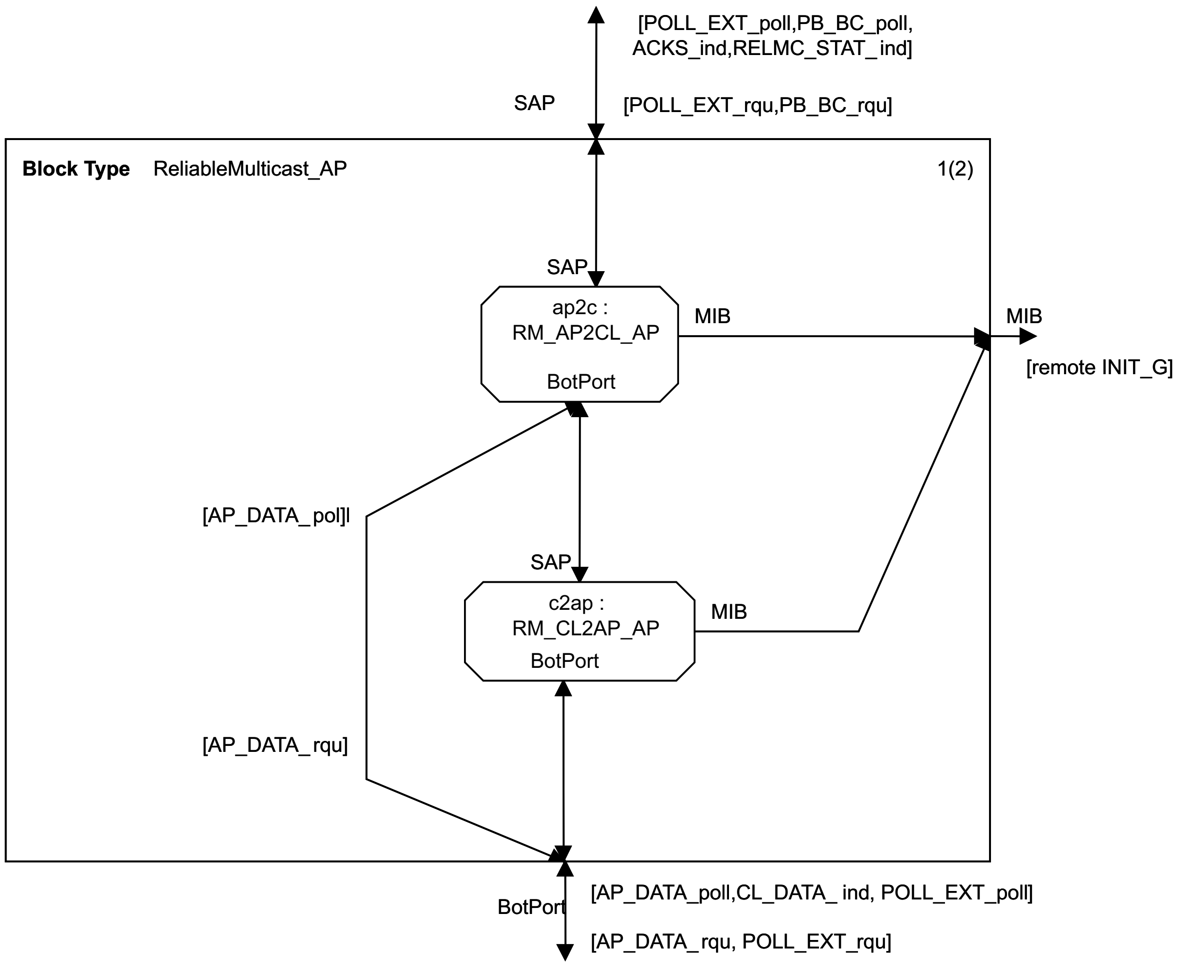


Figure A-2. Reliable multicast protocol, AP role

```
virtual process type RM_CL2AP_AP;

  gate SAP
  in with POLL_EXT_rqu;
  out with POLL_EXT_poll, CL_DATA_ind,
    RELMC_RQU_ind;

  gate BotPort
  in with POLL_EXT_poll,CL_DATA_ind;
  out with POLL_EXT_rqu;

  gate MIB
  out with remote INIT_G;

  /* types */
  value type Station;
    struct
    r Natural := 0;
    sl Natural := 0;
  endvalue type Station;
  synonym StationNull = (. 0,0 .);

  value type StationList
    inherits MyString<Station>;
endvalue type StationList;

/* state */
dcl
station StationList,
s StationId;

/* tmp vars */
dcl
p RM_Rqu_PDU,
RA,TA AddressType,
sdu SDU;
pp RM_Poll_PDU;


start;
  task station  := mkstr(StationNull,
    setsize(import(INIT_G)));
  nextstate normal;

state normal;

  /* Eliminate duplicates */
  input CL_DATA_ind(s,sdu);
    task p := RM_Rqu_PDU(sdu);
    decision station(s).sl = 0 or
```

```
      p.sl > station(s).sl;
    (true):
      output RELMC_RQU_ind(
        s,p.sl,p.res,sdu);
      task station(s).sl := p.sl;
    else:
    enddecision;
    output CL_DATA_ind(s,sdu);
    nextstate -;

  input POLL_EXT_poll(s);
    output POLL_EXT_poll(s);
    nextstate -;

  /* Add round no. */
  input POLL_EXT_rqu(sdu);
    task
    pp := RM_Poll_PDU(sdu),
    station(s).r := station(s).r+1,
    pp.r := station(s).r;
    output POLL_EXT_rqu( SDU(pp) );
    nextstate -;

  endstate normal;

endprocess type RM_CL2AP_AP;



virtual process type RM_AP2CL_AP;

  gate BotPort
  in with AP_DATA_poll, POLL_EXT_poll,
    CL_DATA_ind,RELMC_RQU_ind;
  out with AP_DATA_rqu,POLL_EXT_rqu;

  gate SAP
  out with POLL_EXT_poll,PB_BC_poll,
    RELMC_STAT_ind, ACKS_ind;
  in with POLL_EXT_rqu,PB_BC_rqu;

  gate MIB
  out with remote INIT_G;


  /* types */
  value type QueueItem;
    struct
    sl Natural;
    res Natural;
    sdu SDU;
    tr Natural;
  endvalue type QueueItem;
  value type WaitQueue
    inherits MyString<QueueItem>;
  endvalue type WaitQueue;
  value type TransmState;
    literals idle,wait4ack;
  endvalue type TransmState;

  value type Station;
    struct
    wQ WaitQueue;
    lp Natural;
    acked StationIdSet;
    trstate TransmState;
  endvalue type Station;
  synonym StationNull Station
    = (. emptystring,0,empty,idle .);
```

```
value type StationList
  inherits MyString<Station>;
endvalue type StationList;

value type StationIDList
  inherits MyString<StationId>;
endvalue type StationIDList;

/* state */
dcl
g StationIdSet,
station StationList,
polled_stations StationIDList
  := emptystring,
sg Natural := 0,
s StationId;

/* tmp vars */
dcl
o Natural,
SA AddressType,
sdu SDU,
res, sl Natural,
p RM_Rqu_PDU,
l Natural,
ackedIds StationIdSet;


start;
  task
  g := import(INIT_G),
  station := mkstr(StationNull,
    setsize(g));
  nextstate normal;

state normal;

  input virtual AP_DATA_poll(s);
    decision station(s).trstate;
    (wait4ack):
      decision g <= station(s).acked;
      (true):
        output RELMC_STAT_ind(
          s,acked);
        task
          station(s).wQ :=
          tail(station(s).wQ),
          station(s).acked := empty;
      (false):
        not_acked:
        decision
          first(station(s).wQ).tr =
          first(station(s).wQ).res+1;
        (true):
          decision
            first(station(s).wQ).res
            = OD;
          (true):
            output RELMC_STAT_ind(
              s,ODretries);
          (false):
            output RELMC_STAT_ind(
             s,resretries);
          enddecision;
          task
            station(s).wQ :=
              tail(station(s).wQ),
            station(s).acked :=
              empty;
        (false):
        enddecision;
    enddecision;
  (idle):
  enddecision;

  output PB_BC_poll(s,sg);
  nextstate -;

input PB_BC_rqu(sdu);
  decision station(s).wQ
    /= emptystring;
  (true):
    output AP_DATA_rqu(bcAddress,
    SDU ((.
       first(station(s).wQ).sl,
       sg,sdu,
       first(
        station(s).wQ).sdu .) ));
    task
      station(s).wQ(1).tr
      := station(s).wQ(1).tr+1,
      station(s).trstate :=
        wait4ack;
  else:
    output AP_DATA_rqu(bcAddress,
      SDU(  (. 0,sg,sdu,''B .) ));
    task
      station(s).trstate := idle;
  enddecision;
  nextstate -;

/* eval acks */
input CL_DATA_ind(s,sdu);
  task
  p := RM_Rqu_PDU(sdu),
  ackedIds := empty;
  task '{
  for(dcl i := 0,
  i < length(polled_stations)-1,
  i := i+1)
    {
      if (p.acks(i) = true) {
      station(
       polled_stations(i+1)).sacked
            := incl(s,
                station(
                 polled_stations(i+1)
                 ).sacked);
              ackedIds := incl(
                polled_stations(i+1),
                ackedIds);
            }
          }
        }';
        output ACKS_ind(s,ackedIds);
        nextstate -;

      input RELMC_RQU_ind(s,sl,res,sdu);
        task
        p := RM_Rqu_PDU(sdu),
        station(s).wQ := station(s).wQ
          // mkstring(
             (. sl,res,sdu,0 .));
        nextstate -;

      input POLL_EXT_poll(s);
        task
        sg := sg + 1,
        polled_stations := mkstring(s)
          // polled_stations,
        l := sg - station(s).lp+1,
        polled_stations := substring(
          polled_stations,
          length(polled_stations)-l,l);
        station(s).lp := sg;
        output POLL_EXT_poll(s);
        nextstate -;

      input POLL_EXT_rqu(sdu);
        output POLL_EXT_rqu(
          SDU((. 0,sg,sdu .)) );
        nextstate -;

    endstate normal;

endprocess type RM_AP2CL_AP;
```

Specification of the Client Role

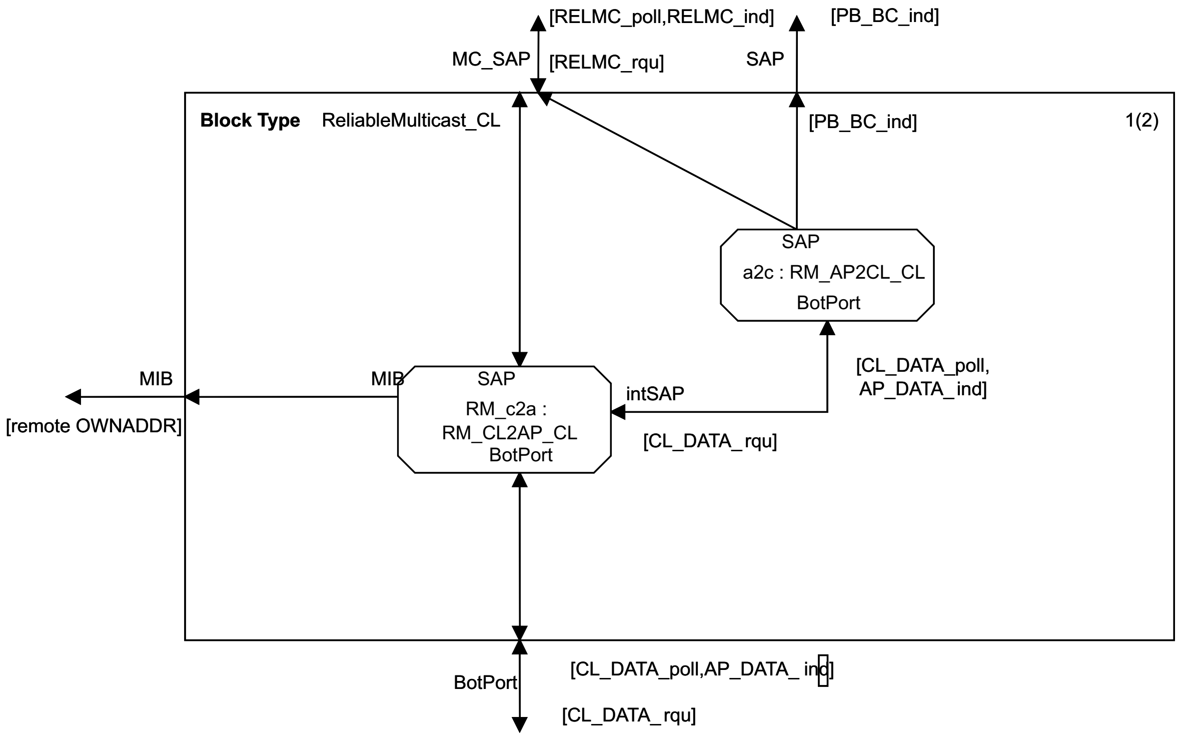


Figure A-3. Reliable multicast protocol, client role

```
virtual process type RM_CL2AP_CL;

  gate BotPort
  in with CL_DATA_poll, AP_DATA_ind;
  out with CL_DATA_rqu;

  gate SAP
  in with RELMC_rqu;
  out with RELMC_poll;

  gate intSAP
  in with CL_DATA_rqu;
  out with CL_DATA_poll, AP_DATA_ind;

  gate MIB
  out with remote OWNADDR;

  dcl /* state */
  curSDU SDU     := ''B,
  curSeq Natural := 0,
  curRes Natural := 0,
  lrts Natural   := 0,
  lr Natural     := 0,
  OA AddressType;

  dcl /* input vars */
  p RM_Poll_PDU,
  rqu RM_Rqu_PDU,
  sdu SDU,
  SA,DA AddressType;

  start;
    task OA := import(OWNADDR);
    nextstate normal;

  state normal;

    input RELMC_rqu(curRes, sdu);
      task
      curSDU := sdu,
      curSeq := curSeq + 1;
      lrts := lr + curRes;
      nextstate -;

    input CL_DATA_rqu(DA,sdu);
      task rqu := RM_Rqu_PDU(sdu),
        rqu.sl := curSeq, rqu.res :=
        curRes, rqu.sdu := curSDU;
      output CL_DATA_rqu(
        bcAddress,SDU(rqu));
      nextstate -;

    input CL_DATA_poll(sdu);
      task p := RM_Poll_PDU(sdu),
      lr := p.r;
      decision lr > lrts
        or curSDU = ''B;
      (true):
        task curSDU := ''B;
        output RELMC_poll;
      else:
      enddecision;
      output CL_DATA_poll(sdu);
      nextstate -;
    input AP_DATA_ind(SA,sdu);

      decision  SA = OA
        and  RM_Bc_PDU(sdu).sl =
          curSeq;
      (true): task curSDU := ''B;
      else:
      enddecision;
      output AP_DATA_ind(SA,sdu);
      nextstate -;

  endstate normal;

endprocess type RM_CL2AP_CL;



virtual process type RM_AP2CL_CL;

  gate SAP
  out with  RELMC_ind, PB_BC_ind;

  gate BotPort
  in with CL_DATA_poll, AP_DATA_ind;
  out with CL_DATA_rqu;


    dcl /* state */
    ack BooleanString := emptystring,
    tc TupleCache := emptystring,
    lr Natural := 0,
    lp Natural := 0;

    dcl /* inp. vars */
    dst,SA AddressType,
    res Natural,
    r Natural,
    sdu SDU;

    dcl /* tmp vars */
    p RM_Poll_PDU,
    pBC RM_Bc_PDU,
    ind Natural;

    start virtual;
        nextstate normal;

    state normal;
```

```
      input CL_DATA_poll(sdu);
        task
        p := RM_Poll_PDU(sdu),
        ack := mkstr(false,
          p.sg - lp - length(ack))
          // ack,
        ack := substring(ack,0,
           MAX_STATIONS),
        lp := p.sg;
        output CL_DATA_rqu(bcAddress,
          SDU( (. 0,0,ack,''B .) ));
        task ack := emptystring;
        nextstate -;

      input AP_DATA_ind(SA,sdu);
        task
        pBC := RM_Bc_PDU(sdu);
        ack := mkstring(true)
        // mkstr(false,
           pBC.sg - lp - length(ack))
        // ack;
        join ElemDupAndDeliver;

    endstate normal;

    connection ElemDupAndDeliver:
      task ind := search(tc,SA);
      decision ind = 0
        or pBC.sl > tc(ind).sl;
      (true):
        output RELMC_ind(SA,pBC.sdu);
        task tc :=
          if ind = 0
          then update(
            tc,ind,(. SA, pBC.sl .))
          else insert(
            tc,(. SA,pBC.sl .)) fi;
      else:
      enddecision;
      output PB_BC_ind(
        pBC.piggyLoad,pBC.sg);
      nextstate -;
    endconnection ElemDupAndDeliver;


endprocess type RM_AP2CL_CL;
```

A.6 Reliable Multicast – Dynamic Group Extensions

Services and Interfaces

```
/* ---------------------------------------------------------------------------

SERVICE PEND_GET

exports the set of pending multicasts when a new
station joins the group

SIGNALS

realized as remote variable PEND

  --------------------------------------------------------------------------- */

remote /* SERVICE PEND, AP only */
    PEND StationIdSet;


/* ---------------------------------------------------------------------------

SERVICE: NEW,EXCL, AP only
inherited from DynMedAcc

SERVICE: JOIN_START_ind, CL only
inherited from Polling

  --------------------------------------------------------------------------- */
```

Specification of the AP Role

```
redefined process type RM_CL2AP_AP;

  gate BotPort adding
  in with NEW_ind, EXCL_ind;
  out with NEW_rsp, EXCL_rsp;

  gate SAP adding
  in with NEW_rsp, EXCL_rsp;
  out with NEW_ind, EXCL_ind;

  state *;

    input NEW_ind(s);
      task
      station := station //
      if s > length(station)
      then
        mkstr(StationNull,
        s-length(station))
      else emptystring fi;
      output NEW_ind(s);
      nextstate -;

    input NEW_rsp;
      output NEW_rsp;
      nextstate -;

    input EXCL_ind(s);
      task
      'shorten "station" if necessary';
      output EXCL_ind(s);
      nextstate -;

    input EXCL_rsp;
      output EXCL_rsp;
      nextstate -;

  endstate;

endprocess type RM_CL2AP_AP;


redefined process type RM_AP2CL_AP;

  gate BotPort adding
  in with NEW_ind, EXCL_ind;
  out with NEW_rsp, EXCL_rsp;

  gate SAP adding
  in with NEW_rsp, EXCL_rsp,
    RELMC_STAT_rsp;
  out with  NEW_ind, EXCL_ind,
    remote PEND;

  dcl /* tmp var */
  accepted Boolean;
  dcl
  exported pend as PEND StationIdSet;

  state *;

    input NEW_ind(s);
      task
      g := incl(s,g),
      station := station //
      if s > length(station)
      then
        mkstr(StationNull,
        s-length(station))
      else emptystring fi;
      task
      'pend := set of all station Ids
      s for which station(s).wQ /=
      emptystring and
      first(station(s).wQ).res = OD';
      task export(pend);
      output NEW_ind(s);
      nextstate -;

    input NEW_rsp;
      output NEW_rsp;
      nextstate -;

    input EXCL_ind(s);
      task
      g := del(s,g);
      task 'shorten "station" if
      necessary';
      output EXCL_ind(s);
      nextstate -;

    input EXCL_rsp;
      output EXCL_rsp;
      nextstate -;

  endstate;

  state normal;

    input redefined AP_DATA_poll(s);
      decision g <= station(s).acked;
      (true):
        output RELMC_STAT_ind(s,acked);
        nextstate wait4STAT_cnf;
      (false):
        join not_acked;
      enddecision;

  endstate normal;

  state wait4STAT_cnf;

    input RELMC_STAT_rsp(accepted);
      task station(s).wQ :=
        if accepted
          then tail(station(s).wQ)
          else station(s).wQ fi;
      output PB_BC_poll(s,sg);
      nextstate normal;

  endstate wait4STAT_cnf;

endprocess type RM_AP2CL_AP;
```

Specification of the Client Role

```
redefined process type RM_CL2AP_CL;

  gate SAP adding
  out with JOIN_START_ind;

  gate BotPort adding
  in with JOIN_START_ind;

  state normal;

    input JOIN_START_ind;
      output JOIN_START_ind;
      nextstate normal;

  endstate normal;

endprocess type RM_CL2AP_CL;
```

```
redefined process type RM_AP2CL_CL;

    start redefined ;
        nextstate joining;

    state joining;

      input CL_DATA_poll(sdu);
        task
        p := RM_Poll_PDU(sdu),
        ack := emptystring,
        lr := p.r,
        lp := p.sg;
        output CL_DATA_rqu(bcAddress,
          SDU( (. 0,0,''B,''B .) ));
        nextstate normal;

      input AP_DATA_ind(SA,sdu);
        task pBC := RM_Bc_PDU(sdu);
        join ElemDupAndDeliver;
```

```
endstate joining;
```

```
endprocess type RM_AP2CL_CL;
```

A.7 Synchronous Channel

Services and Interfaces

```
/* ----------------------------------------------------------------------------

SERVICE: DEC, AP initiated, polled service

Transmission of decisions from the AP to the clients.

SIGNALS

DEC_poll(s): Polls for decision on behalf of 's'.
DEC_rqu(d): Request transmission of 'd'
DEC_ind(d,i): Indicates reception of a decision 'd'. 'i' is the ID of the station
which 'd' relates to.

 ----------------------------------------------------------------------------*/

value type DecType;
  literals accept,reject,nodec, excl, newcl;
endvalue type;

signal /* SERVICE DEC */
  DEC_poll(StationId),
  DEC_rqu(DecType),
  DEC_ind(DecType,StationId);


/* ----------------------------------------------------------------------------

SERVICE: FRAME_OWNER

Delivers the member ID of the owner of the last frame received.

FRAME_OWNER_ind(m): 'm' is the member ID of the frame's owner

 ----------------------------------------------------------------------------*/

signal /* SERVICE FRAME_OWNER */
  FRAME_OWNER_ind(StationId);


/* ----------------------------------------------------------------------------

   INTERNAL SIGNALS

   --------------------------------------------------------------------------*/

signal /* SERVICE DEC_int */
  DEC_int_ind(DecType,Natural);

signal /* SERVICE PB_BC_int */
  PB_BC_int_rqu(Natural, Natural, SDU),
  PB_BC_int_ind(Natural, Natural, SDU, Natural);
```

PDUs

```
value type DecString inherits MyString<DecType>;
```

```
endvalue type;

value type PDUTypes;
  literals init,normal;
endvalue type;

value type SC_normalPDU;
struct
  ptype PDUTypes;
  synchCh DecString;
  piggyLoad SDU;
endvalue type;
```

Specification of the AP Role

```
process type SC_AP;

  /* signature */

  gate SAP
  in with PB_BC_rqu,DEC_rqu;
  out with PB_BC_poll,DEC_poll;

  gate SAP_int
  in with PB_BC_int_rqu;

  gate BotPort
  in with PB_BC_poll;
  out with PB_BC_rqu;

  /* state */

  dcl
  sy DecString := mkstr(nodec,OD+1);

  /* tmp vars */
  dcl d DecType,
  sdu SDU,
  mid Natural,
  gs Natural;

  dcl /* state */
  s StationId,
  sg Natural;

  start;
    nextstate normal;

  state normal;

    input PB_BC_poll(s,sg);
      output DEC_poll;
      nextstate normal;

    input DEC_rqu(d);
      task sy := sub
       string(sy,1,length(sy)-1),
      sy := mkstring(d) // sy;
      output PB_BC_poll(s,sg);
      nextstate normal;

    input PB_BC_rqu(sdu);
      output PB_BC_rqu(
       SDU((. normal,sy,sdu .)) );
      nextstate normal;

    input PB_BC_int_rqu(gs,mid,sdu);
      decision gs;
      (0): output PB_BC_rqu(
        SDU((. normal,sy,sdu .)) );
      else: output PB_BC_rqu(
        SDU((. init,sy,gs,mid,sdu .)));
      enddecision;
      nextstate normal;

  endstate normal;

endprocess type SC_AP;
```

Specification of the Client Role

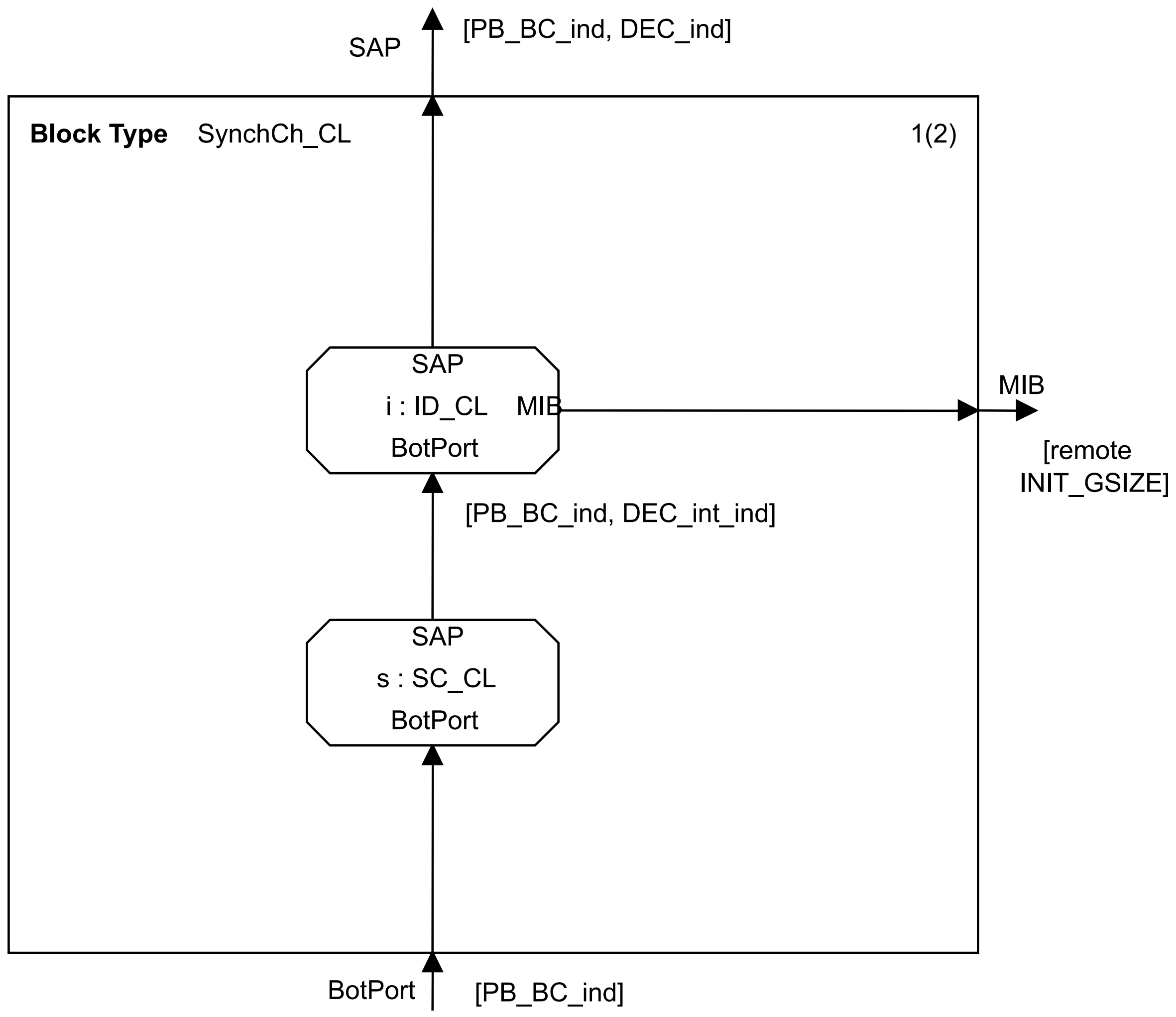


Figure A-4. Synchronous channel, client role.

```
virtual process type SC_CL;

  /* signature */

  gate SAP
  out with PB_BC_ind, DEC_int_ind;

  gate BotPort
  in with PB_BC_ind;

  /* state */
  dcl
  last_sg Natural := 0;

  /* tmp vars */
  dcl
  p SC_normalPDU,
  sg Natural,
  sdu SDU;

  start virtual;
    nextstate normal;

  state normal;

    input virtual PB_BC_ind(sdu,sg);
      task p := SC_normalPDU(sdu);
      task '{
      for(dcl i:=sg-last_sg,
        i>0,i := i-1)
        output(DEC_ind(p.synchCh[i],
        sg-i+1);
      }';
      task last_sg := sg;
      output PB_BC_ind(p.piggyLoad,sg);
      nextstate -;

  endstate normal;

endprocess type SC_CL;
```

```
virtual process type ID_CL;

  /* signature */

  gate SAP
  out with PB_BC_ind, DEC_ind,
    FRAME_OWNER_ind;

  gate BotPort
  in with PB_BC_ind, DEC_int_ind;

  gate MIB
  out with remote INIT_GSIZE;

  /* state */
  dcl
  cur,cur_old Natural := 0,
  gs Natural := 0;

  /* tmp vars */
  dcl
  d DecType,
  sg Natural,
  sdu SDU;

  start virtual;
    task
    gs := import(INIT_GSIZE),
    cur := 0;
    nextstate normal;

  state normal;

    input virtual DEC_int_ind(d,sg);
       output DEC_ind(d,cur);
       task
       cur_old := cur,
       cur := (cur + 1) mod gs;
       nextstate -;

    input PB_BC_ind(sdu,sg);
      output FRAME_OWNER_ind(cur_old);
      output PB_BC_ind(sdu,sg);
      nextstate -;

  endstate normal;

endprocess type ID_CL;
```

A.8 Synchronous Channel – Dynamic Group Extensions

Services and Interfaces

```
/* ----------------------------------------------------------------------------

SERVICE STATES

 ----------------------------------------------------------------------------*/

value type ServiceState;
  literals joined, stopped;
endvalue type;

remote SynchChState ServiceState;


/* ----------------------------------------------------------------------------

SERVICE: NEW_CL, Client only

New notification at the client

SIGNALS

NEW_CL_ind(s): Indicates that the AP has added a new RGCP
station to the polling list. 's' is the ID of that station

 ----------------------------------------------------------------------------*/

signal /* SERVICE NEW_CL */
  NEW_CL_ind(StationId);


/* ----------------------------------------------------------------------------

SERVICE: EXCL_CL, Client only

Exclude notification at the client

SIGNALS

EXCL_CL_ind(s): Indicates that the AP has removed 's'

 ----------------------------------------------------------------------------*/

signal /* SERVICE EXCL_CL */
  EXCL_CL_ind(StationId);


/* ----------------------------------------------------------------------------

SERVICE: FAIL, Client only

Indicate protocol stopped

SIGNALS

FAIL_ind

 ----------------------------------------------------------------------------*/
```

```
signal /* SERVICE FAIL */
  FAIL_ind;


/* -----------------------------------------------------------------------------

SERVICE: JOIN_START, inherited

 ------------------------------------------------------------------------------*/

signal /* SERVICE JOIN_START */
  JOIN_START_ind;
```

PDUs

```
value type SC_initPDU;
struct
  ptype PDUTypes;
  synchCh DecString;
  gsize Natural;
  mid   Natural;
  piggyLoad SDU;
endvalue type SC_initPDU;
```

Specification of the AP Role

```
process type SC_Not_AP;

  /* signature */

  gate SAP
  in with DEC_rqu, EXCL_rsp, NEW_rsp,
    PB_BC_rqu;
  out with DEC_poll, EXCL_ind, NEW_ind,
    PB_BC_poll;

  gate BotPort
  in with DEC_poll, EXCL_ind, NEW_ind,
    PB_BC_poll;
  out with DEC_rqu, EXCL_rsp, NEW_rsp,
    PB_BC_int_rqu;

  gate MIB
  out with remote INIT_G;

  /* types */
  value type Station;
    struct
    sDec DecType;
  endvalue type Station;
  synonym StationNull Station =
   (. nodec .);

  value type StationList
    inherits MyString<Station>;
  endvalue type StationList;

  /* state */
  dcl
  station StationList,
  trycnt Natural := 0,
  mid Natural := 0,
  gs Natural;

  /* tmp vars */
  dcl
  d DecType,
  sg Natural,
  s StationId,
  sdu SDU;


  start;
    task
    gs := setsize(import(INIT_G)),
    station := mkstr(StationNull,gs);
    nextstate normal;

  state normal;

    input EXCL_ind(s);
      task station(s).sDec := excl;
      output EXCL_ind(s);
      nextstate normal;

    input EXCL_rsp;
      output EXCL_rsp;
      nextstate normal;

    input NEW_ind(s);
      task
      station := station //
        if s > length(station)
          then mkstr(StationNull,s-
length(station))
          else emptystring fi,
      station(s).sDec := newcl,
      trycnt := OD+1;
      output NEW_ind(s);
      nextstate normal;

    input NEW_rsp;
      output NEW_rsp;
      nextstate normal;

    input DEC_rqu(d);
      output DEC_rqu(d);
      nextstate normal;

    input DEC_poll(s);
      decision station(s).sDec;
      (excl):
        output DEC_rqu(excl);
        task
        station(s) :=  StationNull,
        gs := gs - 1;
        task 'shrink station list';
      (newcl):
        task gs := gs + 1,
        mid := mid + 1;
        station(s).sDec := nodec;
        output DEC_rqu(newcl);
      else:
        task mid := mid + 1;
        output DEC_poll(s);
      enddecision;
      task mid := mid mod gs;
      nextstate normal;

    input PB_BC_poll(s,sg);
      output PB_BC_poll(s,sg);
      nextstate normal;

    input PB_BC_rqu(sdu);
      decision trycnt;
      (>0):
        output PB_BC_int_rqu(
          gs,mid,sdu);
        task trycnt := trycnt - 1;
      else:
        output PB_BC_int_rqu(
          0,mid,sdu);
      enddecision;
      nextstate normal;

  endstate normal;

endprocess type SC_Not_AP;
```

Specification of the Client Role

```
redefined process type SC_CL;

  gate SAP adding
  out with FAIL_ind, JOIN_START_ind;

  gate BotPort adding
  in with JOIN_START_ind;

  gate SAP_int
  out with PB_BC_int_ind;

  /* tmp vars */
  dcl
  pi SC_initPDU;

  /* state */
  timer toSynchCh :=
    DeltaSynchCh*(1+rho);

  start redefined;
    nextstate stopped;

  state stopped;

    input JOIN_START_ind;
      output JOIN_START_ind;
      nextstate joining;

  endstate stopped;

  state joining;

    input PB_BC_ind(sdu,sg);
      task last_sg := sg;
      join forward_PB;

  endstate joining;

  state normal;

    input redefined PB_BC_ind(sdu,sg);
      set(toSynchCh);
      task p := SC_normalPDU(sdu);
      decision sg - last_sg;
      (> OD+1):
        output FAIL_ind;
        nextstate stopped;
      else:
      enddecision;
      task '{
      for(dcl i:=sg-last_sg-1,i >= 0,
         i := i-1)
      {
        output(DEC_ind(p.synchCh[i],
          sg-i);
      } }';
      task last_sg := sg;
      join forward_PB;

    input toSynchCh;
      output FAIL_ind;
      nextstate stopped;

  endstate normal;

  connection forward_PB:
    decision p.ptype;
    (normal):
      output PB_BC_ind(p.piggyLoad,sg);
    (init):
      task pi := SC_initPDU(sdu);
      output PB_BC_int_ind(
        pi.gsize,pi.mid,
        pi.piggyLoad,sg);
    enddecision;
    set(toSynchCh);
    nextstate normal;
  endconnection forward_PB;

endprocess type SC_CL;



redefined process type ID_CL;

  gate SAP adding
  in with remote SynchChState;
  out with EXCL_CL_ind, NEW_CL_ind,
    FAIL_ind, JOIN_START_ind;

  gate BotPort adding
  in with PB_BC_int_ind,
    FAIL_ind, JOIN_START_ind;

  /* state */
  dcl
  exported SynchChState
    ServiceState := stopped;

  start redefined;
    task cur := 0;
    export(SynchChState);
    nextstate stopped;

  state stopped;

    input JOIN_START_ind;
      output JOIN_START_ind;
      nextstate joining;

  endstate;

  state joining;

    input PB_BC_int_ind(gs,cur,sdu,sg);
      output FRAME_OWNER_ind(cur);
      output PB_BC_ind(sdu,sg);
      task SynchChState := joined;
      export(SynchChState);
      nextstate normal;

    input FAIL_ind;
      output FAIL_ind;
      task SynchChState := stopped;
      export(SynchChState);
      nextstate stopped;

  endstate joining;

  state normal;

    input redefined DEC_int_ind(d,sg);
      task cur_old := cur;
      decision d;
      (excl):
        output EXCL_CL_ind(cur);
        task
        gs := gs -1;
        cur := cur mod gs;
      (newcl):
        output NEW_CL_ind(cur);
        task
        gs := gs + 1,
        cur := (cur + 1) mod gs;
      else:
        output DEC_ind(d,cur);
        task cur := (cur + 1) mod gs;
      enddecision;
      nextstate -;

    input FAIL_ind;
      output FAIL_ind;
      task SynchChState := stopped;
      export(SynchChState);
      nextstate stopped;

  endstate normal;

endprocess type ID_CL;
```

A.9 Atomic Multicast

Services and Interfaces

```
/* --------------------------------------------------------------------------

SERVICE: AMC, inherits RELMC, Client Only

Transmission of atomic multicasts.

SIGNALS

AMC_poll, see RELMC_poll
AMC_rqu, see RELMC_rqu
AMC_ind, see RELMC_ind, adds 'sid'.

 --------------------------------------------------------------------------*/

signal /* SERVICE AMC, client only */
  AMC_poll inherits RELMC_poll,
  AMC_rqu  inherits RELMC_rqu,
  AMC_ind inherits RELMC_ind adding (StationId);


/* --------------------------------------------------------------------------

SERVICE: NEW, from DynMedAcc
SERVICE: EXCL, from DynMedAcc

 --------------------------------------------------------------------------*/
```

Specification of the AP Role

```
virtual process type AMC_AP;

  /* signature */

  gate SAP
  in with DEC_rqu;
  out with DEC_poll;

  gate BotPort
  in with DEC_poll,RELMC_STAT_ind;
  out with DEC_rqu;

  /* state */
  dcl
  d DecType := nodec;

  /* tmp vars */
  dcl
  dt DecType,
  st StatusEnum,
  s StationId;
```

```
start virtual;
  nextstate normal;

state normal;

  input virtual
    RELMC_STAT_ind(s,st);
    decision st;
    (acked, ODretries):
      task d := accept;
    (resretries):
      task d := reject;
    enddecision;
    nextstate -;

  input DEC_poll;
    output DEC_poll;
    nextstate -;

  input DEC_rqu(dt);
    output DEC_rqu(d);
    task d := nodec;
    nextstate -;
  endstate normal;

endprocess type AMC_AP;
```

Specification of the Client Role

```
virtual process type AMC_CL;

  /* signature */

  gate SAP
  out with DEC_ind,
    AMC_ind, AMC_poll;
  in with AMC_rqu;

  gate BotPort
  in with  DEC_ind, FRAME_OWNER_ind,
    RELMC_ind, RELMC_poll;
  out with RELMC_rqu;

  gate MIB
  out with remote INIT_GSIZE;

  /* types */
  value type Station;
    struct
    sdu SDU;
    src AddressType;
  endvalue type Station;
  synonym StationNull Station =
    (. ''B, ''B .);
  value type StationList
    inherits MyString<Station>;
  endvalue type;

  dcl /* state */
  cSDU SDU := ''B,
  cSrc AddressType := ''B,
  station StationList := emptystring;


  /* tmp vars */
  dcl
  d DecType,
  src AddressType,
  sdu SDU,
  r Natural,
  mid StationId;

  start virtual;
    task station := mkstr(StationNull,
      import(INIT_GSIZE));
    nextstate idle;

  state idle;

    input RELMC_ind(src,sdu);
      task
      cSDU := sdu,
      cSrc := src;
      nextstate haveRM;

  endstate idle;

  state haveRM;

    input FRAME_OWNER_ind(mid);
      task
      station(mid).sdu := cSDU,
      station(mid).src := cSrc;
      nextstate idle;

  endstate haveRM;

  state idle,haveRM;

    input RELMC_poll;
      output AMC_poll;
      nextstate -;

    input AMC_rqu(r, sdu);
      output RELMC_rqu(r, sdu);
      nextstate -;

    input virtual DEC_ind(d,mid);
      decision d;
      (accept):
        output AMC_ind(
          station(mid).src,
          station(mid).sdu,mid);
        task station(mid) :=
          StationNull;
      (reject):
        task station(mid) :=
          StationNull;
      else:
        output DEC_ind(d,mid);
      enddecision;
      nextstate -;

  endstate;

endprocess type AMC_CL;
```

A.10 Atomic Multicast – Dynamic Group Extensions

Services and Interfaces

```
/* ---------------------------------------------------------------

SERVICE STATES

 ---------------------------------------------------------------*/

remote AtomMCState ServiceState;


/* ---------------------------------------------------------------

SERVICE: ATOMMC_STAT

SIGNALS

ATOMMC_STAT_ind(s, st): indicates that an atomic multicast of station with ID
's' was accepted ('st' = false) or rejected ('st' = true).

 ---------------------------------------------------------------*/

signal /* SERVICE ATOMMC_STAT */
  ATOMMC_STAT_ind(StationId, Boolean);


/* ---------------------------------------------------------------

SERVICE: NEW, from DynMedAcc
SERVICE: EXCL, from DynMedAcc
SERVICE: FAIL_ind, from SynchCh
SERVICE: JOIN_START_ind, from SynchCh

 --------------------------------------------------------------- */
```

Specification of the AP Role

```
redefined process type AMC_AP;

  gate SAP adding
  out with EXCL_ind,
    NEW_ind, ATOMMC_STAT_ind;
  in with EXCL_rsp, NEW_rsp;

  gate BotPort adding
  in with NEW_ind, EXCL_ind,ACKS_ind;
  out with NEW_rsp,
    EXCL_rsp, RELMC_STAT_rsp,
    remote PEND;


  gate MIB
  out with remote INIT_G;
```

```
/* state */
value type Station;
  struct
  pend StationIdSet;
endvalue type Station;
synonym StationNull Station =
  (. empty .);

value type StationList
  inherits MyString<Station>;
endvalue type StationList;

dcl
station StationList;

/* tmp vars */
dcl
```

```
  a StationIdSet;

  start redefined;
    task
    station := mkstr(StationNull,
      setsize(import(INIT_G)));
    nextstate normal;

  state normal;

    input redefined
      RELMC_STAT_ind(s,st);
      decision st;
      (acked):
        decision station(s).pend;
        (empty):
          task d := accept;
          output RELMC_STAT_rsp(true);
        else:
          task d := nodec;
          output RELMC_STAT_rsp(
            false);
        enddecision;
      (ODretries):  task d := accept;
      (resretries): task d := reject;
      enddecision;
      decision d;
      (accept):
        task '{
        for(dcl i := 0;
        i < length(station); i := i+1)
          station(i).pend :=
          del(s,station(i).pend); }';
        output ATOMMC_STAT_ind(
          s,true);
      (reject):
        output ATOMMC_STAT_ind(
          s,false);
      else:
      enddecision;
      nextstate -;

    input ACKS_ind(s,a);
      task station(s).pend :=
        station(s).pend  - a ;
      nextstate normal;

  endstate normal;

  state *;

    input NEW_ind(s);
      task
      station := station //
      if s > length(station)
      then mkstr(StationNull,
        s-length(station))
      else emptystring fi,
      station(s).pend := import(PEND);
      output NEW_ind(s);
      nextstate -;

    input EXCL_ind(s);
      task station(s) := StationNull;
      task 'shorten "station" if
        necessary';
      output EXCL_ind(s);
      nextstate -;

    input NEW_rsp;
      output NEW_rsp;
      nextstate -;

    input EXCL_rsp;
      output EXCL_rsp;
      nextstate -;

  endstate;

endprocess type AMC_AP;
```

Specification of the Client Role

```
redefined process type AMC_CL;

  gate SAP adding
  out with EXCL_CL_ind, NEW_CL_ind,
    JOIN_START_ind, FAIL_ind,
    remote AtomMCState;

  gate BotPort adding
  in with EXCL_CL_ind, NEW_CL_ind,
    JOIN_START_ind, FAIL_ind;

  gate MIB adding
  out with remote OWNADDR;

  dcl /* tmp vars */
  s StationId,
  OA AddressType;

  /* state */
  dcl
  exported AtomMCState ServiceState
    := stopped;

  start redefined;
    task
    station := mkstr(StationNull,
      import(INIT_GSIZE)),
    OA := import(OWNADDR);
    export(AtomMCState);
    nextstate stopped;

  state stopped;

    input JOIN_START_ind;
      task station := mkstr(
       StationNull,
       import(INIT_GSIZE));
      output JOIN_START_ind;
      nextstate join_idle;
```

```
endstate stopped;

state join_idle;

  input RELMC_ind(src,sdu);
    task
    cSDU := sdu,
    cSrc := src;
    nextstate join_haveRM;

endstate join_idle;

state join_haveRM;

  input FRAME_OWNER_ind(mid);
    task
    station(mid).sdu := cSDU,
    station(mid).src := cSrc;
    nextstate join_idle;

endstate join_haveRM;

state join_idle,join_haveRM;

  input RELMC_poll;
    output AMC_poll;
    nextstate -;

  input AMC_rqu(r, sdu);
    output RELMC_rqu(OD, sdu);
    nextstate -;

  input DEC_ind(d,mid);
    decision d = accept
      and station(mid).src = OA;
    (true):
      output AMC_ind(
        station(mid).src,
        station(mid).sdu,mid);
      task
      station(mid) :=
        StationNull,
      AtomMCState := joined;
      export(AtomMCState);
      decision state;
      (join_idle):
        nextstate idle;
      (join_haveRM):
        nextstate haveRM;
      enddecision;
    (false):
      nextstate -;
    enddecision;

endstate;

state idle,haveRM;

  input redefined DEC_ind(d,mid);
    decision d;
    (accept):
      decision station(mid).sdu;
      (''B):
        task
        AtomMCState := stopped;
        export(AtomMCState);
        output FAIL_ind;
        nextstate stopped;
      else:
        output AMC_ind(
        station(mid).src,
        station(mid).sdu,mid);
        task
        station(mid)
          := StationNull;
      enddecision;
    (reject):
      task
      station(mid) := StationNull;
    else:
      output DEC_ind(d,mid);
    enddecision;
    nextstate -;

endstate;

state *;

  input FAIL_ind;
    task AtomMCState := stopped;
    export(AtomMCState);
    output FAIL_ind;
    nextstate stopped;

  input EXCL_CL_ind(s);
    task station(s) :=
    StationNull;
    task 'shrink station';
    output EXCL_CL_ind(s);
    nextstate -;

  input NEW_CL_ind(s);
    task station := station //
    if s > length(station)
    then mkstr(StationNull,
      s-length(station))
    else emptystring fi;
    output NEW_CL_ind;
    nextstate -;

endstate;

endprocess type AMC_CL;
```

A.11 Membership

Services and Interfaces

```
/* ---------------------------------------------------------------------------

SERVICE STATES

  ---------------------------------------------------------------------------*/

remote MemState ServiceState;


/* ---------------------------------------------------------------------------

SERVICE: JOIN_MEM

allows transmission of a atomic multicast with resiliency OD while joining

SIGNALS

JOIN_MEM_poll: inherits AMC_poll, poll for SDU.

JOIN_MEM_rqu(sdu): request
transmission of atomic multicast sdu. Resiliency is always OD.

 ---------------------------------------------------------------------------*/

signal /* SERVICE: JOIN_MEM */
  JOIN_MEM_poll inherits AMC_poll,
  JOIN_MEM_rqu(SDU);


/* ---------------------------------------------------------------------------

SERVICE: MEM

Indicate membership changes

SIGNALS

MEM_ind(m): indicate that membership has changed and new membership is m.

 ---------------------------------------------------------------------------*/

signal /* SERVICE: MEM */
  MEM_ind(AddressSet);

/* ---------------------------------------------------------------------------

SERVICE: FAIL_ind, from AtomMul

  ---------------------------------------------------------------------------*/
```

PDUs

```
value type StationState;
  literals joining, normal;
endvalue type;
```

```
value type MemElemType;
  struct
  addr AddressType;
  mstate StationState;
endvalue type;

value type MemArray inherits MyArray<StationId,MemElemType>;
  operators
    memset: this MemArray -> AddressSet;
endvalue type;

value type Mem_PDU;
  struct
  m MemArray;
endvalue type Mem_PDU;
```

Specification of the AP Role

```
process type Mem_AP;

  gate BotPort
  in with NEW_ind, EXCL_ind,
PB_BC_poll, ATOMMC_STAT_ind;
  out with NEW_rsp, EXCL_rsp,
PB_BC_rqu;

  gate MIB
  out with remote IDADDR, remote STACK-
PID, remote PL;

  dcl /* tmp var */
  pl PollingList,
  id2addr AddressArray,
  s StationId,
  p Mem_PDU,
  st Boolean;

  /* state */

  /* required type of class
    'Station' */

  value type Station;
    struct
    sState StationState;
  endvalue type Station;
  synonym StationNull Station =
   (. normal .);
  value type StationList
    inherits MyString<Station>;
  endvalue type StationList;

  dcl
  m AddressArray,
  station StationList :=
    mkstr(StationNull,MAX_STATIONS);


  /* Transistions */

  start;
    nextstate normal;

state normal;

  input NEW_ind(s);
    task
    station(s).sState := joining;
    output NEW_rsp;
    nextstate normal;

  input PB_BC_poll(s);
    decision station(s).sState;
    (joining):
      task pl := import(PL);
      task
      'for(dcl i := 0,j := 0,pos :=
         0; i < length(pl.list);
         i := i +1)
       {
         pos := pl.cur + i mod
          length(pl.list);
         pe := pl.list(pos);
         if (pe.peStack =
           import(STACKPID(rgcp)))
         {
          p.m(j).addr :=
          import(IDADDR(pe.peOwner));
          p.m(j).mstate :=
          stations.sState(
            pe.peOwner);
          j := j+1;
         }
       }';
       output PB_BC_rqu(SDU(p));
    (normal):
      output PB_BC_rqu(''B);
    enddecision;
    nextstate normal;

  input ATOMMC_STAT_ind(s,st);
    decision st;
    (true):
       task
       station(s).sState := normal;
    (false):
    enddecision;
    nextstate normal;
```

```
endprocess type Mem_AP;
```

Specification of the Client Role

```
process type Mem_CL;

  gate SAP
  in with JOIN_MEM_rqu, AMC_rqu,
    remote MemState;
  out with JOIN_MEM_poll, AMC_poll,
    AMC_ind, MEM_ind, FAIL_ind;

  gate BotPort
  in with AMC_poll, AMC_ind,
    PB_BC_ind,
    EXCL_CL_ind,NEW_CL_ind,
    JOIN_START_ind, FAIL_ind;
  out with AMC_rqu;

  gate MIB
  out with remote OWNADDR;

  /* state */
  dcl
  exported MemState ServiceState,
  m MemArray,
  s StationId;

  /* tmp vars */
  dcl
  p Mem_PDU,
  src,OA AddressType,
  r Natural,
  sdu SDU;

  start;
    task OA := import(OWNADDR),
    MemState := stopped;
    export(MemState);
    nextstate Stopped;

  state Stopped;

    input JOIN_START_ind;
      nextstate Joining1;

  endstate Stopped;

  state Joining1, Joining2;

    input PB_BC_ind(sdu);
      decision sdu;
      (''B):
        nextstate -;
      else:
        task
        p := Mem_PDU(sdu),
        m := p.m;
        nextstate Joining2;
      enddecision;

      input AMC_poll;
        output JOIN_MEM_poll;
        nextstate -;

      input JOIN_MEM_rqu(sdu);
        output AMC_rqu(OD,sdu);
        nextstate -;

  endstate;

  state Joining2;

    input EXCL_CL_ind(s);
      task m := del(m,s);
      nextstate -;

    input NEW_CL_ind(s);
      task m := ins(m,s,
        (. ''B,joining .));
      nextstate -;

    input AMC_ind(src,sdu,s);
      decision m(s).mstate;
      (normal):
      (joining):
        task m(s) := (. src,normal .);
      enddecision;

      decision src = OA;
      (true):
        task MemState := joined;
        export(MemState);
        output MEM_ind(memset(m));
        output AMC_ind(src,sdu);
        nextstate normal;
      (false):
        nextstate -;
      enddecision;

  endstate Joining2;


  state normal;

    input NEW_CL_ind(s);
      task m := ins(m,s,(. ''B,joining
      .));
      nextstate -;

    input AMC_ind(src,sdu,s);
      decision m(s).mstate = joining;
      (true):
      (false):
        task m(s) := (. src,normal .);
      enddecision;

      output MEM_ind(memset(m));
      output AMC_ind(src,sdu);
      nextstate normal;

    input AMC_poll;
      output AMC_poll;
      nextstate normal;
    input AMC_rqu(r,sdu);
      output AMC_rqu(r,sdu);
      nextstate normal;

    input EXCL_CL_ind(s);
      task m := del(m,s);
      output MEM_ind(memset(m));
      nextstate -;

  endstate normal;

  state *;

    input FAIL_ind;
      task MemState := stopped;
      export(MemState);
      nextstate Stopped;

  endstate;

endprocess type Mem_CL;
```