Preliminary Specification of Basic Services and Protocols

G. Blair, C. Brudna, V. Cahill, A. Casimiro,
R. Cunningham, H. Duran-Limon, J. Kaiser,
P. Martins and P. Veríssimo

DI-FCUL TR–03–18

July 2003

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal

Project IST-2000-26031

CO-operating Real-time senTient objects: architecture and EXperimental evaluation

Preliminary Specification of Basic Services and Protocols

CORTEX Deliverable D5

Version 1.0

February 12, 2003

Revisions

Rev.	Date	Comment
0.1	17/01/2003	Draft document for internal review
0.2	12/02/2003	Final draft for comments
1.0	27/02/2003	Final document

Editor

António Casimiro, University of Lisboa

Contributors

Cristiano Brudna, University of Ulm
Gordon Blair, University of Lancaster
Vinny Cahill, Trinity College Dublin
António Casimiro, University of Lisboa
Raymond Cunningham, Trinity College Dublin
Hector Duran-Limon, University of Lancaster
Jörg Kaiser, University of Ulm
Pedro Martins, University of Lisboa
Paulo Veríssimo, University of Lisboa

Address

Faculdade de Ciências da Universidade de Lisboa
Bloco C5, Campo Grande
1749-016 Lisboa
Portugal

Contents

1 Introduction

Given the objective of CORTEX to explore the fundamental theoretical and engineering issues that will make possible the construction of large-scale proactive applications composed of sentient objects, a preliminary CORTEX system architecture has been proposed in a previous deliverable [17]. The proposed architecture explicitly considers two logical scopes in terms of the problems that must be dealt. In one hand, it considers a local scope, which includes all the issues that respect the constitution and operation of CORTEX nodes or collections of nodes, namely interfaces to application objects and supporting/run-time services. On the other hand, a global scope, which addresses the need to accommodate the heterogeneity of environments in which applications will operate, by proposing solutions that follow a WAN-of-CANs approach and allow de definition of QoS containment zones and the hierarchical composition of such zones.

In order to materialize this architecture, it is necessary to define a set of services to support the envisaged applications, which will architecturally exist, and will be accessed, locally to a CORTEX node, but which may have a distributed, global scope. In deliverable WP3-D4 we have already presented a preliminary view of the local architecture of a node, where it was possible to identify the above-mentioned supporting services. The present deliverable somehow continues the work that has been presented in previous deliverables, in particular in deliverables WP2-D3 and WP3-D4.

The fundamental objective of this deliverable is to present a preliminary specification of the basic services and protocols that should be provided in CORTEX in order to support the envisaged applications. Although it is still necessary to do some work with respect to the integration of some of the services presented in this deliverable, we believe that the overall view of the required services and protocols, which is presented here, constitutes a fundamental step towards that integration.

In terms of the structure of the deliverable, we try to follow a top-down approach by firstly motivating, in Section 2, the need for some services and by describing and specifying what they should provide (i.e., their properties). However, this approach is not always followed, given that in Sections 4.5 and 4.4 we provide two papers as originally presented, therefore including in those sections all the definitions of required properties and proposed interfaces.

Then, in Section 3, we focus on the services from the perspective of the interfaces they should provide. We believe this makes sense because interfaces can be specified independently of some particular implementation.

The last section focuses on the specific protocols and services to be provided in CORTEX. They implement (at least some of) the interfaces described in Section 3. However, since this is a preliminary deliverable, only on a subset of the services and protocols is presented.

2 Overview of basic services and protocols

In this section we provide an overview of the basic services and protocols that we propose for CORTEX. This set of services can be viewed as a middleware layer that exists below CORTEX applications which provides the adequate abstractions and implements the necessary functionalities required to address the needs of this class of applications.

In order to clarify the discussion, and given that it is possible to identify certain groups of services in terms of their purpose, their nature or their location in a service stack, we propose to organize the services and protocols provided in CORTEX in three main groups, as depicted in Figure 1.

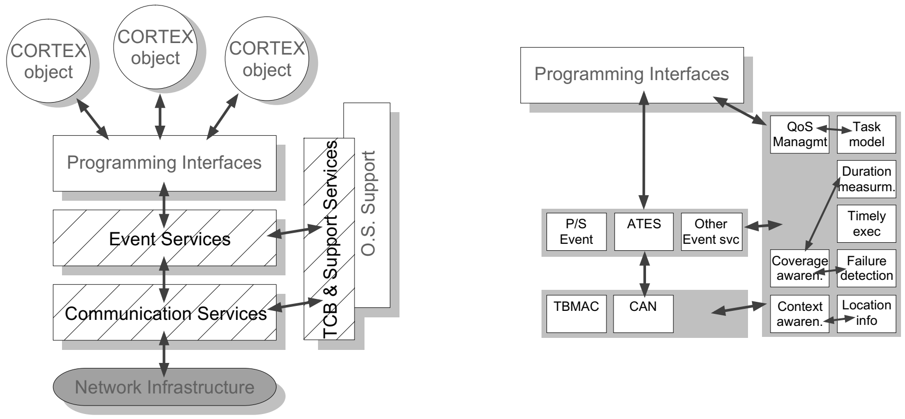


Figure 1: Block diagram of CORTEX basic services.

At a lower level of the architecture we find a group of basic communication services and protocols, which are implemented directly on top of the network infrastructure and enforce abstract network properties such as those related with the provision of guaranteed communication latency or reliability. More specifically, at the right hand side of the figure it is possible to observe that we specifically focus on the CAN controller area network (to provide predictable communication at the CAN level of the WAN-of-CAN structure of CORTEX), and on TBMAC, a protocol specifically designed to address predictability requirements over wireless communication infrastructures.

Above the communication level it is possible to consider that a set of event related services exists. The most relevant service at this level implements the required anonymous communication, based on the publish-subscribe paradigm. Several issues, such as discovery/announcement of events, filtering, and routing of events to interested subscribers, is addressed at this level. Another important aspect is related with the need to address non-functional requirements of applications, namely those concerning timeliness. One approach is provided with the ATES service, which aims at providing the means to integrate timeliness requirements in event-based communication. This is also done with the help of some supporting services, those that constitute the third group here considered.

The Timely Computing Base (TCB) can be seen as an oracle that provides a few very basic but fundamental services to the rest of the system, and is therefore orthogonal to the architecture. Since there exist other support services besides the

TCB, they must be all included in the same group of services, as represented in Figure 1. These other support services, which include QoS management, coverage awareness and context awareness, are also generic services that may be useful as building blocks for implementing the rest of the system (both other middleware services and the applications).

The rest of this section will focus on all these services, providing a motivation for their need and, whenever possible, stating the properties which more formally characterize them.

2.1 TCB services

As described in deliverable WP3-D4 [17], the Timely Computing Base (TCB) can be seen as a special architectural component serving the whole system and providing crucial time related services. In this section we will review the basic TCB properties, those that must be exhibited by a TCB component, which will serve to understand the specification of the services to be provided at the TCB API. This API, which was already presented in deliverable WP3-D4, will be summarized in Section 3.1. The details relative to the internal definition and implementation of TCB services and protocols are presented in Section 4.1.

Figure 2 illustrates the modular composition of a TCB component. The interaction (locally to a site) between payload applications and the TCB component is made through a TCB API. However, the TCB can be internally composed of several modules, which implement the necessary protocols and (internal and external) services.

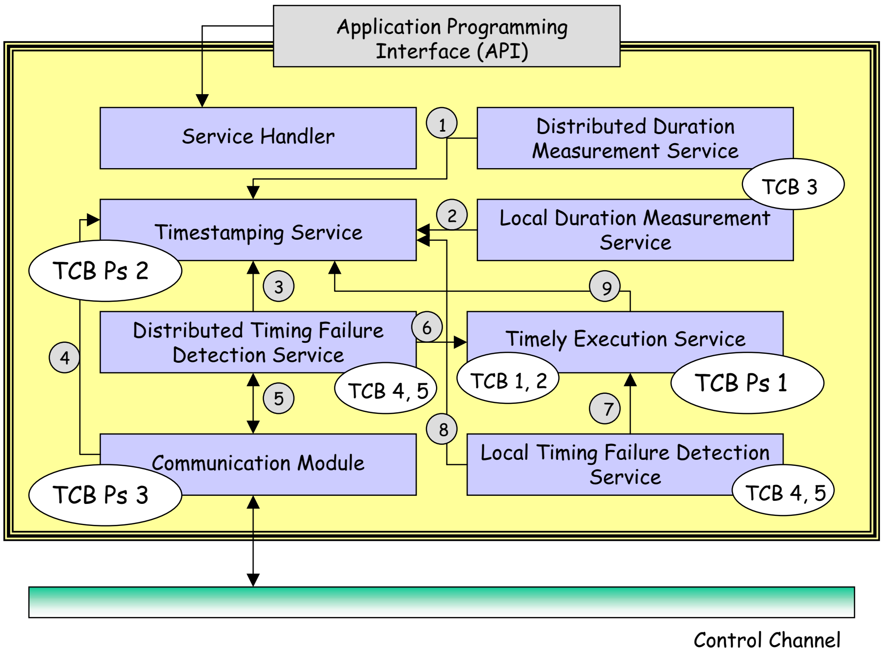

Application Programming Interface (API)
Service Handler
1
Distributed Duration Measurement Service
TCB 3
Timestamping Service
2
Local Duration Measurement Service
TCB Ps 2
3
9
Distributed Timing Failure Detection Service
6
Timely Execution Service
4
TCB 4, 5
TCB 1, 2
TCB Ps 1
5
7
8
Communication Module
Local Timing Failure Detection Service
TCB Ps 3
TCB 4, 5
Control Channel

Figure 2: Architecture of a TCB Component.

Let us review the fault and synchronism model specific of the TCB subsystem. We assume only crash failures for the TCB components, i.e. that they are fail-silent.

Furthermore, we assume that the failure of a local TCB module implies the failure of that site, as seen from the other sites. The crash of a local TCB is easily detected by the other TCB instances and does not affect application processes in other sites (local processes do not exist anymore after a TCB crash). The TCB subsystem enjoys the following synchronism properties:

TCB Ps 1 *There exists a known upper bound $T_{D^1_{max}}$ on processing delays*

TCB Ps 2 *There exists a known upper bound $T_{D^2_{max}}$ on the drift rate of local TCB clocks*

TCB Ps 3 *There exists a known upper bound $T_{D^3_{max}}$ on the delivery delay of messages exchanged between local TCBs*

Property **TCB Ps 1** refers to the determinism in the execution time of code elements by the TCB. Property **TCB Ps 2** refers to the existence of a local clock in each TCB whose individual drift is bounded. This allows measuring local durations, that is, the interval between two local events. These clocks are internal to the TCB. Property **TCB Ps 3** completes the synchronism properties, referring to the determinism in the time to exchange messages among TCB modules in different sites (each site is supposed to have a local TCB module). It is assumed that inter-TCB channels provide reliable delivery, that is, no messages addressed to correct TCBs are lost. The set of all local TCB modules, interconnected by the control channel, constitutes the distributed TCB.

Timely Execution

TCB 1 Eager Execution: *Given any function f with an execution time bounded by T, the TCB is able to execute f within T from the execution start instant*

TCB 2 Deferred Execution: *Given any function f and some delay amount T, for any deferred execution of f triggered at real time t, the TCB will not execute f within T from t*

Duration Measurement

TCB 3 *Given any two events occurring in any two nodes at instants t_s and t_e, the TCB is able to measure the duration between those two events with a known bounded error. The error depends on the measurement method.*

Timing Failure Detection

TCB 4 Timed Strong Completeness: *Any timing failure is detected by the distributed TCB within a known interval from its occurrence*

TCB 5 Timed Strong Accuracy: *Any timely action finishing no later than some know interval before its deadline is never wrongly detected as a timing failure*

Table 1: Basic services of the TCB.

Because the TCB must be a very simple component, it only provides the services considered to be essential to satisfy a wide range of applications with timeliness requirements: ability to measure distributed durations with bounded accuracy;

complete and accurate detection of timing failures; ability to execute well-defined functions in bounded time. Table 1 presents an informal summary of these services (a more detailed presentation was provided in deliverable WP3-D4).

These are the services to be provided at the TCB interface shown in Figure 2. In this figure it is also possible to see the interactions performed among the several modules defined in the component, represented by arrows. The oval shapes labelled with **TCB Ps x** refer to the synchrony properties preserved by the TCB, and the ones labelled with **TCB x** indicate the properties that each module should fulfil in order to provide only correct services to user applications.

The specific definition of the TCB API will be provided in Section 3.1.

2.2 Coverage awareness

The coverage awareness service has been presented in WP2-D3, as a supporting or enabler service for dependable QoS adaptation. The integration between this service and the other support services, namely the TCB services, was also discussed in WP2-D3. Therefore, it is our understanding that it does not make sense to describe this service once again in the present deliverable.

2.3 Resource and Task model

Over the last few years we have seen the proliferation of embedded mobile systems such as mobile phones and PDAs. Pervasive computing is also taking off in which multiple cooperating possibly embedded controllers are used. A new kind of applications can now be envisaged with the emergence of both mobile computing and ubiquitous computing. Applications of such kind are characterised by being largely distributed and proactive, i.e. able to operate without human intervention. A set of further characteristics are also involved such as self context awareness as a means to sense the surrounding environment. Examples of these applications include automatic car control systems in which cars are able to operate independently and cooperate with each other to avoid collisions. Another example is an air traffic control system whereby thousands of aircrafts are proactively coordinated to keep them at safe distances from each other, direct them during takeoff and landing from airports and ensure that traffic congestions are avoided. Smart office systems can also be foreseen in which the intensity of light, room temperature and some other features are automatically tuned according to the user preferences of the persons present in the room.

The CORTEX Project is examining fundamental issues relating to the support of such applications, including the development of middleware for this domain. Importantly, CORTEX applications require the support of anonymous and asynchronous event models, i.e. scenarios including large number of autonomous processing units where a many-to-many communication takes place are well-suited to the anonymous dissemination of information. In addition, systems in which frequent disconnection is likely to happen are well-supported by asynchronous communication as blocking conditions are avoided. Further requirements include support for mobility and non-functional properties such as timeliness and reliability since some of these applications are time-critical. However, current event-oriented middleware technologies do not provide a solution for all the challenges imposed by these applications. As regards timeliness requirements, we believe that resource management plays an important role in providing support for real-time applications. The mechanism that allocates resources in the system should ensure that critical activities will be provided with enough resources to carry out their tasks in a predictable way. Changes in the availability of network resources and periods of disconnection are frequently experienced as mobile computing environments are highly dynamic. This kind of unexpected changes in the environment implies that a means for adapting the system in a dependable way needs to be introduced. Furthermore, mobile applications typically operate on devices with scarce resources, e.g. CPU capacity, system memory and battery life. Therefore, support for the predictable and efficient management of the system resources as well as resource reconfiguration capabilities for achieving adaptation are required. An example of the latter is a redistribution of both CPU-time and memory to the set of activities that the system performs, thus, ensuring that time-critical activities are not disturbed.

Section 3.2 presents a resource management framework which provides support for addressing the timeliness issues mentioned above. The framework makes use of both reflection and component technology. Reflection is a means by which a system is able to inspect and change its internals in a principled way [52]. Basically, a reflective system is able to perform both self-inspection and self-adaptation. To accomplish this, a reflective system has a representation of itself. This representation is causally connected to its domain, i.e. any change in the domain must have an effect in the system, and vice versa. A reflective system is mainly divided into two parts: the base-level and the meta-level. The former deals with the normal aspects of the system whereas the latter regards the system's representation. The meta-level interface is often referred to as the *meta-object protocol (MOP)* [46]. On the other hand, component technology provides great flexibility for the dynamic replacement of components. In addition, the component approach promotes and enhances software reusability. However, component technologies are not mature enough yet. New component standards, such as EJB [57] and the CORBA component model [29], have started to emerge and will consolidate in the next few years. In fact, the implementation of our resource system is developed in OpenCOM [51], which is a lightweight, efficient and reflective model based on Microsoft's COM [55].

2.4 TBMAC Basic Services

In this section the Time-Bounded Medium Access Control (TBMAC) protocol [20, 17] will be reviewed. This section covers background information about the TBMAC protocol that is needed for the discussion in the rest of the deliverable. In particular, building on this review Section 3.4 will present the Application Programmer Interface (API) of the TBMAC protocol provided to higher layers. Finally, Section 4.3 provides an overview of the protocol messages used in the TBMAC protocol and a discussion about the inaccessibility of the TBMAC protocol.

2.4.1 Basics

The Time-Bounded Medium Access Control (TBMAC) protocol is based on time-division multiple access with dynamic but predictable slot allocation. TBMAC uses a lightweight atomic multicast protocol to achieve distributed agreement on slot allocation and employs location information to minimise contention for slots.

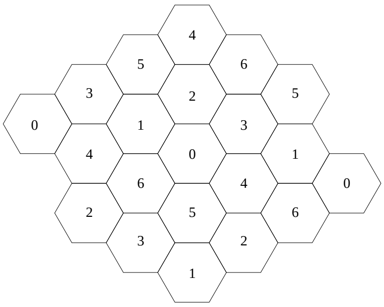


Figure 3: Possible Cell and Channel allocation.

To reduce the probability of the transmissions colliding, the geographical area occupied by the mobile hosts is statically divided into a number of geographical cells. Each cell is numbered and can have arbitrary shape and size but for simplicity, we assume that the cells are hexagons of equal size as illustrated in Figure 3. Each numbered cell is also allocated a distinct radio channel (or CDMA spreading code) to use, maximising the total overall bandwidth available in the ad hoc network.

The boundaries of each of these cells are known to each mobile host in the ad hoc network. To meet this requirement, each mobile host requires access to location information (such as GPS). By a mobile host knowing the cell that it is in, it can then infer the correct radio channel to use.

To further reduce the possibility of collisions, TBMAC divides access to the wireless medium within a cell into two time periods:

1. Contention Free Period (CFP)
2. Contention Period (CP)

Both the CFP and the CP are divided into slots and each period lasts a well-known period of time. Once a mobile host has been allocated a CFP slot, it has predictable access to the wireless medium. The mobile host can then transmit data in its slot until it leaves the cell or fails.

Mobile hosts, that do not have CFP slots allocated to them, contend with each other to request CFP slots to be allocated to them in the CP. The CP is used by mobile hosts that arrive into the cell or that have recently powered on in the cell.

The TBMAC participants reach a distributed agreement on the order of allocations and deallocation of CFP slots. The approach TBMAC uses to provide a total ordering protocol within a cell is to use the synchronous atomic broadcast protocol from Flaviu Cristian [18].

By fixing the number of slots in the CFP, it would appear that we are placing an upper bound on the number of mobile hosts that can be allocated a CFP slot in the cell at any one time. This would mean that the TBMAC protocol is very restrictive and has only limited use. It is possible however to overcome this restriction by allowing a dynamic logical CFP (LCFP) to grow and shrink in the repeating static CFPs [20].

2.4.2 Inter-Cell Communication

As the area occupied by the mobile hosts has been divided up into geographical cells and each cell has been allocated a particular radio channel to use, an obvious question is how does a mobile host communicate with other mobile hosts in neighbouring cells.

If we consider two adjoining cells A & B, then TBMAC uses a static CFP slot for communication from cell A to cell B and another static CFP slot for communication from cell B to cell A. Mobile hosts then atomically broadcast a request to be granted the inter-cell CFP slot. When a mobile host transmits in the inter-cell slot, all the mobile hosts in the adjoining cell will be listening for this transmission.

By using two static CFP slots, it would appear at first glance that TBMAC is placing a severe restriction on the communication between mobile hosts in adjoining cells. However, as we shall see in section 4.3.1, it is possible to allow a dynamically varying number of CFP slots for inter-cell communication.

2.4.3 Entering an Empty Cell

When dividing access to the medium into CFP slots and the CP slots, the most difficult task is to allocate the first CFP slot to a mobile host in a cell. Once there is one mobile host in the cell with a CFP slot then it is possible to use the atomic broadcast protocol to allocate and deallocate slots.

In TBMAC, a mobile host in an empty cell generates a list of CFP slots to use after listening for one full CFP as illustrated in Figure 4.

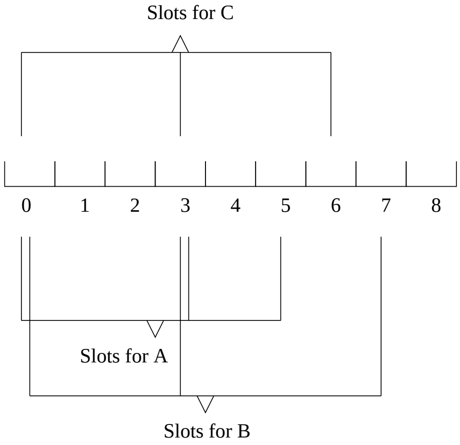


Figure 4: Generated list of slots.

During the next CFP, the mobile host transmits in each CFP slot in its generated list (including the Slot Bitmap in each transmission). The mobile host also listens in each of the other CFP slots that it does not transmit in to gain knowledge about the presence of other mobile hosts and the CFP slots that they are using.

When two or more mobile hosts enter an empty cell and generate a list of CFP slots to use, each mobile host has an inconsistent view of the allocation of CFP slots. After a known number of CFPs (M say), the view of each mobile host of the allocation of CFP slots converges to a consistent view. Figure 5 illustrates how this convergence is achieved.

When a mobile host transmits in each of its generated CFP slots, the mobile host includes information about the other generated slots that it is using. When a mobile host successfully receives a message from another mobile host, the receiving mobile host obtains a list of slots being used by the transmitting mobile host. If a

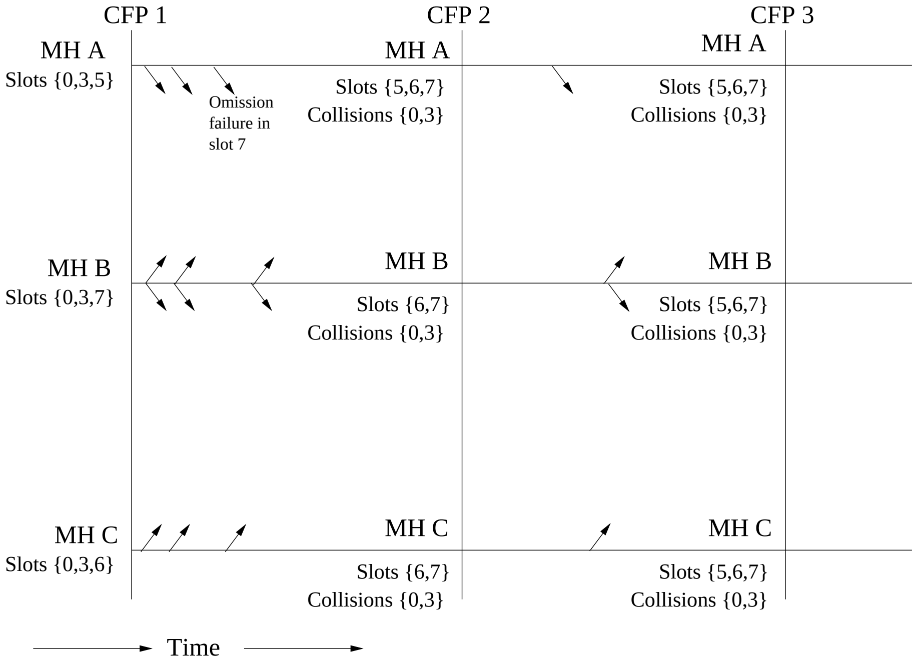


Figure 5: View Convergence of Slot Allocations.

mobile host realises that one of its CFP slots is causing a collision, then the mobile host stops transmitting in this slot in subsequent CFPs.

2.5 CAN

The WAN-of-CAN Structure plays an major role in the CORTEX network architecture. CANs (Controller Area Networks) connect islands of tightly cooperating autonomous computing nodes. At an abstract layer, a CAN constitutes a zone of coherent QoS provision. This may be a fieldbus, an Ethernet, or even a wireless network like bluetooth or an 802.11 infrastructure network. We assume a certain guaranteed level of predictability as an intrinsic property of CANs which usually is considerably higher than what we can expect from WANs. Therefore, although CANs may have a wide spectrum of what level of predictability they provide, their basic protocol mechanisms assist the architecture of higher level communication schemes which allow to define temporal bounds on communication beyond a purely best effort probabilistic approach (although it must be admitted that any system can only guarantee probabilistic bounds!). This actually distinguishes them from WANs where we do not make such assumptions. On a CAN, the protocol and services allow the specification of time bounds and provide mechanisms to enforce these constraints.

In the general interaction model of CORTEX, a CAN may usually link the smart components of a sensor system to the respective computational engines or directly to smart actuators. These components are embodied in an artefact like a smart door, a car, a robot or similar stationary and mobile entities.

Two major issues have to be considered for services and protocols on CANs for the CORTEX specific application environments:

1. Protocols and services have to support the CORTEX event model.

2. A range of timeliness and reliability requirements have to be supported for the dissemination of events.

The basic interaction abstractions are events and event channels through which events are disseminated. An event is specified by a subject and functional and non-functional attributes. The functional attributes reflect the context in which the event was generated, like the location, the time, the mode of operation, etc. The non-functional attributes are related to temporal and reliability issues. They may comprise deadlines for delivery, expiration dates, a coverage and similar parameters. Thus, to a large extend the non-functional attributes of an event specify the requirements for the underlying communication system.

The event channel abstracts the underlying communication system. In application areas like those envisaged in CORTEX [15] non-functional attributes like timeliness and reliability are decisive to support critical system functions. Particularly in a CAN we need to support critical control loops. Hence, the communication system has to provide the adequate resources to meet the respective QoS requirements. An event channel is specified by a subject defining the events which can be disseminated by the channel and non-functional attributes. These attributes characterize the timeliness and reliability properties of the channel, like latencies and the tolerable omission degree. Thus an event channel allows to specify quality parameters explicitly which must be mapped to the underlying network. When setting up an event channel, the respective resources will be allocated.

As a matter of fact, there is a trade-off between the predictability of communication and the needed resources. At the safe end, all communication is statically planned and resources have to be assigned anticipating worst case load and failure assumptions. However, this may only be required for a small number of highly critical services. In fact, critical system services as the TCB [17] could use such highly predictable links to meet its temporal and reliability requirements. Other examples on the application level are tight sensor/actor control loops e.g. for crash avoidance or motor and brake control. In most cases less critical events have to be accommodated by the communication system which also allow a more dynamic system behaviour. However, also for these events, temporal parameters may be needed, e.g. the specification of when an event should be delivered or how long the event is valid. The different requirements are reflected by event channel classes with different properties.

On the architectural level, we distinguish three layers. Figure 6 roughly depicts the layers and the respective abstractions. It relates to Figure 1 provided in the introduction in that the upper middleware layer provides the event specific services.

On the publisher/subscriber layer, the main abstractions are events and different classes of event channels. The layer enables the application to specify channels of different QoS classes and to publish events and subscribe to channels with the respective guarantees.

Mapping the abstractions of the publisher/subscriber layer directly to the underlying network is a tough challenge because the usual abstractions on the network layer are low level messages. Hence, this layer does not match the requirements of group communication, subject-based addressing or the QoS specifications defined for channels. Therefore, an abstract network (AN-) layer is introduced which enriches the properties of the raw network (this may not just be a physical network, but a specific MAC-layer or even a higher OSI level) by additional properties and com-

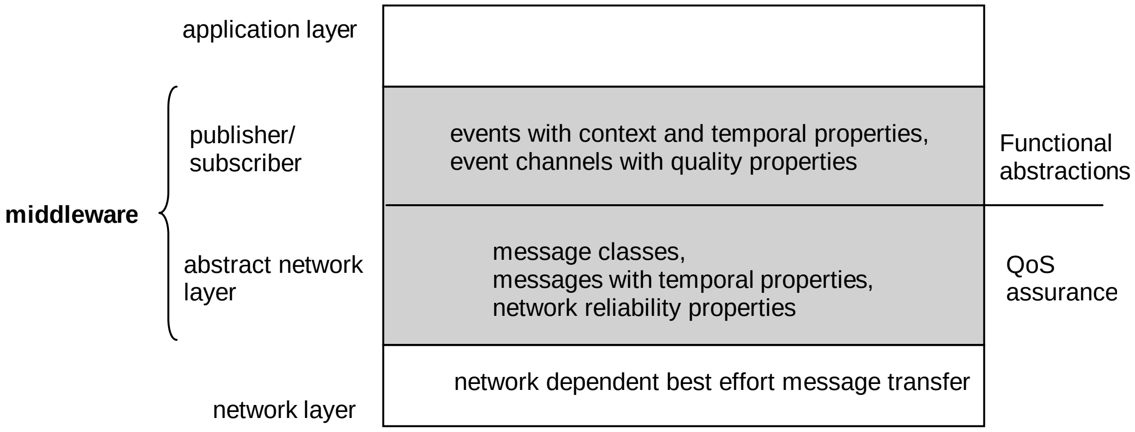

application layer
middleware
publisher/
subscriber
abstract network
layer
network layer
events with context and temporal properties,
event channels with quality properties
message classes,
messages with temporal properties,
network reliability properties
network dependent best effort message transfer
Functional
abstractions
QoS
assurance

Figure 6: Architectural layers.

munication services as e.g. some form of reliable broadcast or group communication and temporal guarantees for the message transfer.

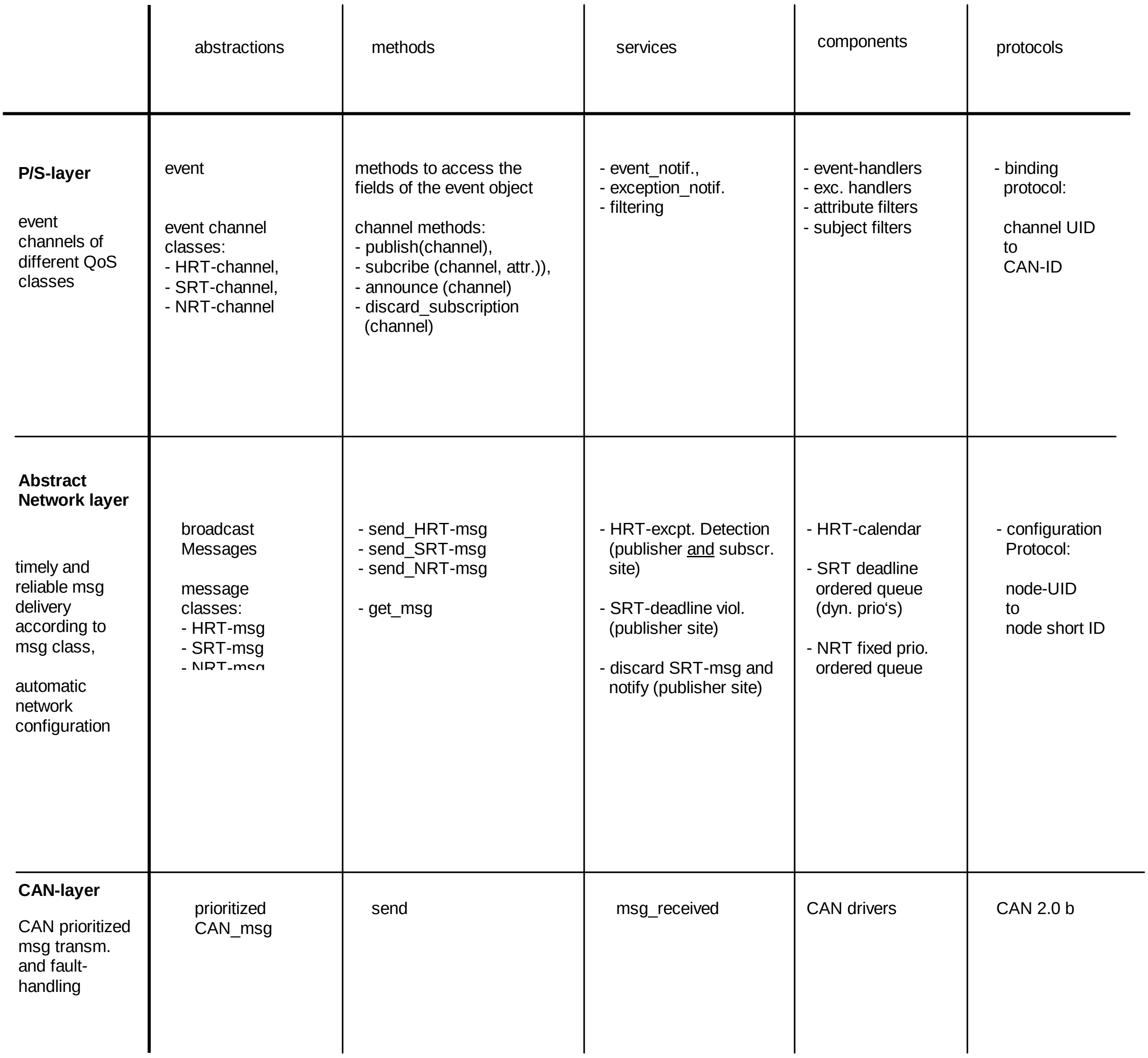
As an example, Figure 7 presents the preliminary specification of protocols and services for a particular CAN (CAN-Bus [28]). It lists the basic abstractions, the methods to deal with these abstractions, the system services available, the components to realize the system services and specific protocols needed to support the abstractions in the distributed architecture. The P/S layer supports events and different classes of event channels with different quality properties. It should be noted that the QoS properties in general depend on what the abstract network layer can provide. Thus, it may not always be possible to e.g. support a hard real-time event channel because the abstract network layer can not provide the respective guarantees. Therefore, the event channel classes supported are dependent on the zones in which publishers and subscribers reside. Figure 7 covers the case of a single zone which is a CAN-Bus.

A detailed description of these abstractions and the mechanisms to enforce them on a CAN-Bus[28] is provided in Section4 of this deliverable with the title: "A Real-Time Event Channel Model for the CAN-Bus". The dynamic binding protocol of the P/S layer and the configuration protocol of the AN-layer are described in [17, 44].

2.6 Adaptable Timed Event Service (ATES)

A proposed solution to handle some of the timeliness requirements of CORTEX system entities is by using a specialized architectural building block, the Timely Computing Base, which provides a set of fundamental services tailored for that purpose. However, since the services provided by the TCB are just the essential ones in order to keep the TCB small, simple and hence reliable, the interface supplied by the TCB is, in a certain way, a low level interface, which should be hidden from sentient objects in the programming model.

The *Adaptable Timed Event Service* (*ATES*) is a middleware event service that extends the CORTEX publish/subscribe programming model with constructors to deal with timeliness requirements. ATES hides from sentient objects the low level services provided by the TCB concerning timing failure detection, coverage stability and awareness, and timely execution, providing high level abstractions more suitable



	abstractions	methods	services	components	protocols
P/S-layer event channels of different QoS classes	event event channel classes: - HRT-channel, - SRT-channel, - NRT-channel	methods to access the fields of the event object channel methods: - publish(channel), - subcribe (channel, attr.)), - announce (channel) - discard_subscription (channel)	- event_notif., - exception_notif. - filtering	- event-handlers - exc. handlers - attribute filters - subject filters	- binding protocol: channel UID to CAN-ID
Abstract Network layer timely and reliable msg delivery according to msg class, automatic network configuration	broadcast Messages message classes: - HRT-msg - SRT-msg - NRT-msg	- send_HRT-msg - send_SRT-msg - send_NRT-msg - get_msg	- HRT-excpt. Detection (publisher and subscr. site) - SRT-deadline viol. (publisher site) - discard SRT-msg and notify (publisher site)	- HRT-calendar - SRT deadline ordered queue (dyn. prio's) - NRT fixed prio. ordered queue	- configuration Protocol: node-UID to node short ID
CAN-layer CAN prioritized msg transm. and fault-handling	prioritized CAN_msg	send	msg_received	CAN drivers	CAN 2.0 b

Figure 7: Properties, services and protocols of architectural layers.

for application programming.

In this context, the main goal of ATES is to extend the CORTEX programming model in order to enable the definition of timeliness requirements of sentient objects.

Events on ATES are timed, in the sense that they must be produced, disseminated and consumed at the subscribers within certain timing constraints. Since ATES is a middleware component executing on the payload part of the system (accordingly to the TCB system model), it can only provide guarantees with respect to the timely production, dissemination and consumption of events, as allowed by the underlying payload system. However, by using the timing failure detection service of the TCB, ATES can take advantage of the semantics of this service in order to enforce the timely execution of safety procedures when the specified timing constraints are not fulfilled.

Timeliness requirements are derived from the sentient objects internal logic whose correctness depends on the time required to execute some actions. However, since the environments envisaged in CORTEX are characterized by attributes like openness, heterogeneity and large-scale, they exhibit poor baseline properties with respect to timeliness and reliability. Therefore, the ability to satisfy the timing constraints of applications can be compromised or, in other words, it may be difficult to enforce the coverage of timing assumptions, which may degrade over the execution.

ATES provides appropriate constructors to cope with the temporal uncertainty of the environment. Adaptability results from the capacity of ATES to provide operation modes (appropriate for the construction of some application classes) in which coverage awareness can be used as a means to ensure coverage stability.

In ATES, the specification of timeliness requirements is made through $< bound, coverage >$ pairs, where a certain timing *bound* must be satisfied with a probability indicated by *coverage*. One of the adaptation modes provided by ATES can be used to ensure that the coverage stability property is preserved. Roughly speaking, in this operation mode ATES will dynamically adjust the relevant timing variables in order to meed the dynamics of the environment. More specifically, ATES will adjust the *bound* value that is used by the application in order to maintain the coverage near to the desired value. ATES provides another operation mode, which can be described as a coverage awareness mode, in which it dynamically estimates the *coverage* that is actually possible to provide for a given *bound*, and lets the application know of this coverage. ATES implements these adaptation functionalities using the TCB *QoS* extensions that were presented in another deliverable [16].

ATES notifies sentient objects of changes on the available *QoS* by calling adaptation procedures previously defined by sentient objects. Moreover, ATES guarantees that these adaptation procedures are triggered in real-time, that is, upon detecting changes on the available *QoS* ATES immediately executes the proper adaptation procedures. For that, the timely execution service of the TCB is used. Section 3.5 will present the ATES application programming interface (API).

3 Service interfaces

3.1 TCB API

A TCB component provides services through a well defined API to user applications that execute on the payload part of the system. Observe that this interface must ensure a correct interaction between a synchronous TCB component and potential asynchronous applications. No timing assumptions are made regarding user applications, whereby the latency on service invocation is not bounded. The same is also true for processing the data provided by the TCB, like the ones resulting from service requests performed by applications.

Duration Measurement

The timestamping service of the TCB is provided through the following function:

```
timestamp ← getTimestamp ()
```

Since applications execute on the payload part of the system, a timestamp provided by the TCB is necessarily affected by the latency of the service invocation. This means that no guarantees can be given about the time at which the timestamp is used and, therefore, about the accuracy of this timestamp in absolute terms. Nevertheless, it is possible to measure the duration of an interval bounded by the instants reflected by two timestamps. If a computation takes place during the mentioned interval, that is, between the two timestamping operations, it is possible to obtain an upper bound for its execution time. The TCB provides support for measuring local durations through the following functions:

```
                id ← startLocalMeasurement (Tevt_start)
Tevt_end, duration ←  endLocalMeasurement (id)
```

The function `startLocalMeasurement` is used to begin a measurement operation of a local action started at `Tevt_start`. The operation is identified inside the TCB by `id`. To finish a measurement operation, an application must call the function `endLocalMeasurement` providing the respective `id`. The service returns a timestamp that signals the end of the measured interval (`Tevt_end`) and, obviously, the measured duration (`duration`).

The TCB also provides support for distributed measurements. A distributed measurement is a measurement of a distributed duration, resulting from the execution of a distributed action between two different nodes of the system. The most simple distributed execution corresponds to the transmission of a message through the (payload) communication channel. The distributed duration measurement service provides the following two functions (only service specific parameters are indicated):

```
                          ← sendMessage (Tevt_start)
Tevt_end, duration, error ←  receiveMessage ()
```

It is assumed that messages can be broadcast on the communication channel and therefore several processes may execute the function `receiveMessage`. The parameters are similar to the ones of the local measurement service, except for the return value `error` of the `receiveMessage` function, corresponding the the measurement error. Note that a measurement algorithm must be used, which delivers an upper bound for the measurement to which corresponds a given error. This function exhibits a blocking behavior, waiting for a message to be received on the payload communication channel. Upon returning, `Tevt_end` contains a timestamp corresponding to the instant at which the TCB received the message.

Timely Execution

The timely execution service of the TCB supports the execution of small and sporadic tasks, satisfying their timeliness requirements. The timely execution service guarantees that the execution of a function will finish before a specified deadline and will not start before a specified liveline. The interface of this service has the following specification (note that this specification merges both eager and deferred execution, corresponding respectively to properties **TCB 1** and **TCB 2** as described in Section 2.1):

```
Tevt_end ← startExecution (Tevt_start, delay, max_exec, func)
```

The deadline for the execution of `func` is given by the parameters `Tevt_start` and `max_exec`. The former indicates a reference instant for the execution start. The latter, a duration specifying the maximum execution time, can be summed to `Tevt_start` to obtain the deadline for the execution. The parameter `delay` specifies the deferral time for starting the execution of `func`, to be counted from `Tevt_start`. The service returns `Tevt_end`, which indicates the termination instant signaled by the TCB for the execution of `func`. An invocation of this service blocks the calling application until the end of the execution.

Observe that it is not possible to accept all service invocations, either because of intrinsic impossibility to satisfy the parameters defined for request (for instance, if a TCB receives a request for an execution that must finish before a deadline t at some time after t), or because of feasibility constraints related with scheduling the request, given the available (processing) resources at the moment the request is issued.

Timing Failure Detection

The ability to detect timing failures in a timely manner is perhaps the most important service provided by the TCB. The timing failure detection service may be used to simplify the development of applications and to improve their reliability. Similarly to the duration measurement service, the timing failure detection service is divided into a service for detecting local timing failures and into another for detecting distributed timing failures. The following two functions describe the interface of the local timing failure detection service:

```
id ← startLocalDetection (Tevt_start, spec, func, deadline)
      Tevt_end, delay, faulty  ← endLocalDetection (id)
```

The service is started by invoking the function `startLocalDetection`. The timestamp `Tevt_start` indicates the start of the observed action and `spec` the maximum duration allowed for its execution. The service must detect failures in a timely manner, therefore it does not accept requests for the observation of timed actions that have already failed. Timely detection implies that if the execution of an action fails the TCB can timely execute a procedure in response to the timing failure, a the procedure specified in `func`, which must be concluded within a `deadline` from the instant at which the timing failure occurs. The particular function that handles the timing failure depends on the application logic. This function may execute a variety of actions, ranging from doing a fail-safe shutdown of the system to the update of variables or the execution of low-level I/O commands to controls external devices. For instance, consider the example of an application that controls the operation of an external hardware device. If the system is unable to secure the correct operation of the device due the occurrence of a timing failure, the function (`func`) must execute the necessary operations in order to stop the device operation in a safe state. Observe that the TCB can use the timely execution service to internally execute `func` (in this case with a non blocking semantic).

To terminate an observed execution, an application must call the function `endLocalDetection` in order to deactivate the failure detection and to receive the results of the respective execution, comprising the instant signaled by the TCB for the final of the execution (`Tevt_end`), its duration (`delay`) and a flag to indicate if the execution was timely or `faulty`.

A distributed action requires that at least one message is exchanged between two processes and allows to observe execution times of actions whose start and end events are causally connected by a such message. A distributed action may be carried out among several processes (such as when a message is broadcast). In such cases, the execution on each distributed interaction must be observed individually.

The service is supplied by the following functions:

```
      id ← sendMessageTFD (Tevt_start, spec, func, deadline)
   id ← sendMessageWRemoteTFD (Tevt_start, spec, func, deadline)
                      id ←  receiveMessage ()[1]
                    Tevt_end ← endDistAction (id)
duration_1, faulty_1, lateness_degree_1 ...  duration_n, faulty_n,
              lateness_degree_n ←  waitInfo (id)
```

Either `sendMessageTFD` or `sendMessageWRemoteTFD` can be used to start a timing failure detection operation. These functions differ in the way and place where the timely execution of the handler is done. In the former case, `func` is executed as soon as a timing failure is detected on the *sender* side. In the latter, `func` is executed on *receiver* side. The timestamp `Tevt_start` signals the start of the observed action and `spec` specifies the duration allowed for the execution. If a timing failure occurs the function `func` must be concluded within a `deadline`. Each operation has an unique `id`.

To signal the TCB of the termination of a distributed action, `endDistAction` can be used after the associated message on the payload channel is returned by

[1] `receiveMessage` is a polymorphic function. When a received message is concerned to a distributed measurement operation the function returns `Tevt_end`, `duration` and `error`. Otherwise, the received message is associated to a timing failure detection operation in which case the function returns `id`.

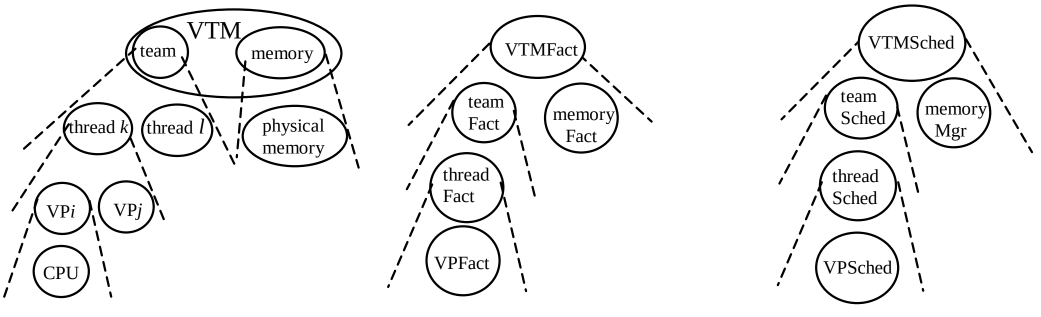
receiveMessage. The function endDistAction returns a timestamp (Tevt_end) that contains the instant at which the TCB has received and marked the termination indication.

To obtain the results of the distributed execution observed by the timing failure detection service it is necessary to call waitInfo. This function will block the calling application until the TCB is able to provide all the information relative to the completion of the operation identified by id. This includes measured durations (duration_x), failure status (faulty_x) and lateness degrees (lateness_degree_x) relative to all destinations addressed involved in the distributed execution.

3.2 Resource Management framework

3.2.1 Resource model

The most important elements of the resource model are abstract resources, resource factories and resource managers [25]. Abstract resources explicitly represent system resources. In addition, there may be various levels of abstraction in which higher-level resources are constructed on top of lower-level resources. Resource managers are responsible for managing resources, that is, such managers either map or multiplex higher-level resources on top of lower-level resources. Furthermore, resource schedulers are a specialisation of managers and are in charge of managing processing resources such as threads or virtual processors (or kernel threads). Lastly, the main duty of resource factories is to create abstract resources. For this purpose, higher-level factories make use of lower-level factories to construct higher-level resources. The resource model then consists of three complementary hierarchies corresponding to the main elements of the resource model. Importantly, virtual task machines (VTMs) are top-level resource abstractions and they may encompass several kinds of resources (e.g. CPU, memory and network resources) allocated to a particular task.



(a) A hierarchy of abstract resources *(b) A factory hierarchy* *(c) A manager hierarchy*

Figure 8: A particular instantiation of the Resource Meta-model.

As an example, a particular instantiation of the framework is shown in Figure 8 (note, however, that the framework does not prescribe any restriction in the number of abstraction levels nor the type of resources modelled). At the top-level of the resource hierarchy is placed a VTM, as shown in Figure 8(a), which encompasses both memory buffer and a team abstraction. The team abstraction in turn includes two or more user level threads. Moreover, a user level thread is supported by one or more virtual processors (VPs), i.e. kernel level threads. At the bottom of the hierarchy are located physical resources. In addition, a VTM factory is at the top of the factory hierarchy and uses both a memory and a team factory. The team factory then is supported by both the thread and the virtual processor factory as depicted in Figure 8(b). Finally, the manager hierarchy is shown in Figure 8(c). The team scheduler and the memory manager support the VTM scheduler to suspend a VTM by temporally freeing CPU and memory resources respectively. The thread scheduler in turn allows the team scheduler to suspend its threads. Finally, the VP scheduler supports the preemption of virtual processors. Conversely, this hierarchy also provides support for resuming suspended VTMs.

3.2.2 The Resources Meta-object protocol

The component types of the resource model are shown in Figure 9. Such component diagram may be used to support a particular instantiation of the resource framework. Passive resource components represent non-processing resources such as system memory and battery life. In contrast, jobs are capable of performing some activity, that is, they receive messages and process them. Both passive resources and jobs are created by factories as shown in Figure 9. In addition, passive resources are managed by managers. However, since jobs are processing resources, they are managed by schedulers instead. In addition, a management policy component determines the management strategy that managers employ. Similarly, schedulers use a scheduling policy component for determining the execution order of their associated jobs.

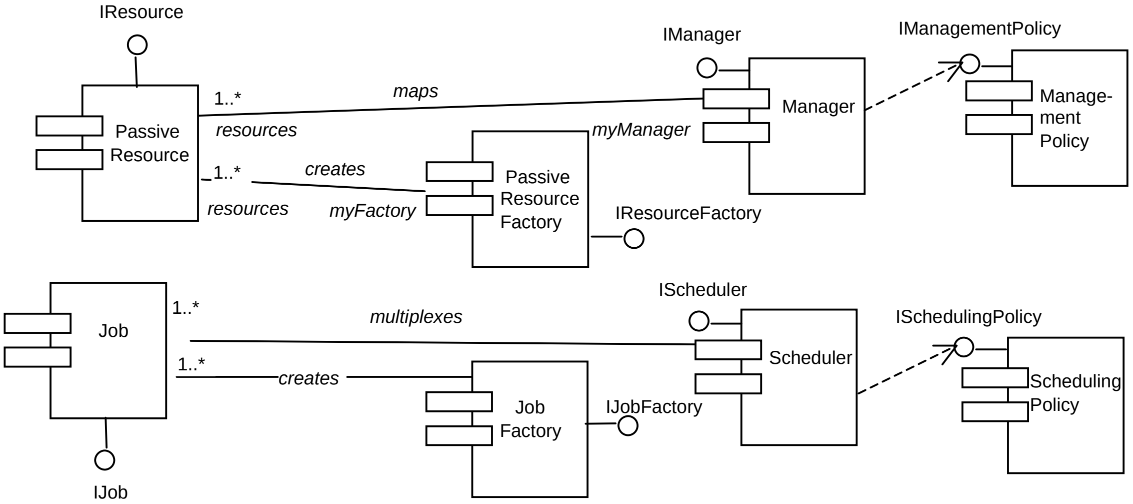


Figure 9: UML component diagram of the Resource model.

All component types support an interface with operations to transverse their associated abstraction hierarchies: `getLL()`, `setLL()`, `getHL()` and `setHL()`. For instance, the resource hierarchy may be traversed by applying the `getLL()` operation at the top-level, i.e. the VTM. This operation would be later applied to the lower-level resources, and so on. Both the higher-level and the lower-level of an entity in the hierarchies may be set by accessing the operations `setHL()` and `setLL()`

respectively. The Access to both the manager and factory of a passive resource can be obtained through the interface Iresource, as shown below. The references to the manager and factory of an abstract resource are registered by the operations setManager() and setFactory() respectively.

```
interface IResource : IUnknown {
    HRESULT getLL([out, size_is(,*maxRes)] IResource**, [out] long* maxRes);
    HRESULT setLL([in, size_is(,maxRes)] IResource**, [in] long maxRes);
    HRESULT getHL([out] IResource**, [out] IJob**);
    HRESULT setHL([in] IResource*, [in] IJob*);
    HRESULT getManager([out] IManager**);
    HRESULT setManager([in] IManager*);
    HRESULT getFactory([out] IResourceFactory**);
    HRESULT setFactory([in] IResourceFactory*);
};
```

The interface of a job includes the operations getSchedParam() and setSchedParam(). The former is in charge of accessing predefined settings. The latter is responsible for performing a control admission test. If successful, resources are reserved and the scheduling parameters are set. The operation run() allows for the execution of a function with associated parameters. Jobs can also be suspended by using the operation suspend() and resume() respectively.

```
interface IJob : IUnknown {
    HRESULT getLL([out, size_is(,*maxRes)] IResource**, [out] long* maxRes,
            [out, size_is(,*maxJob)] IJob**, [out] long* maxJob);
    HRESULT setLL([in, size_is(,maxRes)] IResource**, [in] long maxRes,
            [in, size_is(,maxJob)] IJob**, [in] long maxJob);
    HRESULT getHL([out] IJob**);
    HRESULT setHL([in] IJob*);
    HRESULT getManager([out] IScheduler**);
    HRESULT setManager([in] IScheduler*);
    HRESULT getFactory([out] IJobFactory**);
    HRESULT setFactory([in] IJobFactory*);
    HRESULT GetSchedParam([out] OLECHAR* schedParam);
    HRESULT SetSchedParam([in] OLECHAR* schedParam);
    HRESULT run([in] void* function, [in] void* parameters);
    HRESULT suspend();
    HRESULT resume();
};
```

The interfaces of factories expose an operation for the creation of abstract resources, as shown below. This operation is also responsible for associating the resource with a resource manager. In case of creating processing resources, a job factory is used and the scheduling parameters should be indicated. This interface also provides the operation getResources() that returns references of the resources created by the factory.

```
interface IResourceFactory : IUnknown {
    HRESULT getLL([out, size_is(,*maxRes)] IResourceFactory**, [out] long* maxRes);
    HRESULT setLL([in, size_is(,maxRes)] IResourceFactory**, [in] long maxRes);
    HRESULT getHL([out] IResourceFactory**, [out] IJobFactory**);
    HRESULT setHL([in] IResourceFactory**, [in] IJobFactory**);
    HRESULT newResource([in] int size, [in] OLECHAR* policy);
    HRESULT getResources([out, size_is(,*maxRes)] IResource**, [out] long* maxRes);
};

interface IJobFactory : IUnknown {
    HRESULT getLL([out, size_is(,*maxRes)] IResourceFactory**, [out] long* maxRes,
            [out, size_is(,*maxJob)] IJobFactory**, [out] long* maxJob);
    HRESULT setLL([in, size_is(,*maxRes)] IResourceFactory**, [in] long maxRes,
            [in, size_is(,maxJob)] IJobFactory**, [in] long maxJob);
    HRESULT getHL([out] IJobFactory**);
    HRESULT setHL([in] IJobFactory*);
    HRESULT newResource([in] int size, [in] OLECHAR* policy, [in] OLECHAR* schedParam,
            [out] IJob** interf);
    HRESULT getResources([out, size_is(,*maxJob)] IJob** interf, [out] long* maxJob);
};
```

In addition, the VTM factory should implement the interface `IVtmFactory` which offers operations for obtaining the associated task of a VTM and vice versa, i.e. the `IJob` interface of a VTM. An operation for obtaining the interface `Ischeduler` of the VTM scheduler is also provided.

```
interface IVtmFactory : IUnknown {
    HRESULT getVtm([in] OLECHAR* task_name, [out] IJob** vtm);
    HRESULT getTask([in] IJob* vtm, [out] OLECHAR* task_name);
    HRESULT getVtmScheduler([out] IScheduler** vtmScheduler);
};
```

The manager's interface exposes the operation `admit()` which performs an admission control test that determines whether or not there are enough resources to satisfy a resource request. In a successful case, resources may be reserved by using the operation `reserve()`. Reservations can then be liberated by invoking the operation `expel()`. These three operations are delegated to the policy component. Similar to factories, through the operation `getResources()`, resource managers are able to retrieve the references of the resources that are mapped or multiplexed. Such references may be included or removed from a manager's registry by using the operations `addResource()` and `removeResource()` respectively. In addition, the management policy is obtained by accessing the operation `getPolicy()` whereas the operation `setPolicy()` allows the user to set the management policy of the manager, i.e. the management policy component is dynamically changed.

```
interface IManager : IUnknown {
    HRESULT getLL([out, size_is(,*maxMgrs)] IManager**, [out] long* maxMgrs);
    HRESULT setLL([in, size_is(,*maxRes)] IResource**, [in] long maxRes);
    HRESULT getHL([out] IResource**, [out] IJob**);
    HRESULT setHL([in] IResource*, [in] IJob*);
    HRESULT getPolicy([out] OLECHAR** policy);
    HRESULT setPolicy([in] OLECHAR* policy);
    HRESULT admit([in] OLECHAR* resourceAmount);//may define one or more parameters
    HRESULT reserve([in] OLECHAR* resourceAmount);
    HRESULT expel([in] OLECHAR* resourceAmount);
    HRESULT getResources([out, size_is(,*maxRes)] IResource**, [out] long* maxRes);
    HRESULT addResource([in] IResource**);
    HRESULT removeResource([in] IResource**);
};
```

The scheduler's interface provides similar operations to the manager's but additionally include operations for suspending and resuming processing resources by invoking the suspend() and resume() operations respectively. Lastly, the order of execution of multiplexed resources is determined by the schedule() operation which is also delegated to the policy component.

```
interface IScheduler : IUnknown {
    HRESULT getLL([out, size_is(,*maxMgr)] IManager**, [out] long* maxMgr,
            [out, size_is(,*maxSched)] IScheduler**, [out] long* maxSched);
    HRESULT setLL([in, size_is(,maxMgr)] IManager**, [in] long maxMgr,
            [in, size_is(,maxSched)] IScheduler**, [in] long maxSched);
    HRESULT getHL([out] IScheduler**);
    HRESULT setHL([in] IScheduler*);
    HRESULT getPolicy([out] OLECHAR** policy);
    HRESULT setPolicy([in] OLECHAR* policy);
    HRESULT admit([in] OLECHAR* resourceAmount);//may define one or more parameters
    HRESULT reserve([in] OLECHAR* resourceAmount);
    HRESULT expel([in] OLECHAR* resourceAmount);
    HRESULT getResources([out, size_is(,*maxJob)] IJob**, [out] long* maxJob);
    HRESULT addResource([in] IJob*);
    HRESULT removeResource([in] IJob*);
    HRESULT suspend([in] IJob*);
    HRESULT resume([in] IJob*);
    HRESULT schedule([in] int quantum);
};
```

Policy components are in charge of performing control admission tests, resource reservation and resource liberation. An operation for obtaining the policy deployed by the component is also provided.

```
interface IManagementPolicy : IUnknown {
    HRESULT getPolicy([out] OLECHAR** policy);
    HRESULT admit([in] OLECHAR* resourceAmount);
    HRESULT reserve([in] OLECHAR* resourceAmount);
    HRESULT expel([in] OLECHAR* resourceAmount);

};
```

Scheduling policy components additionally offer operations for scheduling jobs and the dispatch() operation for obtaining the next job to be executed.

```
interface ISchedulingPolicy : IManagementPolicy {
    HRESULT schedule([in] int quantum, [in, size_is(,maxJob)] IJob**,
            [in] long maxJob);
    HRESULT dispatch([out] IJob**);
};
```

3.3 The Task model

3.3.1 Overview

The main feature of the task model is that it offers support for the high-level analysis and design of the system's resource management. More precisely, the task model allows us to define both how system resources are allocated in a distributed system and the resource management policies that are used. This model also prescribes the level of quality of service for services provided by such a system. For instance, for different services of the same type, such as audio transmission, more than one service offering different level of QoS may be defined.

A *task* is defined as a logic unit of computation which has an amount of resources allocated. Examples of tasks are activities performed by the system such as transmitting audio over the network or compressing a video image. From the programmatic point of view, a task may involve either a single invocation sequence or multiple invocation sequences. The simplest case for a sequence is that whereby only one operation is invoked. A *task instance* is then an occurrence of a task; a task may be related to one or more task instances. For example, the task of receiving incoming requests from the network may have several instances when there is a concurrent access to the server and there is a multi-threaded policy for attending requests.

The task model is concerned with both application services and middleware services. Thus, we take a task-oriented approach for managing resources in which services are broken into tasks and are accommodated in a task hierarchy. Top-level tasks are directly associated with the services provided by a distributed system. Lower-levels of this hierarchy include the partition of such services into smaller activities, i.e. *sub-tasks*. Sub-tasks are denoted as follows:

```
Task.sub-task.sub-sub-task...
```

For instance, an audio transmission task referred to as `transmitAudio` may be partitioned into two subtasks, namely `transmitAudio.send` and `transmitAudio.receive`. The former is in charge of sending audio packets whereas the second sub-task is responsible for receiving stream data. This approach offers resource management modelling of both coarse- and fine-grained interactions. The former is achieved by defining coarse-grained tasks (i.e. tasks spanning components and address spaces boundaries) and the latter is done by using task partitioning. In addition, tasks are not necessarily disjoint and may be interconnected. For instance, a component running one task may invoke another component concerned with a different task.

3.3.2 Tasks and VTMs

Tasks have an associated VTM as said before. Hence, a VTM represents a virtual machine in charge of supporting the execution of its associated task. VTMs also represent a higher-level of resource management. They are an encapsulation of the pool of resources allocated for the execution of their associated tasks. VTMs isolate the resources that a service uses to have a better control of the desired level of QoS provided by it. That is, the resources a task uses for execution are localised. Hence, it is straightforward to access the resources of a task when it is under-performing to reconfigure its resources.

The UML model in Figure 10 illustrates other details concerning the relationship between tasks and VTMs. There is a one-to-one mapping between a task hierarchy and a resource hierarchy. The top-level task of a task hierarchy is directly associated with a composite VTM placed at the top of the resource hierarchy. This composite VTM may encompass various local and/or remote VTMs. Tasks of this kind are composite tasks, which are constituted by a number of sub-tasks. Sub-tasks may be either interleaved or disjoint. In the former case, sub-tasks are associated with

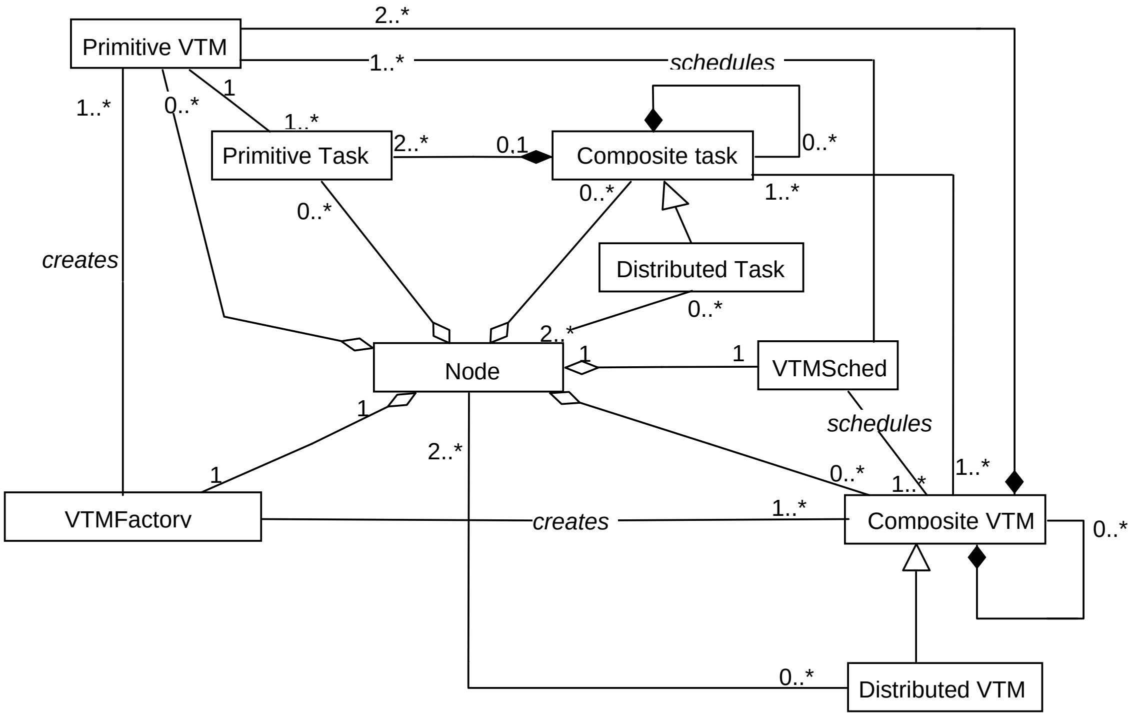


Figure 10: UML diagram of relationships between Tasks and VTMs.

a single operation invocation sequence whereas in the latter two or more sequences are involved. Sub-tasks that are not further partitioned are called primitive tasks and are only related to a single node. However, distributed tasks involve two or more nodes. It should be noted that sub-tasks may also be composite and even distributed. Similarly, VTMs not containing other VTMs as lower-level resources are named primitive VTMs. Hence, the bottom-levels of a task hierarchy consist of primitive tasks which are associated with primitive VTMs. Primitive VTMs are then defined according to the specific platform characteristics of the node of residence. Moreover, composite VTMs are involved with more than one local task and distributed VTMs include two or more nodes. Importantly, distributed VTMs may recursively encompass other distributed VTMs. Finally, VTM schedulers and VTM factories are defined on a per-node basis.

3.3.3 Task graph configurations

A task graph is a directed graph in which an operation invocation sequence (i.e. a task) is represented in terms of components, component interfaces and component interface operations. Different from component configurations, task graphs define the specific interface operations a task path goes through. There is a great variety of the task graph configurations that can potentially exist. An example of this complexity is the fact that several tasks may be defined over the same component configuration path. Since an interface may include several operations, different tasks may run through the same interfaces by accessing different operations. Moreover, different tasks may even go across the same operations when the same component configuration is used for different purposes (e.g. marshalling video streams and marshalling audio streams). Therefore, components, interfaces, and operations may participate in more than one task at the same time. Furthermore, the transition from one task to another one is an important issue that contributes to define how task graphs are interconnected. Such transitions are carried out by task switching points. Interestingly, a task switching point corresponds to a change in the underlying resource pool to support the execution of the task that has come into play. A task switching point is essentially defined as a triplet including a component, an interface, and an exported operation. Imported operations are not used for defining these points for the sake of simplicity. It is only needed one point in the interaction path of two components to define a task switch. This point could be defined in either side of the interaction, however, defining it in the exporter side is more natural as this side is in charge of attending requests.

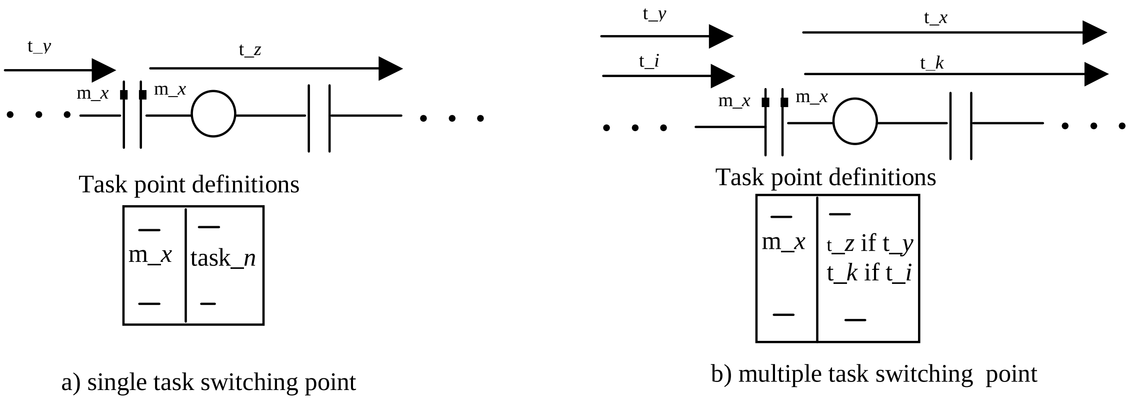


Figure 11: Types of Task Switching Points.

There are two types of task switching points as depicted in Figure 11. A *single task switching point*, depicted in Figure 11(a), can only switch to a single task whereas a *multiple task switching point* may switch to two or more different tasks, as shown in Figure 11(b). The former maps a switching task point to a single task. As an example of a single task switching point consider a protocol stack in which each layer is associated with a different task. Hence, when going from one layer to the adjacent one a task switch takes place. In contrast, a multiple task switching point selects, from a set of tasks, the task to switch according to the current task. To accomplish this, ”If” statements are used to define the possible options within a task point. As an example of a multiple-task switching point, consider a task graph of a video stream connection v$_a$ which includes a filter component bound to a compressor component and these components are part of task t$_y$ and t$_z$ respectively. Similarly, there is another video stream connection v$_b$ that uses the same instances for the both the filter and the compressor, although they are associated with tasks t$_i$ and t$_k$ respectively. Thus, the compressor will switch to task t$_z$ if the filter was executed as part of task t$_y$, otherwise it will switch to task t$_k$.

3.4 TBMAC API

The TBMAC API can be divided into a number of distinct parts:

1. Initialisation
2. Slot Management

3. Communication

Each of these parts will now be discussed followed by a discussion of issues that higher layers, above TBMAC, should be aware of.

3.4.1 Initialisation

Firstly, before the TBMAC protocol starts, the participants in the protocol are required to agree on a number of pieces of information. Firstly, the participants need to agree on the number of slots in the CFP and in the CP. In addition, participants should also agree on the duration (size of packet transmitted) in a CFP and a CP slot.

```
err set_cfp_size( in size )
err set_cp_size( in size )
err set_cfp_slot_size( in pkt_size )
err set_cp_slot_size( in pkt_size )
```

Higher layers, above TBMAC, would also need to query these values during execution of the protocol to, for example, implement QoS and/or Admission Control.

```
err get_cfp_size( out size )
err get_cp_size( out size )
err get_cfp_slot_size( out pkt_size )
err get_cp_slot_size( out pkt_size )
```

Secondly, the TBMAC protocol participants also need to agree on a division of the geographical area, covered by the participants into a set of geographical cells. As a participant in the TBMAC protocol moves, it changes the geographical cell that it is in. A layer above the TBMAC protocol is needed to monitor the mobile hosts position and to then notify the TBMAC protocol of a change of geographical cells. This higher layer would also initialise the TBMAC protocol with the initial position of the mobile host.

When the higher layer notices a change in geographical cells, it then notifies the TBMAC layer passing in the current cell number (i.e. the cell just entered) and a list of the numbers of the cells neighbouring the current cell.

```
err change_in_geocell( in current_cell,
                       in list_of_neighbours[ ],
                       in list_length )
```

Once the participants have agreed on a set of geographical cells, TBMAC would then allocate a corresponding radio channel.

3.4.2 Slot Management

As the TBMAC protocol executes, layers above TBMAC will request CFP slots to be allocated and deallocated. In the TBMAC protocol, a mobile host (which does not have a CFP slot) requests a slot by transmitting a message in the CP. This message transmission can be corrupted due to contention with another mobile host transmitting a similar message. Therefore, the allocation of a cfp slot also needs to be monitored by such a mobile host.

err alloc_cfp_slot(out slot_ref, in address, in req_type)
err dealloc_cfp_slot(in slot_ref)
err monitor_cfp_alloc(in slot_ref)

Layers above the TBMAC layer will also need to query the TBMAC protocol for the number of slots allocated and to which mobile hosts these slots have been allocated.

```
err get_allocated_slots(  out list_length,
                          out slot_refs_owners_list[ ] )
```

Similarly, higher layers will also need to query the TBMAC protocol for the number of inter-cell slots that have been allocated, which mobile hosts these inter-cell slots have been allocated to and which adjoining cell each inter-cell slot will communicate with.

```
err get_inter_cell_slots(  out list_length,
                           out list_of_owners_and_cells[ ] )
```

The last slot management issue is how higher layers become aware of a change in the allocation of CFP slots. One option would be for the higher layer to periodically poll the TBMAC layer for changes in the allocation of slots by calling the above `get_allocated_slots(..)` function. Another option would be for the higher layer to register a callback to be notified when the allocation changes (or when a request to be allocated a CFP slot is received).

```
err add_slot_mgt_callback( out func_ref, in callback_func )
err del_slot_mgt_callback( in func_ref )
```

3.4.3 Communication

Once a mobile host has a CFP slot allocated to it, the mobile host can then transmit messages in its CFP slot. There are two options for the reception of messages. Either messages can be queued by the TBMAC layer or a higher layer can register a call back to be notified when a message arrives.

```
err send( in slot_ref, in pkt, in pkt_len )
err recv( out address, out pkt, out pkt_len )
err add_recv_callback(  out func_ref, in slot_ref,
                        in callback_recv_func )
err del_recv_callback( in func_ref )
```

3.4.4 Higher layer issues

The TBMAC protocol only provides a set of primitives for the management of CFP slots in a cell. The decision as to whether a mobile host should get 3 CFP slots or 1 CFP slot needs to be taken by a layer above the TBMAC protocol. This decision would typically be done by a QoS layer above the TBMAC protocol.

Similarly, the task of allocating CFP slots to newly arriving mobile hosts would fall to an Admission Control layer (possibly in conjunction with the QoS layer to ensure that a certain level of QoS is maintained).

Another problem not addressed by the TBMAC protocol is which mobile host in a cell is allocated a particular inter-cell slot. Again, an Admission Control layer could decide which mobile host is allocated a particular inter-cell slot. The mobile host (or hosts) that is allocated an inter-cell slot takes on the role of a gateway between two cells. This gateway role would also be a function of the routing layer, being used to route packets between cells. Therefore, there is an interaction between the Admission Control layer and the Routing layer which would need to be addressed.

Finally, TBMAC does not specify the number of dynamic inter-cell slots there are between any two adjacent cells. The decision on whether or not to allocate more inter-cell slots should be taken at the Routing layer in conjunction with the QoS layer (or vice versa). However, TBMAC does provide the functionality to allocate and deallocate inter-cell slots between adjoining cells (see Section 4.3.1).

3.5 ATES API

Sentient objects communicate and cooperate using a publish/subscribe event model. Each sentient object should publish events in order to disseminate relevant information to the surrounding environment. When using ATES, before starting to publish events, sentient objects should specify some necessary parameters, including the desired *QoS* level (a $< bound, coverage >$ pair) and the notification procedures that should be called when timing failures and *QoS* changes occur. The following function is used for this purpose:

```
 ← requestPublishing (event, coverage, bound, mode, dev,
on_timing_failure_handler, on_QoS_changing_handler (new_latency|
new_coverage))
```

The above function is used to inform ATES that an operation of publishing the event (`event`) is required. The *QoS* requirement can be specified with an associated `mode` to indicate whether coverage stability or coverage awareness is the preferred operation mode. If choosing the coverage stability mode, `coverage` must be specified by the calling sentient object. On the other hand, in the coverage awareness mode, `bound` should be indicated instead. To be informed in a timely manner of timing failures and *QoS* changes, `on_timing_failure_handler` and `on_QoS_changing_handler` handlers can be defined. The former is executed when timing failures occur. The latter will be called by ATES upon detecting *QoS* changes; a new value for the bound or the coverage (depending on the adaptation mode specified) will also be sent by ATES to the application object. In order to prevent spurious notifications due to irrelevant changes on the available *QoS*, applications must specify a threshold value (dev) that indicates the necessary deviation relative to the current value (of the coverage or bound) that triggers the delivery of a notification.

After executing the operation requestPublishing, a sentient object may start publishing events using the following function:

```
← publishEvent (event)
```

Event subscription requests are performed in ATES using an interface function similar to those existing in traditional publish/subscribe frameworks:

```
← subscribeEvent (event, subscriptionHandler (event))
```

The parameter subscriptionHandler indicates an handler that will be executed by ATES when events of type event are received from the event based communication subsystem. After the execution of this handler, the timed action associated to the transmission of this event is terminated.

4 Definition of Services and Protocols

This section describes the services and protocols defined in the cortex architecture, which may be used to implement the interfaces identified in Section 3. This includes a description of protocols and services required to implement the TCB API, an example of the use of the resource and task models and, finally, a description of communication services and protocols. The specific implementation details relative to other services, for instance ATES, will be provided in the final deliverable relative to basic services and protocols.

The protocols and services described in Sections 4.5 and 4.4 continue the work specified in WP2-D3 and WP3-D4. The goal is the interaction of islands of tight control over a wireless channels following the WAN-of-CANs model in CORTEX. So far, we provided a distributed event channel protocol implementation which was mapped to the CAN-Bus [28] and to a TCP/IP network. The work described in the two contributions: “A Real-Time Event Channel Model for the CAN-Bus” and “Content and Cell based Predictive Routing (CCPR) Protocol for Mobile Ad Hoc Networks” extends this previous approach and solves some of the deficiencies revealed.

The “Real-Time Event Channel Model for the CAN-Bus” introduces typed events and event channel classes with different temporal guarantees. It therefore extends the priority-based best effort approach of the existing protocol with respect to predictability and is a step to include the CORTEX event model on the CAN level. The approach exploits the already existing services and protocols like the subject-based addressing and binding mechanism and the automatic configuration capability. The paper shows how the different classes of real-time event channels are specified and how they are mapped to the underlying CAN-Bus network. To our knowledge, this is the first attempt to provide typed events and real-time event channel classes on the CAN level by a fully distributed architecture. Other approaches are either based on an event server model [34, 30] or have lower level abstractions [59].

To support communication between the CANs, our existing protocol builds event channels on top of TCP/IP [16, 44]. It uses a central server to perform the channel-to-address binding in a subject-based addressing scheme. Although this is useful in a single wireless LAN segment with a known set of nodes (e.g. a small number of cooperating robots), the scheme has problems in a larger and more dynamic setting:

1. The central server scheme does not scale well for larger sets.

2. A publisher may have to maintain a large number of open TCP connections.

3. Mobility of nodes is not supported.

These deficiencies are tackled in the “Content and Cell based Predictive Routing (CCPR) Protocol for Mobile Ad Hoc Networks”. CCPR includes the following services:

1) **Source (and route) discovery.** Based on the event channel subjects, publishers as sources of information are dynamically discovered by the protocol. A request issued by a subscriber to an event channel uses directional confined

flooding to find nodes publishing to the respective channel. Location and direction information is exploited as well as the hop number to define constraints on flooding. No central broker is needed to perform the subject-to-address binding.

2) **Cell-based route construction.** CCPR exploits the geo-cells provided by the TBMAC MAC-layer protocol described in WP3-D4 (and further elaborated in this deliverable) to construct a route. This has the advantage that the routes are stable even if individual nodes move. The cell-based routing thus takes the advantages of cluster-based routing but omits its disadvantages.

3) **Cell-based predictive route maintenance.** As long as cells along the route are populated, a subscriber is connected to the respective publisher(s). However, because of mobility a cell may become empty. This situation is proactively discovered and a search for an alternative route is initiated. If possible, a locally restricted solution for an alternative route is selected. CCPR exploits the predictability provided by the TBMAC layer. Because the latency of message transfer in a cell is bound and the number of hops is specified in the request package, a predictable latency of the entire route can be assumed if all intermediate cells are populated. Because of the location aware predictive routing a situation when a cell becomes empty can early be detected and either an alternative route can be provided or an early warning of a QoS violation can be issued.

4.1 TCB services and protocols

In this section we will describe the protocols and internal details of the services identified in Figure 2.

4.1.1 Timestamping Service

The timestamping module of the TCB implements the function `getTimestamp`. The timestamps are obtained by reading the local clock *pc* of the TCB component.

Figure 12 illustrates an interaction resulting from an invocation of the timestamping service. At real time instant t_a the application invokes the TCB to get a timestamp. Observe that this invocation is processed by the TCB only at instant $t_s > t_a$. At instant t_s, the service reads a timestamp $T_s = pc(t_s)$ from the local clock and sends it to the application. Finally, at instant t_b, the application obtains the timestamp and continues its execution.

4.1.2 Local Measurement Service

The local measurement service is used to measure time intervals comprised between two local events, that is, events that occur on the same machine. The local measurement module implements this service and provides it to applications through the functions `startLocalMeasurement` and `endLocalMeasurement`.

Observe Figure 13. At real time instant t_a the application invokes a local measurement operation to the TCB providing a timestamp that signals the start of

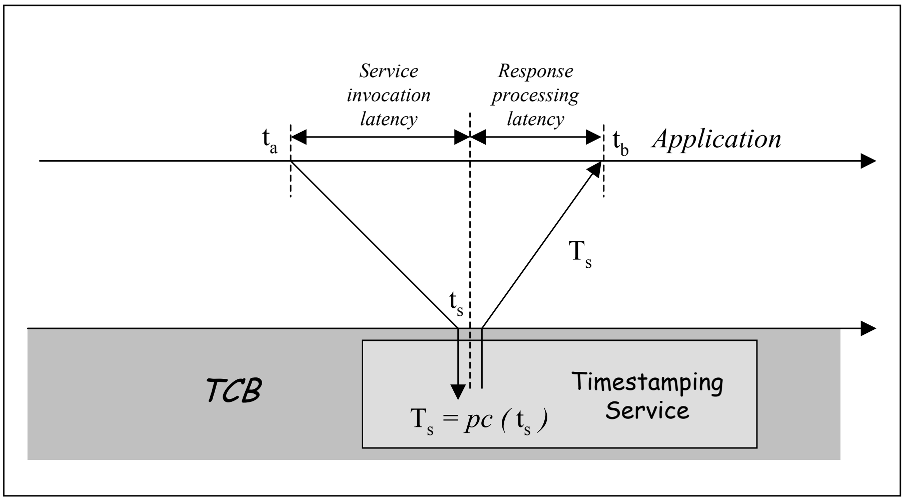


Figure 12: Timestamping service in action.

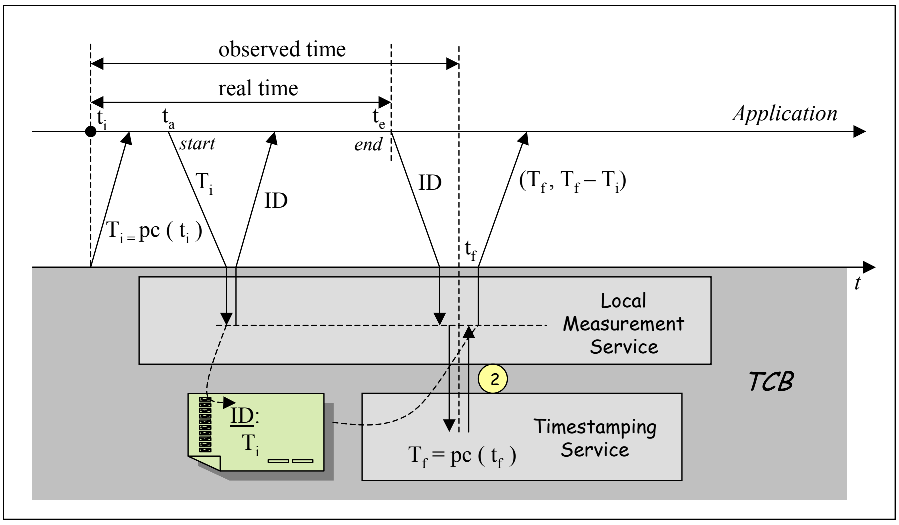


Figure 13: Local measurement service in action.

the interval ($T_i = pc(t_i)$)[2]. The TCB records the request and saves the timestamp provided by the application. The operation is referenced by ID. At instant t_e the application stops the measurement invoking the function `endLocalMeasurement` with the identifier (ID) of the operation. The TCB calls the timestamping service to get a timestamp that signals the end of the measured interval (represented by t_f in the figure). The duration is calculated by subtracting T_i to T_f and is sent to the application together with the timestamp T_f.

The errors associated to the measurements performed by this service are bounded by the following value: $error = (T_f - T_i)\rho + g$, where ρ is the drift of the local clock (**TCB Ps 2**) and g its granularity. Recall the informal definition of the duration measurement service provided in Section 2.1. Considering $T_{D_{min}} = g$ and $T_{D^2_{max}} = \rho$, the service presented in this section verifies property **TCB 3**.

[2] This timestamp must be previously obtained, for instance be calling the TCB timestamping service or using a timestamp provided as a result of a previous invocation of another TCB service.

4.1.3 Distributed Measurement Service

A distributed duration measurement requires mechanisms more elaborated than the simple subtraction of the timestamps required for the local measurement. In this service, the handled actions are bounded by events that may occur in distinct system nodes and therefore marked with timestamps provided by different physical clocks (which are not synchronized).

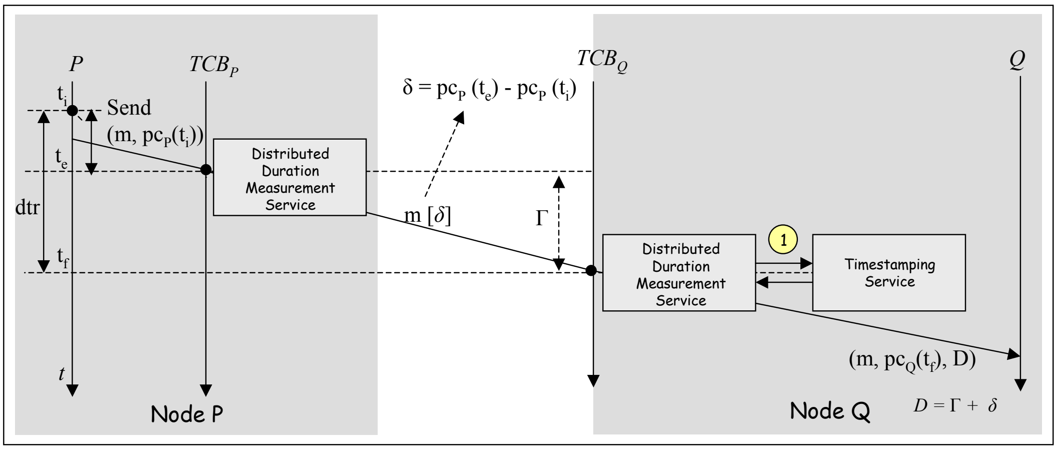


Figure 14: Distributed measurement service in action.

In Figure 14 it is possible to observe the operation of the distributed measurement service. At an arbitrary instant, an application executing in node P sends a message m through the payload communication channel to a process that is being executed in node Q, using for that purpose the distributed measurement service of the TCB (`sendMessage`) and specifying a start instant for the interval under observation (i.e. $PC_P(t_i)$). The duration measurement service of the TCB executing on node P receives the request for the message transmission at instant t_e (in fact, the TCB just performs an interception of the transmission, since the message will be handed over to the normal system transmission layers). Then it calculates the elapsed time since t_i until the instant at which the message interception was performed, that is, $\delta = PC_P(t_e) - PC_P(t_i)$, and re-sends the message (attaching the value of δ to it) to node Q.

When message m arrives to station Q, the duration measurement service of the TCB component executing on node Q intercepts it in order to establish a termination instant for the action and estimate its duration. The timestamping service is called to obtain a termination instant for the execution. Then the TCB estimates the duration of the action bounded by t_i and t_f, that is, $D = \delta + \Gamma$, and forwards the message to the application with an indication of $PC_Q(t_t)$ and D. Observe that to calculate (or estimate) D the service needs to know Γ, that is, the delivery delay of message m. The δ variable is received jointly with the message.

Since it is not assumed that the local clocks of the TCB components are synchronized, Γ must be estimated using mechanisms that do not assume the existence of a global time reference. For this purpose it is possible to use an algorithm based on the round-trip measurement technique [10, 26].

The distributed duration measurement service estimates durations based on the two variables presented in Figure 14: δ and Γ. If a bound can be established on the errors introduced by each one of these two variables, than it is obvious that the sum of the two variables will also have a bounded error. Observe again Figure 14: the variable δ represents a local measure $(PC_P(t_e) - PC_P(t_i))$, therefore the error implied in this measure is introduced by the drift and granularity of the local clock of TCB_P. Ignoring the clock granularity g, δ is calculated by P with an error bounded by $(PC_P(t_e) - PC_P(t_i))\rho$. As mentioned above, the delivery delay Γ can be estimated using a round-trip measurement technique with known and bounded errors.

Given all the above, it can be concluded that the distributed duration measurement service just presented satisfies property **TCB 3**.

4.1.4 Timely Execution Service

The timely execution module of the TCB implements the function `startExecution`. In general terms, the service allows the timely execution of functions provided by user applications. The functions must be accessible to the TCB, that is, must be in an address space that allows the TCB to execute them. The worst case execution time (WCET) of the functions must be indicated to the TCB along with the function definition.

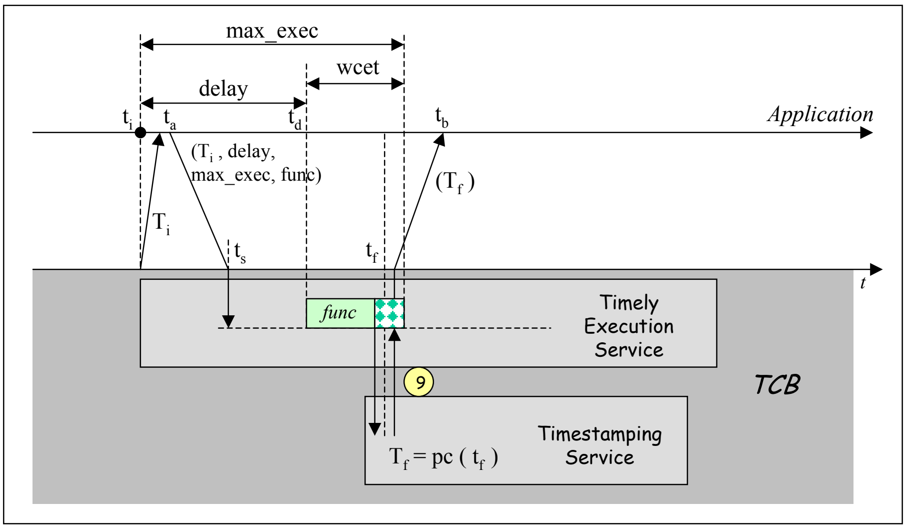


Figure 15: Timely execution service in action.

Figure 15 illustrates the service operation in a scenario in which a request is accepted by the TCB. At real time instant t_a the application invokes the service specifying a deadline for the execution, which corresponds to `max_exec` units counted from a reference instant $T_i = pc(t_i)$. The execution has to be deferred to an instant defined by $T_i + delay$. At instant t_s the request is processed by the TCB and the execution of `func` is delayed until $t_d = t_i + delay$, being triggered at an instant no earlier than t_d. When the execution finishes, the service calls the timestamping service in order to obtain a final instant for the execution (represented in the figure by t_f) and returns T_f to the application. Observe that between t_a and t_b the application stays blocked, waiting for the conclusion of the execution.

However, as mentioned earlier in Section 3.1, to secure the property **TCB 1** (eager execution) some of the requests for timely executions have to be denied by the TCB. Therefore, an admission control layer at the TCB interface has to verify if the two following conditions are fulfilled:

- at instant t_s, when the TCB processes a request, the time that remains for the expiration of execution deadline, that is, $max\_exec - (T_s - T_i)$, must be equal or greater that the WCET of the function;
- at instant $t_d = t_i + delay$, the time that remains for the expiration of the deadline, that is, $max\_exec - (T_s - T_i)$, must be also equal or greater that the WCET of the function.

Although the previously enumerated conditions can be easily verified in runtime, there exists also another restriction related with the need to preserve the timing parameters of the active tasks inside the TCB. In other words, it is necessary to ensure the feasibility of the schedule of all TCB tasks (internal and corresponding to external requests). On-line schedulability analysis is therefore required, for which it is possible to resort to known existing solutions [49].

Given that local clocks have known and bounded drifts (**TCB Ps 2**), property **TCB 2** (eager execution) can secured by the service in a simple way. The service only triggers the function execution when the local clock exhibits a value equal or greater than the instant specified for starting the execution.

However, observe that the rate of drift of the local clock can introduce considerable imprecisions when positioning future events in the timeline, if the durations associated to these events are too long.

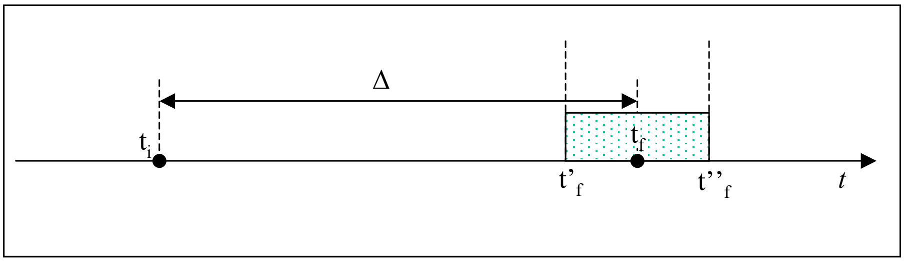


Figure 16: Imprecision of event positioning due to the clock rate of drift.

This imprecision can be clearly observed in Figure 16. Consider a real time instant t_f, specified by a duration Δ counted from a reference instant t_i, that is, $t_f = t_i + \Delta$. With a perfect local clock, we would have $pc(t_f) = pc(t_i) + \Delta$. However, when considering the clock rate of drift, at real time instant t_f the clock may exhibit any value in the interval $pc(t_f) \in [pc(t_i) + \Delta - \Delta\rho,\ pc(t_i) + \Delta + \Delta\rho]$.

Considering $pc(t_f)$ to be the instant specified by an application for starting an execution (measured on the local clock), the service will ensure property **TCB 2** if it only starts the timely execution at instant $pc(t_i) + \Delta + \Delta\rho$.

4.1.5 Local Timing Failure Detection Service

By using the local timing failure detection service an application will have the guarantee that the TCB will promptly execute an handler function (for instance to execute safety procedures) if the observed action does not finish before some specified deadline. In fact, the timing failure detection module of the TCB, which implements the functions `startLocalDetection` and `endLocalDetection`, allows the detection of timing failures on the execution of local computations.

Similarly to the local duration measurement service, the timing failure detection service provides one function to initiate a new detection (`startLocalDetection`) and another to finish it (`endLocalDetection`). In consequence, the service needs to internally store some information about the operations in execution. This mechanism is similar to the one used within the local duration measurement service.

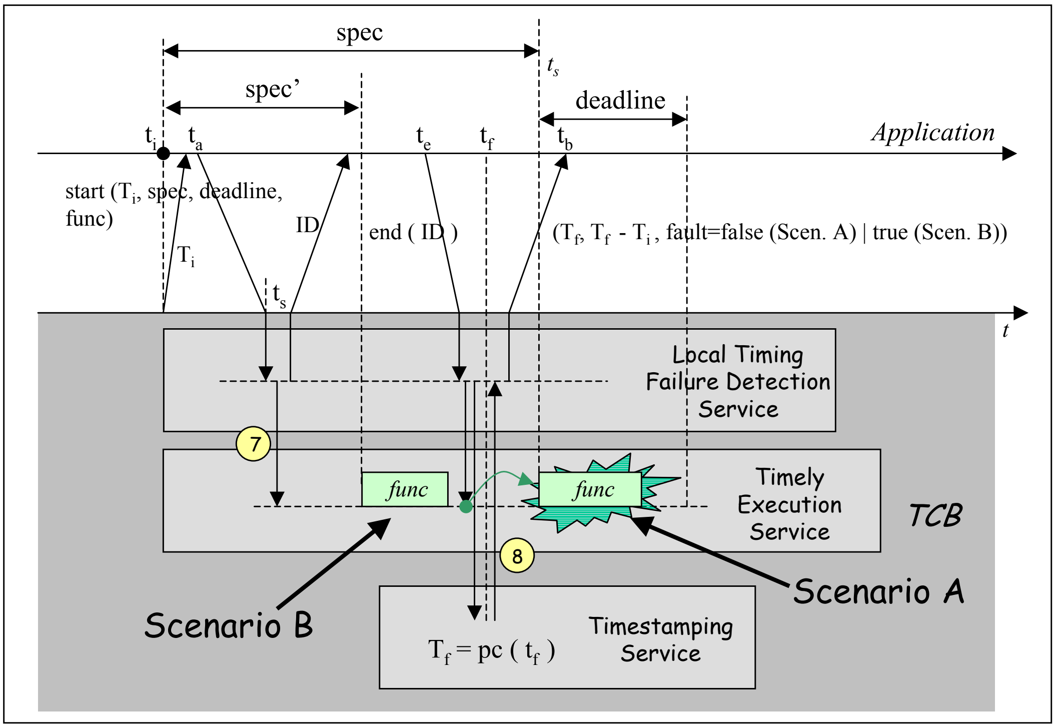


Figure 17: Local timing failure detection in action.

Figure 17 illustrates the service operation for two execution scenarios. In scenario A, the execution of an action observed by the TCB has terminated on time. In scenario B, the action does not finish on time, therefore doing a timing failure.

At real time instant t_a a timing failure detection request is invoked with the following parameters:

- $T_i = pc(t_i)$ - specifies the initial instant (measured on the local clock) for the observation;
- `spec` - specifies the duration allowed for the execution, counted from T_i (in scenario A);
- `spec'` - specifies the duration allowed for the execution, counted from T_i (in scenario B);
- `func` - indicates the function that should be timely executed if a timing failure occurs;
- `deadline` - the deadline for the execution of `func`.

In scenario A, a timing failure occurs at instant $T_i + spec$ (measured on the local clock). However, just like with the timely execution service, the drift of the clock must by taken into account in order to provide a timely timing failure detection. In concrete, it must be assumed that the clock runs at the lowest speed (i.e. has a drift of $-\rho$), which means that at real time instant $t_i + spec$ the clock will have drifted $-spec\rho$ towards the expected value. Hence, in order to provide a timely detection, $pc(t_i + spec - spec\rho)$ must be internally used as the bound for the detection (see Figure 18).

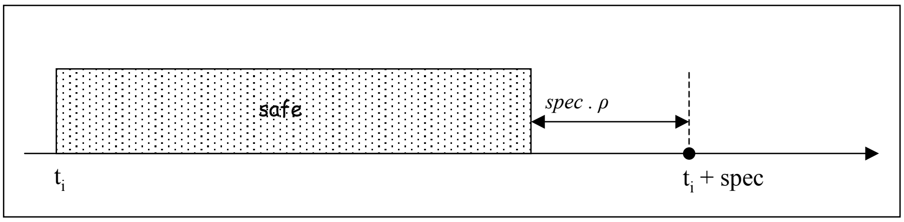


Figure 18: Guaranteeing timely timing failure detection.

At real time instant t_s the request is processed by the TCB, which checks if the execution has already failed (i.e. if $T_s > T_i + spec$[3]), in which case the request is rejected (in the two scenarios presented in the figure the service is accepted). The timely execution service is invoked to ensure the execution[4] of `func` at instant $T_i + spec$ (the timing failure instant) with a deadline received in `deadline`. At this point, unless the timely execution request is cancelled (e.g. if the action terminates on time), the function `func` will be executed by the timely execution service.

At instant t_e the application calls `endLocalDetection` in order to signal the end of the observed execution and to stop the failure detection operation. At instant t_f the TCB processes the request, asking the timestamping service for a timestamp to mark the end of the execution. In scenario A, provided that T_f is smaller than the deadline, meaning that no timing failure occurred, the TCB calls the timely execution service to cancel the execution of `func`. The value of T_f is returned to the application, as well as the duration $T_f - T_i$ and a boolean to indicate if the action was timely.

On the other hand, in scenario B, by t_f (when the TCB processes the `endLocalDetection` request) there has already been a failure (which occurred at $T_i + spec'$) and therefore `func` has already been executed.

In what follows we will show that the timing failure detection service just described preserves properties **TCB 4** and **TCB 5**.

Property **TCB 5** specifies the accuracy of the detector ($T_{TFD_{min}}$). As mentioned earlier, due the drift of the local clock it is necessary to assume that clocks are slow in order to ensure that timing failure detection is done correctly, that is, completely and timely. However, when the clock is perfect and a drift of $-\rho$ is assumed, the detection deadline will be anticipated by $spec\rho$ time units. This imprecision provides an upper bound for the accuracy of the detector, hence we will have $T_{TFD_{min}} = spec\rho$.

Property **TCB 4** imposes a bound on the latency of the detector ($T_{TFD_{max}}$).

[3] The terms resulting from the influence of the clock drift rate are omitted for the sake of simplicity.

[4] Recall that this execution may be refused by the timely execution service.

Provided that the timing failure detection service recursively uses the timely execution service, this bound can be immediately derived from property **TCB 2** (eager execution), and we will have $T_{TFD_{max}} = deadline$.

4.1.6 Communication Module and the Control Channel

In essence, the communication module must be implemented in a manner that satisfies property **TCB Ps 3**. Therefore, since this module operates over the TCB control channel, it is obviously necessary to use a control channel with real-time properties. The protocols used to transmit and receive messages must then preserve these real-time properties.

The module provides a well-defined interface to upper modules, namely to the distributed timing failure detection service. Basically, the following two operations are defined in this interface:

```
                                      ← send (message)
message, Tevt_reception, delivery_delay ← receive ()
```

The function `send` is used to disseminate a `message` on the control channel, addressed to all other TCB components. The service ensures that the message is delivered on time, that is, at most in $T_{D^3_{max}}$ units (**TCB Ps 3**).

The function `receive` can be used to explicitly request a new received `message` from the communication module. Besides the message itself, the function returns a timestamp corresponding to the instant when the message was received by the module (`Tevt_reception`), as well as the estimated upper bound for the message delivery delay (`delivery_delay`). This upper bound is estimated using some internal measurement procedures, based on round-trip durations.

The communication module also provides a detector of remote nodes crashes (which is, in fact, a perfect crash failure detector, using the terminology of Chandra and Toueg [11]). This detector is used by the distributed timing failure detection protocol, which will be described in the next section, to deal with crashes of remote nodes.

In order to guarantee the desired properties for the services provided by this module, several parameters must be bounded a priori. This includes bounding the maximum number of TCB modules (hence communication modules) and bounding the total amount of information to be transmitted during a certain time period. Consequently, this restrictions will have to be taken into account when defining the protocols that use the communication module, as will be seen later on.

4.1.7 Distributed Timing Failure Detection Service

By using the distributed timing failure detection service, applications have guarantees that the TCB will promptly undertake safety procedures if the observed timed actions do not execute on time.

Although a distributed action is started in a singular process, it can nevertheless terminate in several receivers processes (for instance, if a message is broadcast to the network). The distributed computations observed by this service must always include at least one message transmission on the generic communication channel.

Therefore, a distributed action can always be associated to some message transmitted on the payload communication channel.

An operation can be started either by calling `sendMessageTFD` or `sendMessageWRemoteTFD`, depending on the desired behavior in terms of reaction to a timing failure. If the objective is to execute the timing failure handler on the sender side, then the first function must be used. Otherwise, to execute the handler on the receivers side, `sendMessageWRemoteTFD` must be used. A call to one of these functions does two things: it activates the necessary procedures internally to the TCB in order to detect the timing failure and it sends the message to the payload channel.

A distributed action may involve multiple processes, which means that the `receiveMessage` function may be called by several processes for the same message. However, a distributed action only finishes when a receiving process executes the `endDistAction` function. This function requires the specification of an `id`, obtained from a `receiveMessage` call, which identifies the specific distributed action that is being terminated.

Besides detecting timing failures in a timely manner, the service also provides information concerning the delays of observed executions. The function `waitInfo` is used to obtain this information. Moreover, it is guaranteed that all processes involved in a certain distributed action receive this information consistently.

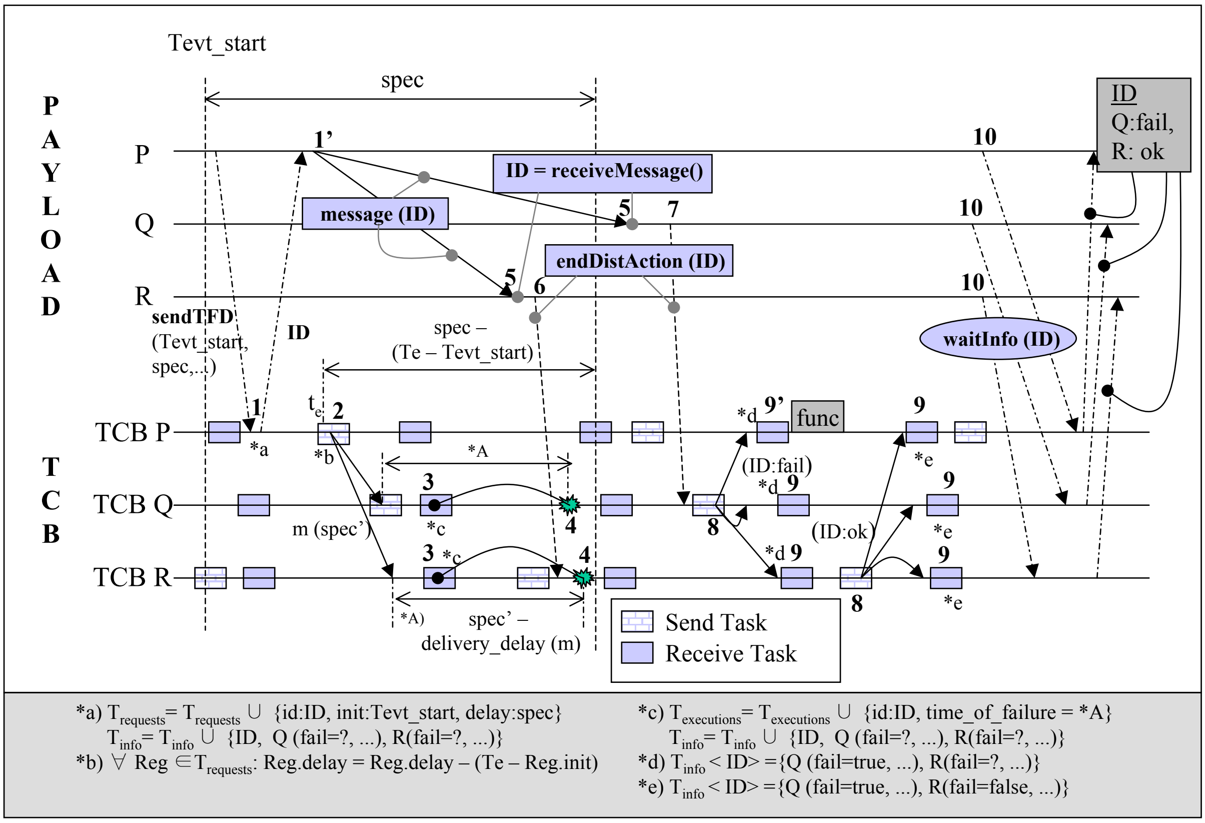
The algorithm that implements the distributed timing failure detection service operates in synchronous rounds through the execution of two periodic real-time tasks: the `read task` and the `send task`. The read task is responsible for processing messages received on the control channel and the send task disseminates information concerning timed actions. For that, they use the communication primitives (`send`, `receive`) supplied by the communication module.

Since the impact of physical clock drifts has already been discussed when explaining the local duration measurement service, and since we have shown that these effects can be taken into account deterministically, in the remainder of the text we will simply assume, for simplicity and without loosing the generality, that these effects can be treated at another level of abstraction. Hence we do not discuss them again.

In Figure 19 it is possible to observe a run of a distributed action executed using the timing failure detection service. The service is invoked by a process that executes in node P, and the message associated to the operation is addressed to two processes that execute in nodes Q and R. This example will be used in the discussion below, to illustrate the description of the algorithm that implements the timing failure detection service.

The execution of the timing failure detection protocol is done in several steps, and involves a few issues, namely:

- processing service invocations;
- dissemination of timing failure detection requests;
- processing timing failure detection requests;
- verification of the timeliness of executed actions;
- dissemination of failure detection results;



*a) $T_{requests} = T_{requests} \cup$ {id:ID, init:Tevt_start, delay:spec}
$T_{info} = T_{info} \cup$ {ID, Q (fail=?, ...), R(fail=?, ...)}
*b) $\forall$ Reg $\in T_{requests}$: Reg.delay = Reg.delay – (Te – Reg.init)

*c) $T_{executions} = T_{executions} \cup$ {id:ID, time_of_failure = *A}
$T_{info} = T_{info} \cup$ {ID, Q (fail=?, ...), R(fail=?, ...)}
*d) T_{info} < ID> ={Q (fail=true, ...), R(fail=?, ...)}
*e) T_{info} < ID> ={Q (fail=true, ...), R(fail=false, ...)}

Figure 19: Example of a distributed failure detection run.

- processing failure detection results;
- handling crash failures.

These issues will be discussed in the following sections.

Processing Service Invocations

A service invocation occurs when an application uses one of the supplied primitives (`sendMessageTFD` or `sendMessageWRemoteTFD`)

to request a message transmission with timing failure detection. In the scenario depicted in Figure 19 the service was invoked with the `sendMessageTFD` primitive, meaning that the timely detection should be performed only on the sender.

The initialization of an operation is performed in two stages. In the first one, a timing failure detection operation is requested to the TCB (point 1 in the figure), to which the TCB replies with a sequence number (ID) that uniquely identifies the started operation. In the second stage, the message associated to the action is relayed on the generic communication channel, along with the ID for the operation (point $1'$ in the figure).

The requests are stored in an internal table ($T_{Requests}$) until being disseminated on the control channel. This table records the delays (`spec`), counted from the reference instants `Tevt_start`, which define the instants at which timing failures occur (i.e., at instants `Tevt_start+spec`).

The state of the ongoing operations, that is, the result of each distributed execution (failure status, lateness degree, etc.) is stored in table T_{Info}.

For each active execution there is a active entry in T_{Info}, which will only be removed after the information it contains is delivered to the interested application.

Dissemination of Timing Failure Detection Requests

The dissemination of the timing failure detection requests is done by the `send task` (represented in the figure by 2). This task is executed at instant t_e, which will be used as a reference instant to detect timing failures of messages referenced in the $T_{Requests}$ table. Therefore, the relevant bound that must be observed for timing failure detection is no longer `spec` (counted from `Tevt_start`), but `spec'=spec-(Te-Tevt_start)`, since the time that has already elapsed since `Tevt_start` (`Te-Tevt_start`) will not be taken into account for the detection. This new bound `spec'` must therefore replace `spec` in every $T_{Requests}$ table record.

The content of $T_{Requests}$ is disseminated on the control channel with the previously calculated values (`spec'`). After the transmission the content of the table is deleted. Of course, in order to establish a detection bound in a remote site it will be necessary to measure the delivery delay of this control message, as will be seen below.

Processing Timing Failure Detection Requests

The requests for timing failure detection operations received from the control channel are processed by the protocol `read task` (represented in the figure by 3).

Given the specified delay included in each transmitted record (`spec'`), the value calculated for the control message delivery delay (`delivery_delay`) and the reception instant of this control message (`Tevt_reception`), the timing failure instants can be calculated at the receivers as follows:

`time_of_failure=Tevt_reception+spec'-delivery_delay`

Recall that `delivery_delay` is an upper bound for the real delivery delay, which means that the instants estimated for timing failures will always occur before the real timing failure instants. This ensures a safe detection and the preservation of the completeness property of the detector (**TCB 4**).

The estimation error of the control message delivery delay has an impact on the accuracy of the timing failure detector. However, given that this error can be bounded (see section 4.1.6) the accuracy property of the detector (**TCB 5**) can also be preserved. In fact, the detection accuracy ($T_{TFD_{min}}$) corresponds to the upper bound of the estimations of message deliver delays.

The requested operations are saved in table $T_{Executions}$. This table contains the information regarding all ongoing operations in that site, namely the estimated timing failure instants (`time_of_failure`) for those actions. There is also a record in the T_{Info} table which keeps the state of the execution.

Verification of the Timeliness of Executed Actions

After receiving a message from the payload channel (point 5 in the figure) using the function `receiveMessage`, an application will obtain the ID associated to that message. With that ID it will be able to terminate the distributed action, using the function `endDistAction`.

An execution is considered timely if it is terminated before the estimated timing failure instant (6 in the figure). Otherwise, if the action is terminated after the timing failure instant (7) the execution is considered faulty.

Dissemination of Failure Detection Results

The results of the observed executions are disseminated on the control channel by the `send task` (8). These results are only available when one of the following conditions is verified:

- a distributed action has finished on time;
- a distributed action has not yet finished, but when the `send task` executes a timing failure for that actions has already been detected.

The known results, which include measured durations, failure status and lateness degrees, are disseminated on the control channel and the corresponding information is removed from the $T_{Executions}$ table.

Processing Failure Detection Results

The results of the timing failure operations are processed by the `read task` (9). For each incoming result the information that reflects the state of its associated execution, presented in T_{Info}, is updated. This means that all information concerning the execution of a certain distributed action (lateness degree, failure status, etc.) will be available when all involved sites finally transmit the local result of that action.

In the scenario depicted in Figure 19, the service was invoked using the `sendMessageTFD` primitive, which means that any reaction to a timing failures has to be done at the sender side. When this primitive is used, the execution on the sender side is sightly different than on the other nodes. If a timing failure occurs in one of the receivers, the timely execution service is called on the sender side when it receives the first failure notification. Hence, a certain function `func` is promptly executed ($9'$). On the other hand, would `sendMessageWRemoteTFD` have been used instead of `sendMessageTFD` and the function would have been executed on node Q instead of node P.

When the results of a distributed action are all available, the TCB will be able to send them to the proper applications. To receive these results, applications must call the primitive `waitInfo` (10).

Handling Crash Failures

In the protocol described above, a TCB node waits for the completion of a distributed action before being able to provide information concerning that action to the application. However, because some nodes may crash (accordingly to the assumed failure model), it is necessary to include a mechanism to prevent a receiving TCB to wait forever for the missing information.

Fortunately this is a quite simple problem to solve, since the communication channel has synchronous properties. It suffices to construct a perfect crash failure detector (which is feasible in synchronous systems) which provides information about the nodes that are crashed. When a TCB is detected to be crashed, all the distributed actions involving that node are considered to have failed.

4.2 Example of the use of the resource and task models

As an example of resource reconfiguration consider a cooperating cars scenario in which cars are sentient objects capable of automatically moving on the roads. Cars communicate with each other by using a publish/subscribe middleware system. The resources system is partitioned into two different tasks: *drive car* and *Assisted Terrestrial Transportation (ATT).* The former is in charge of carrying out actions that allow cars to avoid colliding by detecting other car positions and obeying traffic lights. The latter is then responsible for receiving information related to traffic jams, weather forecast, and accordingly perform calculations for optimal routes. The task drive is associated with vtm$_{\texttt{drive}}$ whereas the task assisted terrestrial transportation is related to vtm$_{\texttt{ATT}}$. As an example of resource reconfiguration consider that the number of events generated in heavy traffic jams is considerably higher than in normal conditions. As a result the task vtm$_{\texttt{drive}}$ becomes overloaded. The system is using an EDF scheduling policy [49] which optimises the use of CPU up to 100%. However, during transient overload the algorithm clearly behaves unpredictable. The QoS manager of a car system entering the traffic jam zone has detected such an overload and has opted to change the policy to rate-monotonic [49]. This algorithm provides a lower bound of CPU utilisation than that offered by EDF but behaves better in overload conditions [54, 68]. Thus, the following operations are performed:

```
pIVtmFactory->getVtm('drive', &pIJob_vtm$_{drive}$ );
pIVtmFactory->getVtm('ATT', &pIJob_vtm$_{ATT}$ );
pIJob_vtm$_{drive}$->setSchedParam(schedParam$_{drive}$);
pIJob_vtm$_{ATT}$->setSchedParam(schedParam$_{ATT}$);
pIVtmFactory->getVtmScheduler(&pIScheduler_vtmSched);
pIScheduler_vtmSched->setPolicy('rate-monotonic');
```

Pointers to the interfaces IJob of the VTMs associated with each task are first obtained. This is followed by two operations in charge of setting the appropriate scheduling parameters to the VTMs. Afterwards, the VTM scheduler is accessed and its policy is changed.

Consider now the case whereby there are certain geographical areas in which cars experiment a shortage of network bandwidth, i.e. communication is difficult due to the obstruction caused by physical objects such as buildings, hills and trees. This situation is tackled by suspending non-critical tasks on behalf of critical ones. The tasks associated with the VTM with the lowest criticality would be suspended. The QoS control mechanism would proceed as follows, after finding out which is the lowest critical VTM, which happens to be vtm$_{\texttt{ATT}}$:

```
pIVtmFactory->getVtmScheduler(&pIScheduler_vtmSched);
pIScheduler_vtmSched->suspend(pIJob_vtm_ATT)
```

As a result, the VTM scheduler will suspend the corresponding VTM. The operation of suspending this VTM encompasses the suspension and liberation of their underlying resources, i.e. threads, network connections and memory.

4.3 TBMAC protocol messages and inaccessibility

4.3.1 TBMAC protocol messages

TBMAC uses a number of different protocol messages to manage the CFP slots in a cell. These messages include:

1. Data
2. Null
3. Slot Allocation Request
4. Slot Deallocation Request
5. Inter-cell OUT Slot Request
6. Inter-cell IN Slot Request
7. Inter-cell INOUT Slot Request
8. Inter-cell Deallocation Request
9. Slot Failure

When a slot in the CFP is allocated to one mobile host in the cell, a mobile host sends a Null message in its slot even if it does not have a message to send.

If a mobile host does not have a CFP slot allocated to it, it then transmits a Slot Allocation Request to other mobile hosts to allocate a CFP slot to it. This request results in an atomic broadcast being executed by the other mobile hosts in the cell. Information related to the atomic broadcast is included in the header of CFP packets. Upon completion of the atomic broadcast, the allocation of CFP slots is updated to include the new CFP slot allocated to the original mobile host. A CFP slot is deallocated in a similar way by atomically broadcasting a Slot Deallocation Request.

To allow TBMAC to manage the inter-cell slots, a number of different protocol messages are used. As described in 2.4.2, in each CFP, there is at least one static CFP slot allocated for communication from one cell to another. In Figure 20, cell A would have a static CFP slot allocated for communication from mobile hosts in cell A to mobile hosts in adjoining cell B. Cell A would also have a static CFP slot allocated for communication from mobile hosts in cell B to mobile hosts in cell A.

To allow dynamic inter-cell slots to be allocated and deallocated between two adjoining cells, then agreement between these two cells is needed. This agreement is achieved by adapting the atomic broadcast protocol so that the execution time of the atomic broadcast protocol is increased. This increase in the execution time reflects the increase in the number of cells (or the increase in the number of potential mobile hosts) where agreement is sought.

With this adapted atomic broadcast protocol in place, then allocating and deallocating inter-cell slots in the CFP is relatively simple. TBMAC has three different types of requests for inter-cell slots. These three types of requests reflect the different possibilities for communication between two adjoining cells. Referring again to Figure 20, the three different inter-cell requests are for a CFP slot for communication from cell A to cell B (OUT), cell B to cell A (IN) and two CFP slots for communication between cell A and cell B (INOUT).

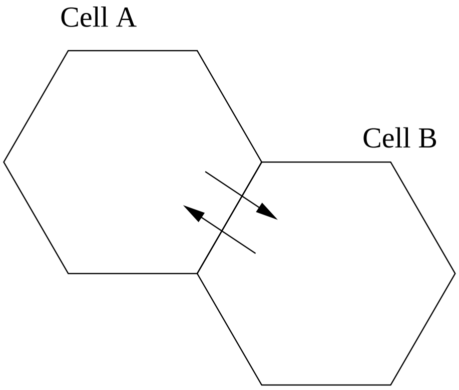


Figure 20: Two adjoining cells.

Finally, when a mobile host fails to use their allocated CFP slot (after a number of CFPs), then other mobile hosts transmit information about this failure by using a Slot Failure message. When a mobile host receives failure information about a slot from a majority of mobile hosts with CFP slots, then this mobile host atomically broadcasts a request to deallocate the failed CFP slot.

4.3.2 TBMAC protocol inaccessibility

In a computer system, inaccessibility [71] is when certain components are temporarily unresponsive without having actually failed. A classic example of inaccessibility in wired networking is token loss and recovery in a Token Ring network [65]. In TBMAC, an important aspect of inaccessibility occurs when a mobile host moves from one cell to another.

Time bounds (and associated probabilities) for this aspect of inaccessibility will now be derived in the following sections. Obviously, since the number of mobile hosts entering the cell is unknown a priori, this time bound is not certain (probability equal to 1) and therefore we derive the probability of this time bound being obtained.

4.3.2.1 Modelling arrival/departures We model the arrival and departure of mobile hosts into a cell using a self-service model (M/M/∞):(GD/∞/∞) (variation of a simple birth-death process [69]). Let λ be the arrival rate of mobile hosts into the cell and let μ be the service rate.

In our case, λ includes both mobile hosts arriving into the cell and mobile hosts powering on in the cell. Similarly, μ is the average time that a mobile host spends in the cell before leaving the cell or powering off. Note that μ would be related to the average speed of and the average distance travelled by mobile hosts in the cell and to the size of the cell.

Let $\rho = \lambda/\mu$, then the probability of k mobile hosts being in the cell in the time interval $[0, t]$ is given by

$$Prob(k\ hosts\ present) = \frac{(\rho t)^k}{k!} e^{-\rho t} \tag{1}$$

The time bound for an arriving mobile host acquiring a CFP slot depends on whether the cell, that the mobile host is entering, is empty or not. We will first consider the case of the cell being non-empty and then address the case when the cell is empty.

4.3.2.2 Non empty cell When a mobile host enters a non-empty cell, the worst case time bound, in the absences of collisions, occurs when a mobile host enters the cell just as the CFP begins. The mobile host must then wait until the end of this CFP and the following CP before listening at the beginning of the next CFP. The mobile host then broadcasts a request resulting in an atomic broadcast being transmitted that takes $2*(CFP+CP)$ to be delivered. Therefore the time bound before a mobile host has a CFP slot allocated to it equals $CFP+CP+CFP+CP+2*(CFP+CP)$.

By choosing a random slot in the CP, two or more mobile hosts could choose the same slot and therefore collide. We would now like to derive the probability of avoiding these collisions.

Let the number of slots in the CP be N_{CP}, then a mobile host randomly chooses a slot with probability $\frac{1}{N_{CP}}$. Given k mobile hosts arriving in the cell, the probability of these k mobile hosts choosing different slots equals

$$Prob(k\ distinct\ slots\ chosen) = \frac{(N_{CP})(N_{CP}-1)\cdots(N_{CP}-k)}{{N_{CP}}^k} = \frac{N_{CP}!}{(N_{CP}-k)!{N_{CP}}^k} \tag{2}$$

Therefore, combining probabilities (1) and (2) we get,

$$Prob(\ k\ hosts\ in\ cell\ and\ no\ collisions\ occur) = \\ Prob(\ k\ hosts\ present) * Prob(\ k\ distinct\ slots\ chosen)$$

Finally in this section, the probability that the time bound is obtained is equivalent to there being no collisions in the cell and is given by

$$Prob(\ time\ bound\ is\ obtained\ ) = \\ \sum_{k=1}^{\infty} Prob(\ k\ hosts\ in\ cell\ and\ no\ collisions\ occur\ ) \quad (3)$$

4.3.2.3 Empty cell Again, when a mobile host enters an empty cell, the worst case time bound, in the absence of collisions, occurs just as the CFP begins. The mobile host must then wait until the end of this CFP and the following CP before listening at the beginning of the next CFP.

On realising that no mobile hosts have been allocated slots in the CFP, the mobile host generates a list of random CFP slots (as described in [20]) and executes the collision detection steps described in Section 2.4.3 for a number of CFPs.

The time bound before a mobile host has a CFP slot allocated to it equals $CFP + CP + CFP + CP \quad (4)$. However, the mobile host will not have a consistent view of the allocation of CFP slots until M CFPs later. Therefore, the more important time bound is $CFP + CP + CFP + CP + M * (CFP + CP) \quad (5)$.

Now the probability of time bound (5) being obtained equals the probability of time bound (4) being obtained multiplied by the probability of the mobile hosts having a consistent view of the allocation of CFP slots after M CFPs.

We would now like to derive the probability of time bound (4) being obtained. Let N_{CFP} be the number of slots in the CFP and let j be the number of slots that each of the k mobile hosts generate.

Then, the probability of at least one of the j generated slots of each of the k mobile hosts not colliding with any other slot of the other mobile hosts is

$$\begin{gathered} Prob(\ All\ k\ hosts\ generate\ at\ least\ 1\ distinct\ slot\ ) \\ = j^k \tfrac{(N_{CFP})\cdots(N_{CFP}-k)}{{N_{CFP}}^k} \left(\tfrac{(N_{CFP}-k)\cdots(N_{CFP}-k-(j-2))}{{N_{CFP}}^{j-1}}\right)^k \\ = j^k \tfrac{N_{CFP}!}{(N_{CFP}-k)!{N_{CFP}}^k} \left(\tfrac{(N_{CFP}-k)!}{(N_{CFP}-k-(j-1))!{N_{CFP}}^{j-1}}\right)^k \quad (6) \end{gathered}$$

The second term of the above formula represents the probability of k mobile hosts choosing different slots. Once these k collision free slots have been allocated out of the available N_{CFP} slots, there are $N_{CFP} - k$ CFP slots remaining. The third term represents the probability of the k mobile hosts choosing their remaining $(j - 1)$ slots in the $N_{CFP} - k$ slots and thus not colliding with the first k slots.

The first term represents the number of possible ways the k mobile hosts can choose a slot, out of their list of size j, to be collision free. The first mobile host has j possible choices for its collision free slot the second mobile host also has j choices, etc.

Note that the above formula for the probability of k mobile hosts generating at least 1 collision free slot from a generated list of j slots corresponds to formula (2) when j is set to 1.

Similar to before, the probability of k mobile hosts being in an empty cell at time t and each of these k hosts generating at least one collision free slot is

$$\begin{gathered} Prob(\ k\ hosts\ present\ and\ each\ has\ a\ distinct\ slot\ ) = \\ Prob(\ k\ hosts\ present\ at\ time\ t\ ) * Prob(\ k\ hosts\ generate\ 1\ distinct\ slot\ ) \end{gathered}$$

The probability of the time bound being obtained in the empty cell is

$$\begin{gathered} Prob(\ time\ bound\ (4)\ being\ obtained\ ) = \\ \textstyle\sum_{k=1}^{\infty} Prob(\ k\ hosts\ in\ cell\ and\ each\ has\ a\ distinct\ slot\ )\ (7) \end{gathered}$$

In addition to the above probability, we would also like to calculate the probability of all the mobile hosts having a consistent view of the allocation of CFP slots after executing the collision detection steps for M CFPs.

Let P_{pkt} be the probability of a packet being corrupted. As in [24], let p_i be the probability of a mobile host, after the i^{th} CFP, remaining unaware of an update to the allocation of slots. Then, the probability, p_{i+1}, the mobile host remains unaware of this update after the $i + 1^{st}$ CFP is

$$p_{i+1} = p_i \left(P_{pkt}\right)^{k(1-p_i)} \quad (8)$$

where k, from before, is the number of mobile hosts in the cell. Note that $p_1 = P_{pkt}$ and p_M is the probability of the mobile hosts having a consistent view of the allocation of CFP slots after M CFPs. Note that p_{i+1} converges rapidly to 0 when P_{pkt} is small (less than 0.1).

The final term in formula (8) corresponds to the probability that the transmissions of mobile hosts, that know about the update, are all corrupted. This formula differs from the push anti-entropy in [24] due to the fact that communication is point-to-point in [24] while every mobile host in the TBMAC protocol processes the CFP header of a correctly received packet.

4.4 Content and Cell based Predictive Routing (CCPR) protocol for mobile Ad Hoc networks

This paper proposes a Content and Cell based Predictive Routing (CCPR) protocol to provide predictability in mobile ad hoc environments as envisaged in CORTEX. CCPR is based on main models and mechanisms developed in CORTEX. It exploits the subject-based publisher/subscriber model in the route discovery procedures and benefits from TBMAC's predictability properties. Exploiting TBMAC's cell structure for routing path construction instead of individual nodes provides a more reliable and predictable routing paths. Proactive and local route maintenance in CCPR can further improve the stability of a path in the presence of mobile entities.

4.4.1 Introduction

Mobile ad hoc networks are dynamically self-organized, rapid deployable networks without a fixed infrastructure or an access point [37]. Compared to wired networks, the mobile ad hoc networks have unique characteristics, such as the dynamic changing network topology, the variable link capacity, the scarce bandwidth and the constrained energy. In a mobile ad hoc network, nodes move arbitrarily, and due to their limited transmission range, nodes need to cooperate autonomously to construct multi-hop communication paths.

Because of the dynamic nature of the mobile ad hoc networks, it is a tough challenge to design robust, scalable and effective routing protocols which can adapt to the un-predictive and frequent network topology changes [66].

The publisher/subscriber model is an anonymous, scalable, inherently asynchronous communication paradigm and can quickly adapt to environment changes [3, 2, 31, 19]. The publisher/subscriber model has many benefits in dynamic cooperative applications compared to the traditional point-to-point communication, and it seems to be well suited for the mobile ad hoc networks [31, 35, 36]. In [2, 45], a subject based scheme of publisher/subscriber model is introduced, in which the content of a message is related to a unique subject identifier. A subject has one-to-one relationship with a logical event channel which disseminates information from the publishers to the subscribers. Therefore, in a multi-hop network, an event channel is made up of the routing paths from the publishers to the subscribers which are interested in the same kind of events.

The publisher/subscriber model in routing avoids the addressing problem. In mobile ad hoc networks, normally there is no central server from which the mobile nodes can query their network layer addresses [7]. The auto-configuration procedure in address-based ad-hoc networks incurs extra overhead in time and bandwidth and can not guarantee the address uniqueness when network partitions exist. Using the subject based routing protocols, nodes can join and leave a network autonomously. Additionally, the subject based routing protocol is suitable for anonymous many-to-many communication, in which the subjects represent different multicast groups. In contrast to this, explicit membership management is needed in an address based multicast routing protocol.

In mobile environments, it is difficult or even impossible to exploit the publisher/subscriber model with a central event broker as proposed in [44]. This will only work satisfactorily in a network restricted in size. Also, a scheme utilizing specific distributed brokers may suffer from significant maintenance overhead when the network topology changes rapidly. When utilizing the publisher/subscriber model in a mobile environment, a major challenge is to develop an efficient scheme which can take advantage of the merits of the publisher/subscriber model without incurring extra overhead.

With an increasing interest for mobile applications, active research work in recent years has developed a variety of routing protocols [6] for mobile ad hoc networks. But, most of the existing protocols work in "best effort" way, and can not provide support for safety-critical applications. Safety-critical applications, such as disaster search and rescue operations, cooperating autonomous robots, and traffic management are typical application scenarios of mobile ad hoc networks. One unique feature of safety-critical applications is the need of high predictability and reliability. Hence, providing network layer predictability and reliability is an important prerequisite. Compared to wired networks, reliability and predictability requirements impose additional challenges on protocol design in mobile ad hoc networks.

In order to improve network layer predictability and reliability, the following issues should be addressed in routing protocol design for mobile ad hoc networks:

1. A MAC layer protocol should reduce the time uncertainty of the access to wireless communication channels and support resource reservation.

2. The time uncertainty introduced by the nodes which may cause single point of failures, e.g. the unpredictable re-election procedures of the clusterheads in a cluster based routing protocol, should be omitted.

3. The routing protocol should choose the routing paths which can provide the predictability and reliability during the routing path construction and maintenance.

4. A wireless link has constantly changing capacity and may disconnect unpredictably. Hence, it is necessary to provide redundant routing information and construct robust communication paths.

5. Frequent routing path rediscovery and maintenance procedures following path disconnection may cause dissemination of a large mount of control packets and introduce time uncertainty. Therefore, a mechanism is needed to extend the life-span of the routing information and decrease the frequency of route re-discovery and maintenance operations.

6. It is obvious that the local path recovery or maintenance operation provides a better predictability than constructing a new path from scratch. Hence, a global path recovery should be used only when a local approach does not work.

In this paper, we propose the Content and Cell based Predictive Routing protocol (CCPR) in order to improve the routing predictability and reliability in mobile ad hoc networks. Support for QoS needs to be provided at different system levels. Without loss of generality, we assume that there is a middleware layer above CCPR, which provides the subject of messages and the timeliness, proximity or reliability requirements specifications for the communication. TBMAC, a predictive MAC protocol, is selected as the MAC layer protocol of CCPR. The remainder of this text is organized as follows. Section 4.4.2 gives a short introduction for TBMAC. Section 4.4.3 provides the basic operations of CCPR. In section 4.4.4, we present the related work, and section 4.4.5 gives the conclusion and future work.

4.4.2 TBMAC

In general, it is hard to guarantee the network layer predictability and reliability without support from the MAC layer. This is even harder in a mobile ad hoc network. TBMAC is selected as the MAC layer protocol for CCPR because it provides a predictable access to the wireless medium for mobile nodes (see Section 3.4 and [17]).

We will only give a short introduction of TBMAC here. In TBMAC, we assume that every node can assess its geographical position. The entire geographical area is statically divided into a number of cells. The size of the cells is related to the transmission range of the radio equipment, i.e., a cell and all of its adjacent cells are within the transmission range. Orthogonal spreading codes are allocated to cells to omit collisions and the hidden terminal problem.

For intra-cell communication, the access to the medium is divided into two time periods, the Contention Free Period (CFP) and the Contention Period (CP). The additional Inter-cell Communication Period (ICP) is used for the message exchanges between adjacent cells. Each period has a well-known duration, and is divided into time slots. A communication round comprises one CFP, one CP and one ICP. The communication rounds are synchronous in all cells. Figure 21 illustrates the structure of a communication round. In TBMAC, the protocol to allocate and deallocate time slots is similar with the PCF (Point Coordination Function) in IEEE 802.11 standard, but it is fully distributed and has no dependence on the access points. The Synchronous Atomic Broadcast protocol [18] is exploited to achieve consistent agreement.

To facilitate the inter-cell communication, we made minor changes to the inter-cell time slots assignment scheme of TBMAC. As specified in TBMAC [17], when a node wants to send packets across cell boundaries, it issues a request for time slot assignment in the ICP. After the requested time slots being allocated to the node, it begins its inter-cell transmission. This scheme is flexible because within a cell, every node can apply for the time slots in the ICP when it wants to perform inter-cell communication. However, this scheme may cause packet collisions. Before a node can transmit packets to another cell, it needs to wait a certain time due to the atomic broadcast algorithm used for time slots allocation. Additionally, because the ICP time slots assignment is consistent only for nodes within a cell, collisions

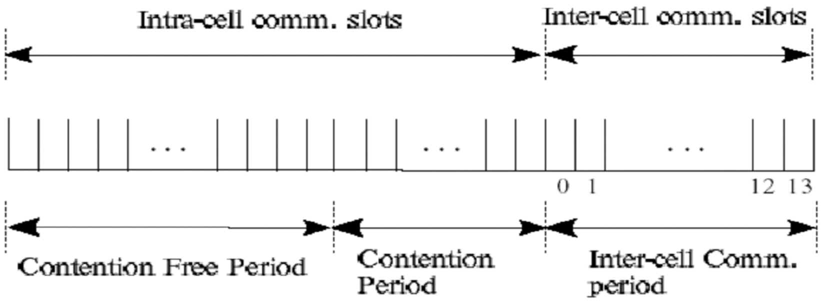


Figure 21: The structure of a communication round.

will happen when nodes in different cells use the same ICP time slot and transmit packets to the same cell. To avoid transmission collisions, adjacent cells should have coordinating time slots assignment in the ICP. Obviously, this coordination will introduce extra overhead and complexity. To deal with this problem, we statically assign time slots in ICP to each adjacent cell, and gateway nodes are dynamically elected to use these time slots and perform inter-cell communication. If a node wants to send a packet to an adjacent cell, it sends the packet to the gateway node which is responsible for the inter-cell communication with that cell. In a sparsely populated cell, a node may have the responsibility to communicate with multiple adjacent cells. We propose a gateway node election scheme which extends the time slots allocation and de-allocation requests of TBMAC by including an extra field which represents the power level or computing resource level information of the sending node. After a node moves in or out, these requests will be atomically broadcasted, and all nodes in a cell will receive the respective information. A node can decide if it is a gateway node or not according to this consistent information. Because the procedure is combined with the time slots allocation and de-allocation procedures, it adapts to the membership changes of cells and incurs very little overhead.

4.4.3 Content and Cell based Predictive Routing (CCPR) protocol

In CCPR, nodes are organized based on the geographically divided cells as in TB-MAC. To improve the robustness of the routing paths, cells are used to construct routing paths instead of individual nodes. Therefore, mobile nodes within a cell need to maintain some consistent information.

Obviously, in CCPR, nodes in a cell should maintain the same routing information. So, the movement of a single node and the disconnection of a single link will not result in a complete routing path disconnection. Consequently, the hop number is calculated using the number of cells a packet traverses along the routing path, and not in a node-to-node style. When a routing information change occurs, all the nodes within an related cell update their routing tables according to the change. Because of the shared routing information, each node in a cell has the capability to act as a gateway node. The consistency of the routing information in a cell is guaranteed by the atomic broadcast.

In CCPR, the cells are addressed to reflect their geographical relationship. Because the cell structure is static, the number of hops between any two nodes can be pre-calculated and remains unchanged as long as the respective cells are populated. With the predictive MAC layer protocol and resource reservations, the time needed for packet transmission from one cell to another can be calculated from the number of cells it will traverse.

In a publisher/subscriber protocol, the subscriber subscribes to a certain subject of information rather than to an explicit publisher. Among the many possibilities to solve this problem, we propose a dynamic discovery protocol which we think is the most appropriate way in a network lacking any infrastructure and which is exhibiting a high degree of mobility. The discovery protocol is supported by Cell Resource Tables (CRT), which maintain information about the publishers and the respective subjects provided in a cell. A CRT is available in each node in the cell and the consistency of the information is maintained by the respective update protocol. Once discovered, the discovery protocol binds the subject requested by some subscriber to the network identifiers of a node hosting the respective publisher. Because this is a local activity of the cell, it supports fast adaptation if a publisher moves and therefore, reflects the needs in a mobile environment. During the information discovery procedure, every node within a cell can decide on the basis of the CRT whether a request can be satisfied by this cell. A CRT entry includes the subject (channel ID) and the IDs of nodes which host publishers belonging to the channel. When the membership of a cell changes, the content of the CRT will be updated accordingly. This update procedure can be combined with the time slot allocation and de-allocation operations when a node moves in or out of a cell.

CCPR has three phases, namely, the route discovery phase, the routing path construction phase and the route maintenance phase. In the following subsections, we give description of these phases.

4.4.3.1 The route discovery phase To reflect the dynamic nature of the mobile ad hoc network, a subscriber floods discovery requests to the network in order to search for the publishers providing the requested subject. The search area information, such as the maximum hop number, the proximity, the direction or cell addresses restricts the flooding of the discovery requests. Therefore, the route discovery procedure in CCPR is tightly coupled with the resource discovery of the publisher/subscriber model. After a publisher is discovered, a routing path construction procedure is initiated to connect a publisher with the requesting subscriber. A similar scheme is used in Directed Diffusion [39] exploited in the sensor networks.

In the route discovery phase, firstly, a route discovery (RDIS) is issued by a subscriber. The RDIS packet includes the cell address of the subscriber node, the subject, the sequence number, the maximum hop number, the search area, and the current hop number. The structure of the RDIS is given in Table 2. The sequence number is used to distinguish different RDIS packets issued by the same subscriber. The search area gives the constrained flooding area of the RDIS packets.

Requesting Cell Address	Subject	Maximum Hop Number	Current Hop Number	QoS attribute	Search Area	Sequence Number

Table 2: The route discovery packet (RDIS) packet.

According to the cell division method used in CCPR, all gateway nodes of the subscriber residing cell will receive the RDIS. After receiving an RDIS packet, a gateway node checks the subject field and sequence number field of the packet. If the same packet has been received recently, the packet is dropped. Otherwise, it checks the search area field of the RDIS, increases the current hop number field in the RDIS by one, and selectively forwarding RDIS to the adjacent cells. The same operation will continue until the search area is covered or the current hop number equals the maximum hop number. In Appendix A, the flow chart of the route discovery algorithm is given.

Flooding is a crucial limitation for the scalability of routing protocols. In CCPR, only the gateway nodes are involved in the route discovery procedure. Moreover, the maximum hop number and the search area constraints effectively reduce the flooding overhead.

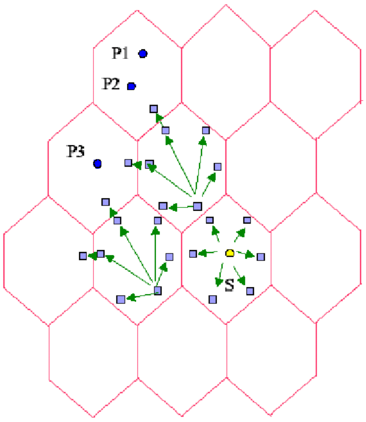


Figure 22: The flooding of the route discovery packets.

Figure 22 depicts a simple example to illustrate the route discovery procedure. A subscriber S issues the RDIS packets which have the maximum hop number of 2 and the direction information is northwest. After the route discovery procedure, publishers P1, P2 and P3 are found within the search area. Publishers which are out of the flooding area will not be discovered because of the maximum hop number and the search area constraint.

4.4.3.2 The routing path construction phase After receiving a new RDIS request, a gateway node checks the Cell Resource Table (CRT). If the CRT includes the same channel information as the subject field in the RDIS, the gateway node checks the routing table. In CCPR, a routing table entry includes the subject of an event channel, the destination cell address, the maximum hop number to the destination cell, the current hop number to the destination cell, the next hop cell address, and the Time-To-Live (TTL) field. The maximum hop number is defined according to the channel's temporal requirement. If a routing table entry is not updated within TTL time period, this entry will be assumed as obsolete and be deleted. The structure of a routing table entry is given in Table 3.

Subject	Dest. cell addr.	Source cell address	Maximum hop number to dest. cell	Current hop number to dest. cell	Previous hop cell addr.	Next hop cell addr.	TTL

Table 3: Routing table entry

If there is no routing table entry for the subject and the requesting cell, a new path has to be constructed. To do this, the gateway node forwards a route reply (RREP) packet back to the subscriber's cell. During this packet forwarding procedure, a routing path is constructed. The proper adjacent cells can be selected to construct the routing path according to the subscriber specified QoS requirements.

The structure of the RREP is given in Table 4. A RREP packet includes the cell address of the publisher node, the cell address of the requesting subscriber, the subject of the channel, the maximum hop number, the current hop number, the QoS attribute used for routing path construction and a time stamp.

Reply Cell Address	Requesting Cell Address	Subject	Max. Hop Number	Current Hop Number	QoS attribute	Time Stamp

Table 4: The Route REPly packet (RREP).

During this backward path construction, the QoS attribute field must be addressed. The geographical information obtained from cell addresses is used to assist this procedure. The gateway node selects the next hop cell based on the QoS attributes and the geographical information. The gateway node increases the current hop number field in the RREP packet by one and forwards the RREP packet to the selected next hop cell. Before that, it creates a new routing table entry using the subject and destination cell address fields in the RDIS packet. Then a routing table consistency procedure is initiated within the cell. The operation described above will continue until the RREP arrives the subscriber residing cell or the current hop number equals the maximum hop number. The algorithm of routing path construction is given in Appendix B.

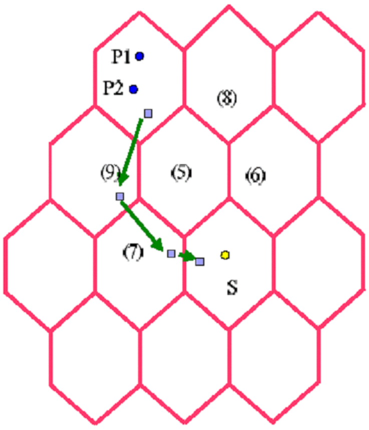


Figure 23: Routing path construction.

Metrics such as the bandwidth, the reliability or the shortest path can be used to select the next cell along the routing path. From the header of the MAC messages, nodes in a cell know the time slots assignment and node number information of its adjacent cells. Generally, this information can be used as criteria to indicate the bandwidth and reliability level of the adjacent cells. The densely populated cells are more reliable, but have less bandwidth resource than the sparsely populated cells. If the reliability is used as the metric for routing path construction, the node density information should be used to select the next hop cell to the requesting cell. Figure 23 shows the routing path construction procedure from the publishers (P1 and P2) residing cell to the subscriber (S) residing cell using node number as metric for routing path construction. In this figure, the squares represent gateway nodes. For simplification reason, only nodes which directly related to routing path construction are given.

4.4.3.3 The route maintenance phase The main challenges of the routing protocol design for the mobile ad hoc networks originate from the node mobility and the dynamic network topology changes. One advantage of CCPR is that the influence from the node mobility is reduced. Compared to the routing protocols which use individual nodes as routers, the cell based approach provides redundant routing information. Because all nodes within a cell maintain the same routing and channel information and all of them have the capability to act as gateway nodes. Only in the following three cases, node movement will initiate the route maintenance procedures:

Case 1: a publisher moves out;

Case 2: a subscriber moves out;

Case 3: an intermediate cell along the routing path becomes empty when the last node moves out.

The effect of node movement may be compensated by a local adaptation of the routing path. Thus, it may be possible to find a local solution to connect the moving node to the already established path. In this case, most of the existing path is still be used. The alternative would be a complete reconstruction of the path including the flooding of the network of the route discovery protocol. It is obvious, that the local solution is by far more efficient. It also can be performed proactively. If a node, which is aware of its position, speed and direction of movement, detects that it will soon move to a particular adjacent cell, it initiates a route maintenance protocol, which already prepares the local route adaptation. This leads to an improved predictability of communication because now, the latency of the path in number of hops can either be preserved or it changes at most by one hop. This sharply contrasts to the temporal uncertainty of a complete route reconstruction. Only in cases of a very sparsely populated environment, i.e. most of the adjacent cells are empty, a complete reconstruction will be inevitable. The following subsections describe the route maintenance operations in more detail for different cases pointed out above.

Dealing with the publisher movement When a node, hosting a publisher, detects that it will leave a cell soon, it sends a maintenance request to the adjacent cell which it has identified as target. A gateway node in the target cell then checks its routing information to determine whether the cell is already part of the path or not. If there already exists a routing table entry which can be used, there is not much to do because the cell already was part of the path. If there is no such routing table entry for this publisher, the gateway node must create a new routing table entry to connect the cell of which the publisher is moving out. In fact, this operation constructs a local routing path segment and connects it to the existing routing path. The changes of the route path, such as the hop number change and the source cell change, need to be sent explicitly along the routing path or be piggybacked with the next data packet to the cells along the routing path and the subscriber's cell.

Dealing with the subscriber movement A similar approach is used for a subscriber movement. Before a subscriber moves across a cell boundary, it proactively sends a route maintenance request to the cell it moves in. The request includes the routing information currently used by the subscriber. As in 4.4.3.3, the maintenance operation is confined to the cell which the subscriber is leaving and the target cell. The routing path update message will be sent explicitly to the nodes along the routing paths of the event channel.

Dealing with the empty intermediate cells The mobility of the nodes may lead to the situation that an intermediate cell along a multi-hop path becomes empty. In CCPR, an alternate routing path proactively is constructed before the last node moves out of the cell. This requires two steps. Firstly, a node must be able to detect that it is the only node in a cell. This is achieved through the time slot assignment vector which always comprises all nodes in a cell. Let's call the cell with only one node inside an unstable cell. The second step is the local adaptation of the path which has an unstable cell. The node in the unstable cell tries to find an alternate routing path segment in the adjacent cells. In principle, this proceeds like the maintenance protocols described above. However, there are two differences which can be seen as optimizations for this special case. Firstly, because the last node in a cell acts as the gateway node for all adjacent cells, it knows their time slots assignment, the bandwidth and the reliability information. Secondly, there is not only one target cell as in the case above, but there are more options. Thus a route maintenance request is sent to all target cells which can provide enough QoS support for the routing path. After receiving the request, the nodes in the selected adjacent cells update their routing table entries respectively. As in the final part of the maintenance operation, the routing information in the unstable cell is deleted.

The full path route maintenance It is possible that an alternate routing path segment can not be found because the requirements of the subscriber, such as the maximum hop number, the bandwidth or the reliability can not be satisfied. In this case, the full path route maintenance is needed, and a new routing path construction procedure will be performed. As pointed out previously, this happens only in the case when no adequately populated cells are in reach. We assume that this will be a rare case.

4.4.3.4 Keeping the routing tables consistent When a mobile node is moving into a populated cell, it sends a routing table consistency request. After receiving the request, the first node which gets its time slots in CFP sends a route consistency update. The update comprises the routing information shared within the cell. The new comer updates its routing table using the information.

During the routing path construction phase, after a next hop cell being selected for a new routing path, a gateway node sends a routing table update. This update includes the routing table entry for the new routing path. During the route maintenance procedures, a routing table update will be sent to a cell for proactive and local maintenance purpose. This update may include the routing information used by a publisher moving in, a subscriber moving in or the node is an unstable cell. Because of the broadcast nature of the wireless channels, all nodes will receive these routing table updates. Nodes will change their routing tables according to these updates. The atomic broadcast is used to guarantee the consistency of routing information.

4.4.4 Related work

Many routing protocols have been proposed for the mobile ad hoc networks in recent years. The most commonly used classification method divides routing protocols for mobile ad hoc networks into the proactive routing, the reactive routing and the hybrid routing according to the approaches used to acquire and maintain routing information. The proactive routing protocols proposed for mobile ad hoc networks were adopted from widely used approaches for wired networks with modifications adapting to the features of mobile ad hoc networks. For proactive routing protocols, nodes need to maintain routing information for all reachable nodes in the network by continuously exchange topology or link information. The Destination Sequenced Distance-Vector routing protocol (DSDV) [61] exploits the sequence number to distinguish stale routes from new ones and thus avoid the formation of loops. The Adaptive Distance Vector Routing Algorithm [5] adapts to the changing network load and node mobility by varying the size and frequency of updates dynamically. The Optimized Link State Routing protocol (OLSR) [40] uses link state routing approach with modification addressing the convergence problem.

With the presence of node mobility and frequent network topology changes, large amount of control packets are disseminated to maintain the updated routing information for the proactive routing protocols. Therefore, the proactive routing protocols are suitable for the small networks or networks with slow node movement and network topology change. The advantage of proactive routing protocols is, when a node wants to send packets it can use the already existing routing information immediately. Furthermore, the link state based proactive routing protocols can be exploited to support QoS routing [??].

In an reactive routing protocol, nodes do not need to maintain the up-to-date topology information of the whole network. A routing path is constructed only when it is needed, and normally a route query flooding is exploited to find the destination node. The Ad hoc On-demand Distance Vector Routing (AODV) [62] is an reactive routing protocol based on the similar mechanism with DSDV algorithm, and it minimizes the number of required broadcasts by creating routes on demand. For Dynamic Source Routing Protocol [21], every packet has a list of all the hops the packet will traverse on its way to the destination. A node maintains route caches containing the source routes that it is aware of. The Temporally Ordered Routing Algorithm (TORA) [60] is based on the link reverse algorithm, and limits the propagation of control information about the topology change only to a small number nodes near the event.

The main advantage of the reactive routing protocols is they have better scalability compared to the proactive routing protocols because nodes only store the routing information about the active routing paths. Although the reactive routing protocols omit the overhead for maintaining the whole view of the network topology, they also have intrinsic shortcomings. Firstly, when utilizing reactive routing protocols, a node may wait for a long time for route discovery and routing path construction before it can really send data packets. This substantially will affect predictability. Additionally, the flooding operation in route discovery procedure may incur heavy traffic overhead.

An hybrid routing protocol tries to incorporate aspects from proactive routing and reactive routing. The Zone Routing Protocol (ZRP) divides the network into several routing zones [33]. The proactive Intra-zone Routing protocol (IARP) is used inside the routing zones and the reactive Inter-zone Routing Protocol (IERP) is used between the routing zones, respectively.

From the view point of the hierarchy, most of the routing protocols mentioned above can be thought as flat routing protocols, in which nodes have the same function and importance and routing paths are constructed on the basis of individual nodes. In flat routing, because of the unpredictable node mobility and the variable link state, a single node mobility and a single link breakage may cause a routing path disconnection. The frequent routing path re-discovery or maintenance procedures following the path disconnection may exacerbate the time uncertainty. Additionally, the flat routing approaches can not provide scalability for large networks. The cluster-based routing protocol are proposed to solve these problems.

The Clusterhead Gateway Switch Routing protocol (CGSR) [13] is a proactive hierarchical routing protocol in which nodes are grouped into clusters. Different spreading codes used across clusters and clusterhead controlled token protocol is used inside cluster. Clusterheads maintain shared routing tables similar to the DSDV. Packets are forwarded in two steps, firstly from one clusterhead to another clusterhead, then from clusterhead to the destination node. Gateways are used to forward packets between two clusterheads. In CGSR, the clusterhead-token infrastructure is exploited to provide QoS support.

For the Cluster Based Routing Protocol (CBRP) [42], a clusterhead is elected to manage the membership information of the nodes inside each cluster. Inside a cluster, membership information is used for route discovery. Request flooding is used to find routes between clusters in the same way as in a reactive routing protocol.

The cluster based routing protocols can improve the reliability by selecting the stable nodes as clusterheads, but they can not provide satisfying predictability guarantees. Firstly, for a cluster based routing protocol, the movement of a clusterhead may cause the unpredictable cluster merging and splitting. Secondly, the temporal property of the clustering algorithms not only depends on their computing complexity, but also the context information, such as the number of nodes and clusters in the neighborhood. Additionally, the clusterheads or the gateway nodes introduce single point of failure, and normally they suffer from heavy traffic overhead and need more computing resources.

With geographical information obtained from GPS or similar equipment, the location based routing protocols make routing decisions according to the position relationship between the packet forwarding node and the destination node and the mobility feature of the nodes. In [47], the Location-Aided Routing (LAR) proposes using position information to improve the route discovery procedure of reactive routing protocols. The Distance Routing Effect Algorithm for Mobility (DREAM) [4] also use the location and mobility information for routing, but the location information is used to constrain the data packets flooding into a small region. The existing location based routing schemes reduce the amount of control packets for the routing path discovery, but do not provide predictability and reliability supports.

In addition to the topology based routing and the location based routing, some content based routing schemes are proposed for the mobile ad hoc networks. In [72], the Content Based Multicast (CBM) routing model is proposed for military and emergency applications. The content of the multicast data determines the receiver set for the data in CBM. Source nodes using push protocol broadcast warning message to a certain push area. The pull protocol is used by the receivers to get information from the block leader of expected area it will be after certain time period. The block leaders are needed to maintain the membership of nodes in one area. In CBM, the timeliness and proximity are supported. But it still exploits a cluster like structure, and does not provide the reliability support. In [36], the problem to construct an explicit publish/subscribe tree from a publisher to corresponding subscribers is addressed, and a greedy algorithm is proposed for tree building. In general, a tree structure is fragile in a mobile environment, so it is not suitable for networks with fast moving nodes.

QoS routing is an active research area for the mobile ad hoc networks. In CEDAR [67], a set of nodes are elected to form a backbone of the network. The backbone is called ”core” and covers the whole network. Bandwidth information of the stable links is propagated among the core nodes. The route discovery operation utilizing broadcasting along the core to find a routing path P to the destination node, then the routing path which can guarantee the bandwidth requirement is find with the help form P and the core nodes. In [12], a ticket based scheme is proposed. Multiple routing paths are searched in parallel and the tickets are used to constraint the route discovery packets flooding. These protocols are proposed for multimedia applications, and links are selected according to their bandwidth and stability for routing path construction. Some routing protocols are developed to support multimedia applications [64, 13] by exploiting cluster based approaches.

4.4.5 Conclusion and future work

Safety-critical applications are an important and emerging application domain of mobile ad hoc networks. They have high predictability and reliability requirements. Such requirements add tough challenges for the routing protocol design which includes the discovery of communication paths meeting QoS requirements and, particularly important, the maintenance of the paths, i.e., preserving the QoS under mobility constraints.

The Content and Cell based Predictive Routing (CCPR) protocol is proposed to support safety-critical applications in mobile ad hoc environments. To improve the predictability of the routing paths, mobile nodes are organized by cells. Cells are exploited as the basic units for multi-hop routing construction instead of individual nodes. This has the advantage that it allows to calculate the latency of a path by simply adding the latency in the cells and preserve this latency of the path in the presence of a constantly changing node population inside the cells. Of course, this requires that cell latency is predictable and secondly, that the cells are stable, i.e. there is enough redundancy in a cell to assure stable routing in spite of the mobility of individual nodes. Replication of routing information and proactive route maintenance strategies are used to meet this goal. If possible, a route is actively reconfigured to preserve the specified properties

For route discovery, the subject based addressing of the publisher/subscriber model is exploited. Cell Resource Tables are provided to quickly determine whether a requested event channel is available in a cell. This search style reduces the control packet flooding and saves the valuable bandwidth of the wireless channels. Only a fraction of nodes are involved in control packets flooding, and the search areas of route discovery packets are constrained.

Our future work includes investigation of redundant routing information to increase the robustness of the routing paths, the information aggregation for the content based routing in mobile ad hoc networks, the publisher/subscriber model in QoS routing for mobile ad hoc networks, and the performance simulation work.

4.5 A real-time event channel model for the CAN-Bus

The text provided in this section describes a real-time event channel model in a publisher/subscriber communication scheme. The model specifically considers temporal and reliability attributes and suggests an API that integrates the real-time aspects in the event, channel model. According to the need in most real-time systems, we support event channels with different timeliness and reliability classes. Hard real-time event channels are considered to meet all temporal requirements under the specified fault assumptions. The resource requirements for this type of channel are statically assigned by an appropriate reservation scheme. Soft real-time event channels are scheduled by their deadlines, but they are not guaranteed under transient overload conditions. Non real-time event channels are used for events without any specified timeliness requirements in a best-effort manner. The last part of the text presents how the different channel classes are mapped to the mechanisms necessary to implement the model on the CAN-Bus.

4.5.1 Introduction

The publisher/subscriber (P/S) model has been recognized as an appropriate high-level communication scheme to connect autonomous components in large distributed control systems [63, 2]. Particularly, P/S supports autonomy of components by an asynchronous notification mechanism omitting any implicit control transfer coupled with the exchange of information. This is in contrast to other high-level interaction models as remote procedure calls or remote invocations which create global dependencies and side effects which require complex mechanisms to handle temporal and functional fault situations. Secondly, P/S relies on a content-based communication scheme. This means that the content of a message is used to route a message rather than an address. A subscriber, which is interested in particular information, e.g. the temperature data of some sensor, subscribes to the particular information rather than to a specific sensor. This has the advantage that the subscriber does not have to know any specific sensor name or address and thus, a content-based addressing mechanism substantially encourages dynamic adaptability and extensibility requirements. Moreover, it may be the basis of transparent fault-tolerance mechanisms in which multiple redundant sources provide the same information. In our P/S protocol we adopted a slightly less general variant of content-based addressing which maps an arbitrary content to a subject field. Subject-based addressing supports our event channel concept and is more suited in a real-time environment because of predictability reasons. Additionally, subject-based addressing can be optimized to meet the requirements of the restricted computational resources found in a control system composed from smart sensors and actuators [44]. A more detailed discussion of the P/S model is beyond the scope of this paper and can be found in [58, 53].

In this paper we will describe our P/S model which is based on event channels and particularly intended for real-time control applications. We will briefly review the specific problems when implementing the P/S model in a system composed from smart sensors and actuators which usually have only limited computational performance. The focus of our paper is the description of different real-time event channel classes and their programming interface. Finally, we will show, how to map this model to lower level mechanisms of the CAN-Bus to enforce the required real-time guarantees.

4.5.2 Events and event channels

Events and the associated event channels are central conceptual constructs in our system. An event is related to an occurrence in the real world, e.g. observed by a sensor, or an in the control system itself, e.g. generated by some control program. Subscribers, which have stated their interest in an event, are asynchronously notified when this event occurs. An ***event*** is an instance of an event type, which is characterized by a subject, attributes and content.

event := <subject, attribute_list, content>

According to the subject-based addressing in the P/S protocol, the subject is a tag related to the content of an event. In our system, a subject is represented by a unique identifier. The attributes are related to the context, in which an event is generated like location, time, mode of operation, etc. and to quality aspects like a validity interval (expiration time) and a deadline[5]. They represent non-functional properties of the event. The content of an event carries the data and is represented as a structured set of functional parameters. The fields of the content are accessible by specific methods.

An ***event channel*** is an architectural component of the middleware which disseminates all events of a certain subject. Publishers and subscribers interact with the event channel to publish events or receive notifications. An event channel is an instance of an event channel type. It comprises the subject of events which can be disseminated by the channel and attributes.

event_channel := <subject, attribute_list >

In contrast to the attributes of an event which describes the properties of a single individual occurrence of an event, the attributes of the event channel abstract the properties of the underlying communication network and dissemination scheme. Therefore attributes include e.g. latency, dissemination constraints and reliability parameters. An event channel is dynamically created whenever a publisher makes an announcement for publication or a subscriber subscribes for an event notification. For every event type there is at most one event channel. An event channel may handle multiple publishers and multiple subscribers, thus, representing a many-to-many communication channel. When a publisher announces publication, the respective data structures of an event channel are created by the middleware. When a subscriber subscribes to an event channel, it may specify attributes of the event and the event channel. As described later, these attributes are used for type checking and filtering.

[5] Note that for a soft real-time event, the deadline may be missed. In this case the expiration time defines the interval after which the event may be dropped entirely.

4.5.2.1 Routing, filtering and binding Implementing the publisher/subscriber model requires to map the abstractions of that model like events and event channels to the elements provided by the technical infrastructure of the system such messages and addresses. In the publisher/subscriber scheme, events are routed by their content from a publisher to an interested subscriber. Therefore the issue of how to map this to a network includes the problem of 1. getting a message to the right destination (routing) and 2. that the destination should only receive those messages which it is interested in (filtering). A purely content-based communication scheme [9, 56] seems to be not appropriate in a real-time control system. This is mainly because of the temporal unpredictability related to this approach. Firstly, the routing task just uses a broadcast or some central server facility to disseminate every message to every potential destination and leaves the selection of the right event to the local filtering task. This obviously will create a high overhead and a large degree of unpredictability. Secondly, the complex filter mechanism, which has to work on arbitrary message length and many formats, creates another source of temporal uncertainty. Additionally, if the nodes are smart sensors and actuators, the content-based scheme is hardly feasible simply because of the usually limited computational resources of these components. Also approaches that reduce the network traffic by confining the flooding of events [1] still need the filtering capabilities in the nodes and thus are too heavy weight for the envisaged scenario.

A subject-based mechanism will primarily ease the routing and filtering functions. Now, the evaluation of arbitrary predicates on the content of an event can be confined to a dedicated subject field in the message. This quite perceptibly leads to the notion of event channels, which exclusively disseminate events carrying a specific type of information. Because event channels are objects of the architecture, additional attributes can be assigned to them as it was introduced above. These event channel attributes then may be used for additional filtering. As a further benefit, the knowledge about which publishers and subscribers exist for a specific channel can be exploited for a more economic routing mechanism. In our system, a subject is specified by a unique identifier [44].

We introduce an additional optimisation for routing events in a subject-based model trading the degree of flexibility against improved routing performance and filtering overhead. It has to be noted that even in the subject-based scheme, every event message has to be examined by the respective subscriber because the subject identification is in the body of the message. The idea now is to dynamically bind a subject of an event to an address of the underlying network. Then, the network hardware automatically takes on the job of routing the message to the respective destinations. The local communication controller filters all messages that don't match the subject out of the message stream. Hence, the subject filtering does not put any burden to the embedded computational component of a smart sensor or actuator. The details of this approach, particularly for a CAN-Bus have first been presented in [45] and later extended in [44].

The middleware architecture that supports our event system is completely distributed. It is composed from local event channel handlers, which maintain the data structures for the local channels and provide services as attribute filtering event notification and exception handling. This contrasts to the centralized server solutions of event systems as described in [34, 30]. The event channel handlers perform the dynamic binding protocol transparently for the application. Therefore, publishers and subscribers just need to know the subject of an event channel to communicate.

In this paper we will focus on the definition of real-time event channels, their API and the mechanisms to enforce the specified temporal and reliability properties on the CAN-Bus.

4.5.2.2 Real-time event channels According to the need in most real-time systems, event channels with different timeliness and reliability properties should be supported. Therefore, we distinguish three event channel classes: hard real-time event channels (HRTEC), soft real-time event channels (SRTEC) and non real-time event channels (NRTEC). A HRTEC offers rigorous guarantees for discrete control based on sporadic events as well as for continuous control requiring periodic events like sensor readings and control feedback. For sporadic events a maximum latency will be guaranteed while for periodic events the goal is to achieve a low period- and latency-jitter. The guarantees are maintained under an anticipated number of network failures. Events published to a SRTEC are scheduled according to the earliest deadline first (EDF) algorithm. As outlined below, deadlines may be missed in situation of transient overload or due to the arbitrary arrival times of messages. Finally, a NRTEC disseminates events that have no timeliness requirements.

The transport of events through a hard real-time event channel (HRTEC) is synchronous and reliable. The properties of a HRTEC are defined by: 1.) a known upper bound for the transport latency, i.e. the interval between the point in time when an event message becomes ready and its delivery; 2.) a known upper bound for the latency jitter, i.e. the variance of the transport latency; 3.) a known upper bound of the period jitter for periodic events, i.e. the variance on the period; 4.) a fault assumption under which the properties 1.)- 3.) are valid. In order to offer such properties, a HRTEC transparently handles redundant transmissions of events and guarantees that the respective publisher has a privileged access to the communication network. Access is based on the reservation of network resources according to a TDMA mechanism (TDMA: Time Division Multiple Access) similar to the time-triggered protocol [48]. It means that events published to the HRT C have to be ready at the start of the assigned TDMA time-slot. As explained below, in contrast to most TDMA schemes, we exploit the specific priority mechanisms of the CAN-Bus to enforce the temporal guarantees in a more flexible way.

A SRTEC has timeliness requirement which are expressed by deadlines and validity intervals (expiration time). Different from HRTCs, SRTECs do not use reservations. Soft real-time event messages become ready at any time and are scheduled according to their transmission deadlines by an earliest deadline first (EDF) algorithm. The transmission deadline is defined as the latest point in time when a message has to be transmitted. As described later, the priority mechanism of CAN is exploited for this purpose. However, because a message can not be interrupted during its transmission and messages may become ready at arbitrary points in time, EDF will not always take the right scheduling decisions (only a clairvoyant scheduler [43] would be able to do so) and situations of temporal conflicts and transient overload may occur. In theses situations, messages will still be transmitted at a later time in a best effort manner. An SRT event message eventually will be discarded if its transmission time is delayed beyond its temporal validity. The expiration time is an application specific parameter, which may be defined according to some value function [41].

NRTCs are used for events that do not have timeliness requirements. They are primarily intended for configuration and maintenance purposes. While HRTC and SRTC disseminate events of restricted length to meet the responsiveness requirements of real-time systems, NRTC may transfer bulk data in a sequence of message fragments.

Subsequently, we will describe the application-programming interface of the different event channel types in more detail.

Hard real-time event channels The transfer of events through a HRTC is certain, i.e. all the necessary resources are reserved to transmit event messages timely under specified fault assumptions. The API for a HRTC is presented in Figure 24. HRTCs need to set up the infrastructure before communication in compliance with the required guarantees. This is initiated by an application through calling the announce method:

channel.announce(subject,attributeList,eHandler);

The *announce* method enables the local middleware components to set up the data structures representing the respective event channel and performing the binding of the event channel subject to a network address. Three arguments are specified for the method: The subject, represented by the unique identifier of the event channel, the *attributeList*, and an exception handler. The *attributeList* describes the specific attributes of the channel, e.g. whether the publication is periodic or aperiodic, reliability requirements and data rates. This information is used to allocate and reserve the respective resources. For a hard real-time channel it is not common to provide exception handling because it usually is based on fault masking and worst-case assumptions about temporal properties. However, it should be noted that in a distributed system, local exception handling may contribute to an early detection of a fault and thus may increase the safety of the system. The lower levels of the communication system may detect a failure, which cannot be handled by the fault masking mechanism, and propagate this information through the middleware to the respective subscribers of a channel.

```
class hrtec {
private:
subject subject_uid;
public:
// constructor and destructor of the class
hrtec(void);
~hrtec(void);
// methods used for publishing
int announce(subject, attribute_list, exception_handler);
int publish(event);
// methods used for subscribing
int subscribe(subject, attribute_list, event_queue, not_handler, exception_handler);
int cancelSubscription(void);
}
```

Figure 24: Declaration of a HRTEC class in C++.

When the HRTC is established, the application can publish events to the channel using the method:

channel.publish(event);

For subscribers the following methods are provided:

channel.subscribe(subject, attributeList, eQueue, nHandler, eHandler);

channel.cancelSubscription();

The *subscribe* method establishes the necessary channel data structures and creates the binding of the subject to a network address. It corresponds to the *announce* method for publishers. The *attributeList* specifies a list of attributes used for allocating the respective resources and for filtering. For instance, we generally assume that publishers and subscribers are connected by a channel which spans multiple networks, e.g. a field bus, a wireless network and a wired wide area network[6]. In such a scenario, a subscriber may be interested in receiving events only from publishers in the same network, i.e. those connected to the same field bus. In such a case, the respective attribute can be set accordingly and any event, which has been generated outside the field bus, will be filtered out and will not trigger a local event notification. It should be noted, however, that the HRT-channels are statically assigned to time-slots and have predefined temporal and reliability attributes. Therefore, the filtering is usually less important for this channel class because only a particular publisher is allowed to publish in a certain time-slot (see chapter 4). The subscriber to such a channel is thus always aware which entity is expected to transmit. The known time of transmission itself therefore will be exploited as a filter for a HRT-channel.

Because events can be aperiodic, the event notification service of the middleware provides an asynchronous notification mechanism for applications. When an event has passed the filters, the middleware stores the event in some predefined memory area and calls the application's notification handler *nHandler*. The notification handler comprises application code that is executed when an event is received. Thus, the notification handler retrieves the event from memory using the *getEvent* primitive and then proceeds performing the respective operations. As for the publisher of a HRTC, an exception handler is also specified for the subscriber. Because a HRTC is based on reservations, the time when a message is expected is known and thus, the event channel handler on the subscriber side can detect a missing message.

Finally, the *cancelSubscription* method removes a subscription. Note that a cancel subscription is a strictly local operation and releases the resources in the local event handler. Only subscribers can dynamically cancel subscription to a HRTC.

Soft real-time event channels SRTC do not use reservations. In SREC transmission deadlines are used to dynamically schedule the event traffic. Figure 25 depicts the declaration of the SRTC class. Although the structure looks similar to the HRTC, the differences are substantial and primarily are substantiated in the different attributes defined for SRTCs. Events published to a SRTC specify a transmission deadline and an expiration parameter in the attribute list of the event. As already discussed, events are scheduled by the EDF algorithm which may lead to missed deadlines because of the non-preemptive nature of the message transmission and because of overload situations. This situation requires notifying the application

[6] An example is described in [44].

for awareness reasons. Two exceptional situations may occur: A missed deadline and an expired validity. In both cases, the local exception handler is called. This local notification allows the application to react and adapt to such situations. When the validity interval is expired, the event is completely removed from the local send queue.

```
class srtec {
private:
subject subject_uid;
public:
// constructor and destructor of the class
srtec(void);
∼srtec(void);
// methods used for publishing
int announce(subject, attribute_list, exception_handler);
int cancelPublication();
int publish(event);
// methods used for subscribing
int subscribe(subject, attribute_list, event_queue, not_handler, exception_handler);
int cancelSubscription(void);
}
```

Figure 25: Declaration of a SRTEC class in c++.

The announce method also establishes the local data structures and initiates the binding mechanism but no reservations are made. Chapter 3 provides further details about the realisation on the specific CAN network infrastructure. Another difference to a HRTC is that a publisher can stop its publications to a SRTC and release the local resources by a *cancelPublication* method.

Non real-time event channels Non real-time event channels are used for events that do not have timeliness requirements. The declaration of a NRTEC class in c++ is shown in Figure 26. A NRTEC has a fixed priority. The priority is specified by the application during the announcement of the channel. However, as further discussed in the next chapter, only priorities within a predefined range are accepted by the middleware. The *announce* method has the format:

channel.announce(subject,attribute_list, fixedPriority);

NRT-channels are particularly used to configure and maintain the smart networked devices of the system. This may require to send a considerable amount of data over the network, like memory images, electronic data sheets, or test patterns. Because message frames on the CAN-Bus are limited to a payload of 8 data bytes, a mechanism to chain individual CAN messages to a larger application specific message is needed. Such a ”fragmentation” mechanism for NRT channels which publishes long event messages in multiple fragments is provided by the middleware. Fragmentation is an inherent attribute of a NRT-channel and therefore, on the publisher side, fragmentation is defined during the announcement of the event channel as an entry in the *attributeList*.

```
class nrtec {
private:
subject subject_uid;
fixed_priority fixedPriority;
boolean fragmentation;
public:
// constructor and destructor of the class
nrtec(void);
~nrtec(void);
// methods used for publishing
int announce(subject, attribute_list, fixed_priority);
int cancelPublication();
int publish(event);
// methods used for subscribing
int subscribe(subject, attribute_list, event_queue, not_handler, exception_handler);
int cancelSubscription(void);
}
```

Figure 26: Declaration of a NRTEC class in c++.

4.5.3 Event channels on a CAN-Bus network

We now present a mapping of the described abstractions to a field bus network. We will first describe the reservation scheme for the HRTCs which is similar to a scheme used in time-triggered protocols like TTP [48], TTP/A [32], and TT-CAN [27]. However, a substantial advantage over a TDMA scheme is that due to CAN-bus properties, bandwidth which was reserved but is not needed by a HRTC can be used by less critical traffic. Because of the conservative worst-case assumptions of a HRTC, this can be a large share of the overall bandwidth. For SRTC the priority-based message dispatching of the CAN-Bus is exploited to realize an EDF-based scheduling of messages. Constraints on the priority level of SRTC prevent any interference with reserved HRTCs. Finally, the low fixed priorities of NRTCs enable the use of any free bandwidth for non-critical bulk data transfer.

4.5.3.1 The reservation scheme Hard real-time communication is organized in rounds. A round is divided into time slots that are assigned to HRTCs. A round specifies the cycle in which the schedule of the communication medium is repeated. The data structure which stores the schedule of a round is called a calendar and corresponds to the Round Descriptor List (RODL) in the TTP protocol [48]. The intention of the reservation-based scheme is to avoid collisions by statically planning the transmission schedule. Multiple slots may be reserved for an individual node within a round. The correctness of the reservations regarding timing conflicts and temporal overlap are checked by an admission test. We assume that this is done before any new reservation is confirmed and that the reservations are made off-line. Figure 27 depicts the organization of a communication round composed from the reserved time-slots and contention periods (CP) in which soft real-time and non-real-time messages are scheduled. A gap ΔG_{min} between reserved slots has to be inserted to prevent any temporal conflict between reserved HRT slots because of the time-skew of the synchronized clocks.

The event channel approach of the P/S protocol leaves open which publisher provides the respective information to a subscriber. However, when defining a HRT event channel, the slot reservation has to be done according to a specific node, which is allowed to exactly send a message within this time interval. Hence, if multiple publishers provide input to the same channel, multiple slots have to be reserved. As it becomes clear later, the specific handling of reserved slots on the CAN-Bus allows a flexible treatment and reuse of reserved slots. Thus, if a node has already successfully published a certain event in a hard real-time channel, subsequent publications of the same subject can possibly be suppressed and the slot can be used by less critical traffic.

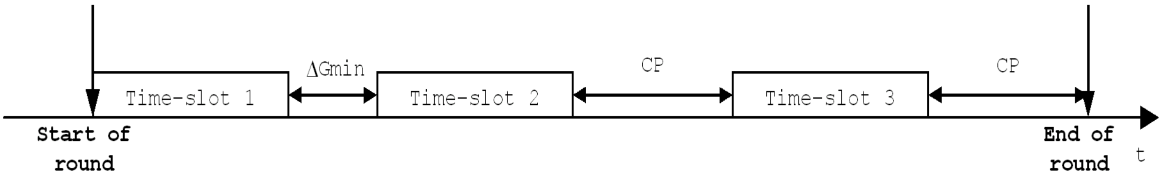


Figure 27: Organization of a Round.

4.5.3.2 Structure of the time-slots Hard real-time messages use time-slot reservations. This mechanism is based on a global time which, in our protocol, is provided by synchronized clocks. The length of the time-slot is defined by the worst-case transmission time of a HRT message. The worst-case transmission time considers the length of a message and includes assumptions about the failure modes during message transmission. We use time redundancy and forward error recovery to cope with network omission faults and temporary node faults. An analysis of the worst-case transmission times under fault assumptions is published in [50]. To deliver a message at its predefined deadline, the transmission has to be launched at the Latest Start Time (LST) (see Figure 28). We exploit the CAN priority mechanism to enforce that a HRT message will definitely be sent at this point in time. The highest priority is exclusively reserved for HRT messages. When the LST is reached this priority is assigned to the message. The priority mechanism of the CAN-Bus will then assure that this message will win the bus arbitration. Thus, our scheme exploits the global time to avoid any conflicts between HRT messages. The priority mechanism of the CAN-Bus enforces the reservations and omits any interference with less critical messages. To guarantee that SRT- and NRT-messages do not interfere with HRT-messages, we additionally have to consider that an ongoing message transfer cannot be preempted. It may happen that messages (SRT or NRT), which can be sent at any time, will be started just before the HRT-message becomes ready, i.e. at the LST. In this case, it would steel reserved time from the HRT-message and jeopardize HRT guarantees. Therefore, the slot for HRT has to be extended and HRT-messages must be ready for transmission at the latest ready time (see Figure 28) which is: LST - ΔT_{wait}. The time ΔT_{wait} corresponds to the transmission time of the longest CAN-message (154 μsec at 1Mbit/sec). The actual successful transmission of the HRT-message can then occur at any time inside the time-slot which introduces the problem of jitter. To avoid the jitter for the application, HRT messages are always delivered by the middleware at the predefined transmission deadline.

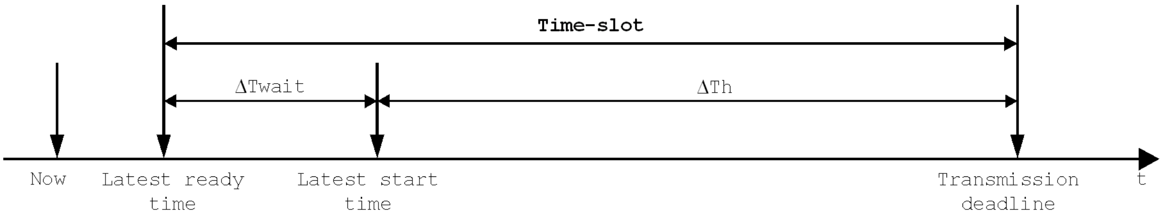


Figure 28: Structure of a time-slot.

Thus, our reservation mechanism differs from other time-triggered approaches in many substantial points. Firstly, we exploit time **and** the priority mechanism to enforce HRT guarantees. Time is used to separate HRT-messages which all have the same maximum priority in the system. This guarantees that they are sent when they become ready. However, the CAN-Bus allows to determine, without any additional overhead, whether all operational nodes including the sending node have received a message successfully. Observing that we included time redundancy to tolerate transmission faults, this redundancy is not needed any more if all nodes have correctly received the message. Therefore, the sending node will stop to further transmit the message in this case. Now, if there are pending SRT- or NRT-messages, the priority mechanism of CAN will automatically schedule the message with the highest priority for transmission. Thus time redundancy only costs bandwidth if faults really occur, which, compared to the overall traffic, may be relatively rare. Therefore, very conservative fault assumptions are possible because the penalty is low in the average. This is not possible in schemes which only use global time to enforce reservations.

Secondly, most TDMA schemes as TTP [48] and TT-CAN [27] enforce the constraints on jitter on the network level by sophisticated and expensive mechanisms to guarantee start times of messages. In the time-triggered CAN-Bus protocol (TT-CAN [27]) there are extra slots to prevent that an ongoing message transfer interferes with a reserved slot, thereby wasting additional bandwidth. In our protocol, jitter is prevented by defining a precise delivery deadline of a message and the event channel handler which has access to the global clock assures that the message is delivered at that point. Thus, jitter is handled on the middleware layer rather than on the network layer. It also should be noted that a HRT-message will be correctly received at some time within the slot, which cannot be predicted because faults may occur or not. In TTP and TT-CAN, messages are just resend up to a certain number of times which corresponds to a specified omission degree. This of course fills up the reserved slot and avoids jitter but for the price of valuable bandwidth. To reuse the remaining slot time means that a message transfer is successfully completed within the slot and the delivery has to be handled independently of the network at a higher system layer.

4.5.3.3 Scheduling soft real-time- and non-real-time-messages One of the most important properties of the CAN-Bus is its priority-based distributed arbitration mechanism. This mechanism is exploited to schedule SRT messages according to the EDF scheme. NRT messages use a low fixed priority. Table 7 summarizes the priority assignment. We use an 8-Bit explicit priority field in the CAN message identifier (see Figure 30) which result in 256 priority levels. As described in 4.5.3.2, HRT-messages reserve the highest priority 0. Scheduling SRT-messages requires a range of priorities which reflect the deadlines in time. This is outlined in 4.5.3.4. NRT messages are assigned to fixed low priorities. The relation between the priorities of HRT, SRT and NRT messages can be expressed by the relation[7]: $\mathrm{P}_{HRT} < \mathrm{P}_{SRT} < \mathrm{P}_{NRT}$. The assignment avoids that NRT and SRT messages ever gain access to the bus against any pending HRT message. The middleware rigorously has to enforce the above relation.

Priority	***Message Type***
0	HRT
1	SRT
...	SRT
N_{SRT}	SRT
1 + N_{SRT}	NRT
$\ldots 2^8$-1	NRT

Table 5: Priority mapping.

4.5.3.4 Soft real-time messages SRT messages are scheduled according to an EDF scheme which means that the assigned priorities reflect the deadline order of message transmissions. Mapping deadlines to priorities is outlined in Figure 29. We assume that we have 250 priority levels for SRT messages (this can be flexibly determined by application needs. In this example we define 1 priority level to HRT-messages and 5 levels to NRT-messages). These 250 priority levels have to be mapped on a time scale to express the temporal distance of a deadline. The closer the deadline, the higher is the priority (see Figure 29). Mapping deadlines to priorities will cause two problems. The first problem is that static priorities cannot express the properties of a deadline, i.e. a point in time. A priority corresponding to a deadline can only reflect this deadline in a static set of messages. When time proceeds and new messages become ready, a fixed priority mechanism cannot implement the deadline order any more. It is necessary to increase the priorities of a message when time approaches the deadline, i.e. with decreasing laxity. Therefore, the priority of a message is dynamically increased with a granularity of Δtp which defines a priority slot (see Figure 29). The priority of a SRT message will reach the highest possible priority at its transmission deadline. Secondly, there is a trade-off between the length of a priority slot and the quality of the derived schedule. When mapping a transmission deadline for a message to a priority slot, two deadlines which are close may be mapped to the same slot and thus, to the same priority. The order between the deadlines is then arbitrary determined by the other fields of the CAN identifier. This would motivate a very small slot length, which decreases the probability of equal priorities. However, this raises the problem of a tight time horizon. The

[7] On CAN-Bus the highest priority corresponds to the lowest binary value.

time horizon is given by $\Delta H = (P_{max} - P_{min}) * \Delta t_p$. Any deadlines, which are beyond this value are mapped to the same priority and thus may be scheduled incorrectly. Figure 29 depicts this situation. The described trade-off is particularly a problem when the number of priority levels is low (e.g. in [73]). Considering the 250 priority slots provided in our scheme and a priority slot length of approximately one CAN-message, we can accommodate 250 message transfers within the time horizon. Because the number of nodes connected to a CAN-Bus is usually in the range of 32 to 64, the time horizon seems to be sufficiently large. A decision about the trade-off has also to consider specific application requirements.

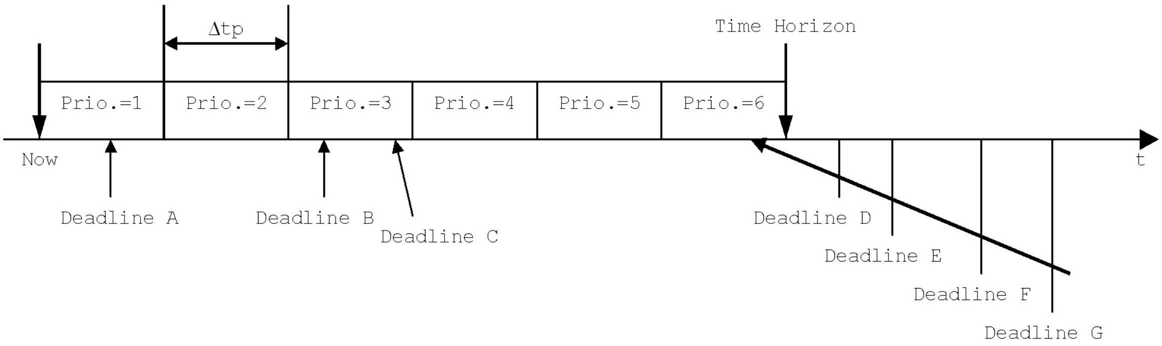


Figure 29: Mapping deadlines to priorities.

4.5.3.5 Structuring the CAN message identifier The transport mechanism for CAN-Bus uses the basic communication mechanisms described in [45] to support different event channel classes on the CAN-Bus. An event channel class (HRTEC, SRTEC, NRTEC) is mapped to the respective message class on the CAN-Bus. The CAN 2.0B standard with 29-bit identifiers is used to express the priority of the message as well as to identify the channel. The identifier is structured into multiple fields to represent the priority and the subject of an event channel.

On the bus level the CAN message identifier is used to identify the event channel (subject) and to express the priority of the event message. Figure 30 outlines the format of a CAN message. The priority of the message is expressed in an 8-bit priority field providing 256 priority levels. A 7-bit field *TxNode* is used to ensure the uniqueness of the CAN-ID. This is a requirement of the CAN-Bus protocol [28] because after the arbitration phase, a consistent decision has to be derived which node is allowed to use the bus. The *TxNode* is dynamically assigned during the configuration phase [44]. The *etag* field (14-bit) represents the subject of the event channel. Whenever a new channel is created in a node (by an announce or a subscribe), the binding protocol assigns an *etag* to the respective unique subject identifier of the channel. A detailed description of both protocols can be found in [45].

4.5.4 Comparison with related higher level CAN protocols

On the CAN-Bus [28], schemes using message priority and time have been explored to achieve predictability of communication. Standard protocols based on CAN-Bus

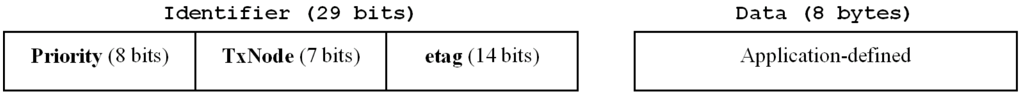


Figure 30: CAN-Bus event message.

such like CAL [14], SDS [38], and DeviceNet [22] are based on fixed priority schemes which are not able to reflect temporal requirements. To overcome this problem, the deadline-monotonic priority assignment [70] uses an off-line feasibility test to meet the deadlines of periodic and sporadic messages. However, it supports only static systems and does not distinguish hard and soft deadlines. More flexible protocols like the dual priority scheme [23] or the schemes proposed in [8, 73] support the use of both hard and soft real-time communication on the CAN-Bus. However, they have the disadvantage of providing a time horizon that is too short.

A protocol that uses time-slot reservation on CAN-Bus is the TT-CAN protocol [27]. As already mentioned, TT-CAN does not fully exploit the CAN-Bus facilities and therefore, uses bandwidth less efficiently. Secondly, non real-time messages can only be sent in the respective contention slots. This is less flexible compared to our approach. Moreover, it is based on a master-slave mechanism which we wanted to avoid in our system because the master constitutes a single point of failure. [8]The TTP/A protocol [32] aims to the same application area as TT-CAN and uses similar mechanisms. Here, the master always initiates the communication with the slaves sending their own messages in a predefined manner.

4.5.5 Concluding remarks and future work

In the paper we presented an event channel model that supports functional and non-functional attributes. Three types of channels are provided for applications: HRTEC, SRTEC, and NRTEC. The concept of event channels represents a high level programming interface for real-time communication. It provides the abstractions to ease the programming and enables the programmer to reason about non-functional attributes like temporal and reliability characteristics without being involved in low level network details. Communication via HRT channels is certain, i.e. it guarantees timely delivery under the specified fault assumption. SRT channels constitute a versatile and flexible way to express deadlines as timing constraints but provide awareness if the constraint is violated. It also allows specifying an expiration attribute, which is used to discard an event when the temporal validity is expired. Again, the application will be notified to enable corrective application related actions. Finally, NRT channels are established to transport non real-time traffic. Here, we allow arbitrary long messages to support configuration and maintenance data like ROM-images or electronic data sheets, describing a device.

The paper also shows how these channel abstractions are represented on a CAN-Bus, which is a popular field-bus in the area of industrial automation and particularly in the automotive area. On the CAN-Bus, the channel classes are directly mapped to respective message classes. The protocol for CAN-Bus provides three types of messages: hard real-time, soft real-time, and non real-time. Hard real-time messages use a reservation scheme to avoid any conflicting requirements between them. The

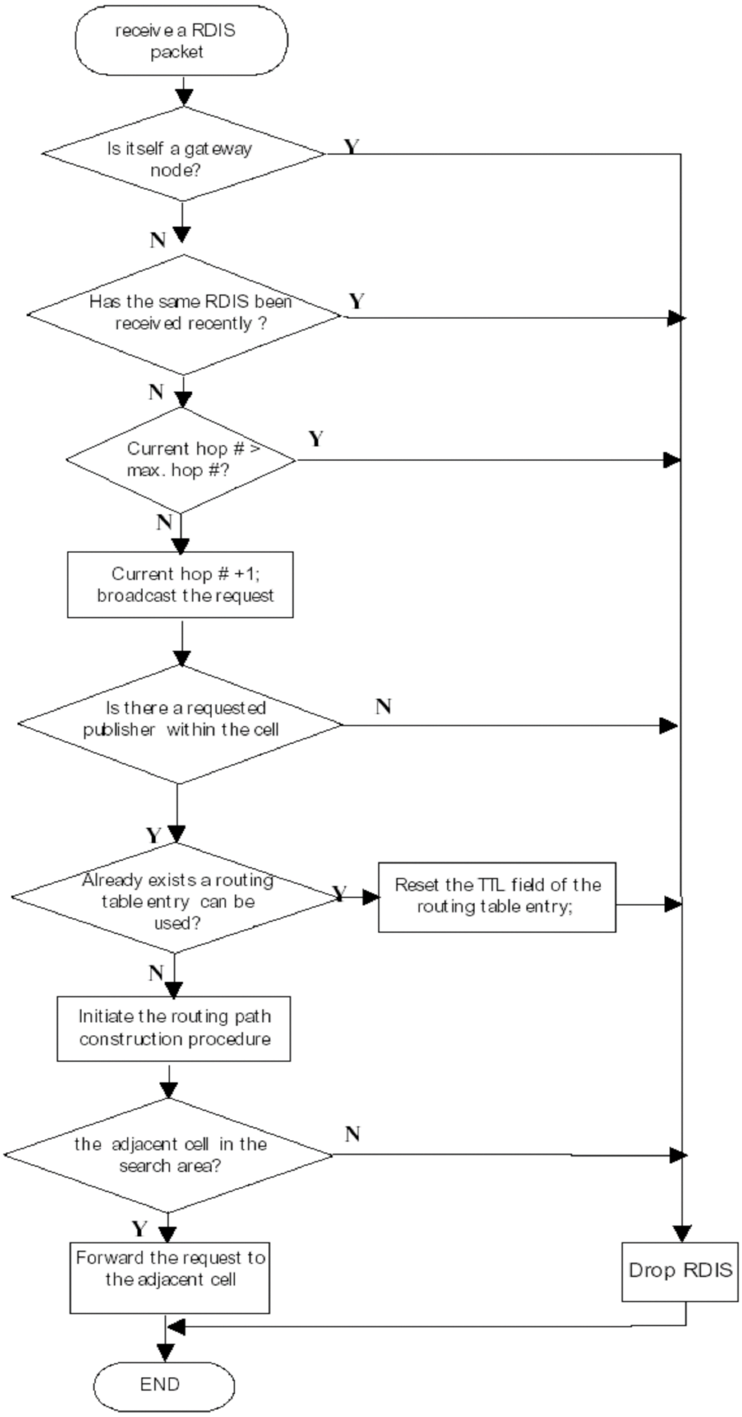
[8] This protocol is not based on CAN-Bus.

slot reservation mechanism also allows using time redundancy to tolerate omissions and crash failures. The priority mechanism of CAN is exploited to enforce the bus access in the asserted time-slot. For less stringent timeliness and reliability requirements, SRT messages are provided. In this type of communication deadlines are considered and flexible mechanisms to handle deadline violations are introduced. Finally, messages that do not have timeliness requirements will exploit the basic priority and fault handling mechanism of the CAN-Bus.

An additional advantage of our protocol compared to other time-triggered schemes is a better utilization of bandwidth. Firstly, when a reserved slot is not used, the priority mechanism of CAN will automatically assign this slot to some other (lower priority) message. This may occur when a sporadic HRT message has a reservation but is not sent. Secondly, the CAN-Bus provides a unique technique to determine when all nodes correctly receive a message. This property is exploited to reduce the overhead of time redundant message transfer. If all receivers have received the message correctly, the sender can detect this and skip the redundant message transmission.

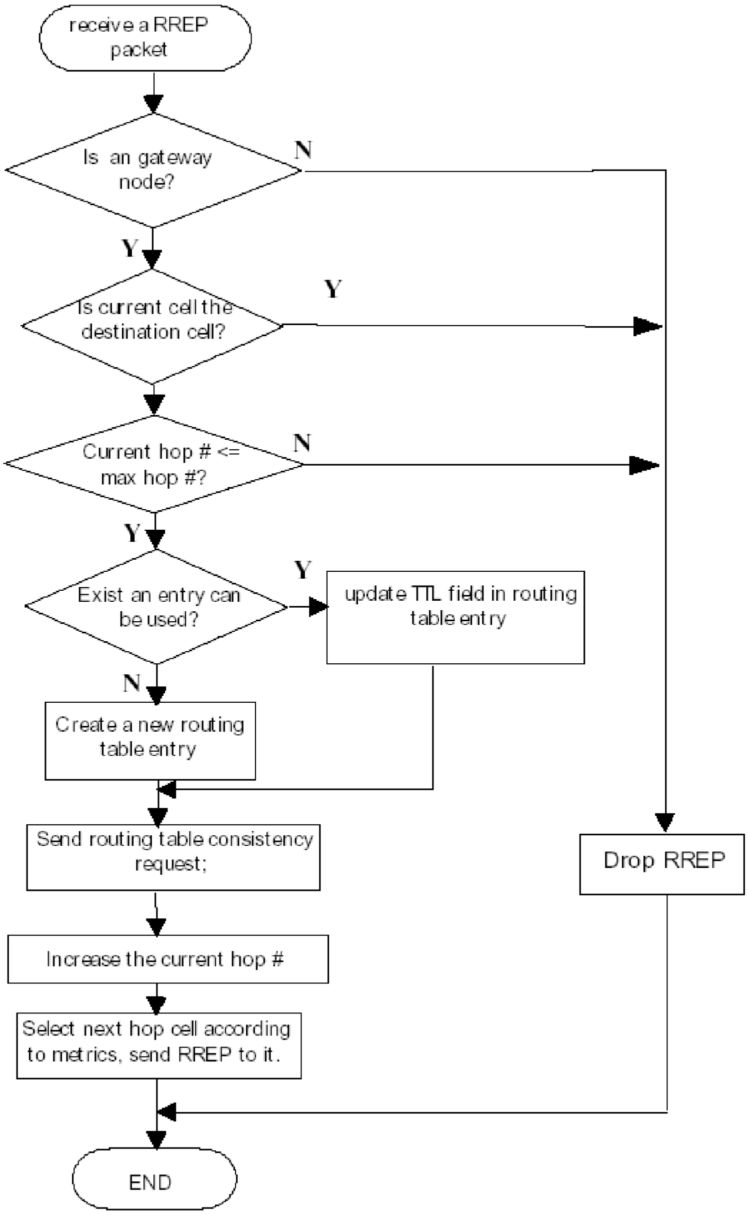
A first prototype of the P/S protocol is available. Presently, it provides the API for a single event channel class without non-functional attributes. It allows the automatic network configuration of the attached devices and supports the dynamic binding protocol. The middleware has a low memory footprint, which enables the use on various 16- and even 8-Bit micro controllers. A Linux and an RT-Linux version have also been realized. Future work includes the integration of the different event channel classes presented in this paper.

A Flowchart of the route discovery algorithm



receive a RDIS packet
Is itself a gateway node?
Y
N
Has the same RDIS been received recently ?
Y
N
Current hop # > max. hop #?
Y
N
Current hop # +1; broadcast the request
Is there a requested publisher within the cell
N
Y
Already exists a routing table entry can be used?
Y
Reset the TTL field of the routing table entry;
N
Initiate the routing path construction procedure
the adjacent cell in the search area?
N
Y
Forward the request to the adjacent cell
Drop RDIS
END

B Flowchart of the routing path construction algorithm


receive a RREP packet
Is an gateway node?
N
Y
Is current cell the destination cell?
Y
Current hop # <= max hop #?
N
Y
Exist an entry can be used?
Y
update TTL field in routing table entry
N
Create a new routing table entry
Send routing table consistency request;
Drop RREP
Increase the current hop #
Select next hop cell according to metrics, send RREP to it.
END

References

[1] D. S. Rosenblum A. Carzaniga and A. L. Wolf. Interfaces and algorithms for a wide-area event notification service. Technical Report CU-CS-888-99, Department of Computer Science, University of Colorado, October 1999.

[2] A. Seigel B. Oki, M. Pfluegl and D. Skeen. The information bus - an architecture for extensible distributed systems. *Operating Systems Review*, 27(5):58–68, 1993.

[3] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content based publish/subscribe systems. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 262–272, 2000.

[4] S. Basagni, I. Chlamtac, V. Syrotiuk, and B. WoodWard. A Distance Routing Effect Algorithm for Mobility (DREAM). In *Proc. 4th MOBCOM*, 1998.

[5] R.V. Boppana and S.P. Konduru. An adaptive distance vector routing algorithm for mobile ad hoc networks. *INFOCOM 2001*, 3:1753–1762, 2001.

[6] J. Broch, D.A. Maltz, D.B. Johnson, Y.C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proc. MOBICOM*, pages 85–97, 1998.

[7] S. R. Das C. E. Perkins, E. M. Royer. IP address autoconfiguration for ad hoc networks. IETF MANET, Internet draft, draft-perkins-manet-autoconf-00.txt.

[8] H. Thane C. Eriksson and M. Gustafsson. A communication protocol for hard and soft real-time systems. In *Proceedings of the EURWRTS'96*, 1996.

[9] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, apr 1989.

[10] A. Casimiro, P. Martins, P. Veríssimo, and L. Rodrigues. Measuring distributed durations with stable errors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 310–319, London, UK, December 2001.

[11] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[12] S. Chen and K. Nahrstedt. Distributed quality-of-service routing in ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 17(8), August 1999.

[13] C.C. Chiang, T. C. Tsai, W. Liu, and M. Gerla. Routing in clustered multihop, mobile wireless networks with fading channel. In *The Next Millennium, The IEEE SICON*, 1997.

[14] CiA. CAN Application Layer (CAL) for Industrial Applications, May 1993. *CiA Draft Standards 201..207*.

[15] Definition of application scenarios. CORTEX project, IST-2000-26031, Deliverable D1, October 2001.

[16] Preliminary definition of the interaction model. CORTEX project, IST-2000-26031, Deliverable D3, March 2002.

[17] Preliminary specification of the system architecture. CORTEX project, IST-2000-26031, Deliverable D4, April 2002.

[18] Flaviu Cristian. Synchronous atomic broadcast for redundant broadcast channels. In *Journal of Real-time Systems*, pages 195–212. Kluwer Academic Publishers, 1990.

[19] G. Cugola, E. Di Nitto, and G. P. Picco. Content-based dispatching in a mobile environment. In *Proceeding of WSDAAL 2000*, Ischia, Italy, September 2000.

[20] Raymond Cunningham and Vinny Cahill. "Time Bounded Medium Access Control for Ad Hoc Networks". In *The Second Workshop on Principles of Mobile Computing*, pages 1–8, September 2002.

[21] D. A. Maltz D. Johnson. Dynamic source routing in ad hoc wireless networks. In T. Imielinski and H. Korth, editors, *Mobile Computing*. Kluwer Acad. Publ., 1996.

[22] S. Siegel D. Noonen and P. Maloney. Devicenet application protocol. In *Proceedings of the 1st International CAN Conference*, 1994.

[23] R. Davis. Dual priority scheduling: A means of providing flexibility in hard real-time systems. Technical Report YCS230, University of York, UK, May 1994.

[24] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. "Epidemic Algorithms for Replicated Database Maintenance". In *The Sixth Symposium on Principles of Distributed Computing*, pages 1–12, August 1987.

[25] H. A. Duran-Limon and G. S. Blair. Reconfiguration of resources in middleware. In *7th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002)*, January 2002.

[26] C. Fetzer and F. Cristian. A fail-aware datagram service. In *Proc. of the 2nd Workshop on Fault-Tolerant Parallel and Distributed Systems*, Geneva, Switzerland, April 1997.

[27] Th. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time Triggered Communication on CAN (Time Triggered CAN-TTCAN). In *7th international CAN Conference*, 2000.

[28] Robert Bosch GmbH. Can specification version 2.0. Technical report, September 1991.

[29] Object Management Group. Corba 3.0 new components chapters. Technical Report CCM FTF Draft ptc/99-10-04, 1999.

[30] Object Management Group. Corbaservices: Common object services specification - notification service specification, version 1.0. Technical report, 2000.

[31] G. Gugola and E. Di Nitto. Using a publish/subscribe middleware to support mobile computing. In *Advanced Topic Workshop Middleware for Mobile Computing*, Heidelberg, Germany, November 2001.

[32] M. Holzmann H. Kopetz and W. Elmenreich. A universal smart transducer interface: Ttp/a. *International Journal of Computer System, Science Engineering*, 16(2), March 2001.

[33] Z. J. Haas. The Zone Routing Protocol (ZRP) for ad hoc networks, November 1997. Internet Draft.

[34] T. Harrison, D. Levine, and D. Schmidt. The design and performance of a real-time corba event service. In *Proceedings of the 1997 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, pages 184–200, Atlanta, Georgia, USA, 1997. ACM Press.

[35] Yongqiang Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. In *Proceedings of the Second ACM International Workshop on Data Engineering for Wireless and Mobile Access*, 2001.

[36] Yongqiang Huang and H. Garcia-Molina. Publish/subscribe tree construction in wireless ad hoc networks. Technical report, Stanford University, November 2001. http://dbpubs.stanford.edu:8090/pub/2001-54.

[37] Ietf manet charter. http://www.ietf.org/html.charters/manet-charter.html.

[38] Honeywell Inc. Smart distributed systems, application layer protocol version 2. Technical report, 1996. *Micro Switch Specification GS 052 103 Issue 3*.

[39] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Sixth Annual International Conference on Mobile Computing and Networks (MobiCOM 2000)*, Boston, Massachusetts, USA, August 2000.

[40] P. Jacquet, P. Muhlethaler, and A. Qayyum. Optimized Link State Routing Protocol, 1998. IETF MANET, Internet draft.

[41] E. D. Jensen and J. D. Northcutt. Alpha: A non-proprietary os for large, complex, distributed real-time systems. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, pages 35–41, Huntsville, Alabama, USA, October 1990. IEEE Computer Society Press.

[42] Mingliang Jiang, Jinyang Li, and Y. C. Tay. Cluster Based Routing Protocol (CBRP). Internet draft,draft-ietf-manet-cbrp-spec-01.txt.

[43] J.W.Liu. *Real-Time Systems.* Prentice Hall, Upper Saddle River, New Jersey, USA, 2000.

[44] J. Kaiser and C. Brudna. A publisher/subscriber architecture supporting interoperability of the can-bus and the internet. In *IEEE Int. Workshop on Factory Communication Systems*, Västeras, Schweden, August 2002.

[45] J. Kaiser and M. Mock. Implementing the real-time publisher/subscriber model on the CAN-Bus. In *Proceedings of the 2nd IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC'99)*, Saint-Malo, France, May 1999. IEEE Computer Society Press.

[46] J. des Rivieres Kiczales, G. and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Presss, 1991.

[47] Y. B. Ko and N. H. Vaidya. Location aid routing (lar) in mobile ad hoc networks. In *Proc. ACM/IEEE MOBICOM*, October 1998.

[48] H. Kopetz and G. Grünsteidl. Ttp - a time-triggered protocol for fault-tolerant real-time systems. Technical Report rr-12-92, Institut für Technische Informatik, Technische Universität Wien, Treilstr. 3/182/1, A-1040 Vienna, Austria, 1992.

[49] C. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[50] M.A. Livani and J. Kaiser. Evaluation of a hybrid real-time bus scheduling mechanism for can. In *7th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'99)*, San Juan, Puerto Rico, April 1999.

[51] N. Parlavantzas M. Clarke, G. Coulson and G. S. Blair. An efficient component model for the construction of adaptive middleware. In *IFIP/ACM Middleware'2001*, November 2001.

[52] P. Maes. Concepts and experiments in computational reflection. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'87)*, pages 147–155, October 1987.

[53] R. Meier and V. Cahill. Taxonomy of distributed event-based programming systems. Technical Report TCD-CS-2002, Dept. of Computer Science, Trinity College Dublin, Ireland, March 2002.

[54] S. Savage Mercer, C. W. and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[55] Microsoft. Com: Delivering on the promises of component technology. Technical report, Microsoft Corporation, 2000. http://www.microsoft.com/com/default.asp.

[56] Sun Microsystems. Javaspace specification. Technical report, March 1998. http://java.sun.com/products/jini/specs.

[57] Sun Microsystems. Enterprise javabeans technology. Technical report, 2001. http://java.sun.com/products/ejb/.

[58] R. Guerraoui P. Th. Eugster, P. Felber and A.-M. Kermarrec. The many faces of publish/subscribe. Technical Report DSC ID:200104, EPFL, Lausanne, Switzerland, 2001.

[59] G. Parado-Castellote, S. Schneider, and M. Hamilton. Ndds: The real-time publish subscribe network. In *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, CA, USA, 1997.

[60] V.D. Park and M.S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. *Proceedings of the Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'97)*, 3:1405–1413, 1997.

[61] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. *ACM Comput. Commun. Rev. (ACM SIGCOMM'94)*, 24(4):234–244, October 1994.

[62] C.E. Perkins and E.M. Royer. Ad hoc on demand distance vector routing. In *Proceedings of the Second IEEE Workshop on mobile computing systems and applications (WMCSA'99)*, pages 90–100, 1999.

[63] M. Gagliardi R. Rajkumar and L Sha. The real-time publisher/subscribe inter-process communication model for distributed real-time systems: Design and implementation. In *IEEE Real-time Technology and Applications Symposium*, June 1995.

[64] R. Ramanathan and M. Steenstrup. Hierarchically-orangized, multihop mobile wireless networks for quality-of-service. *ACM/Baltzer Mobile Networks and Applications*, 3(1):101–119.

[65] J. Rufino and P. Veríssimo. A study on the inaccessibility characteristics of ISO 8802/4 Token-Bus LANs. In *Proceedings of the IEEE INFOCOM'92 Conference on Computer Communications*, Florence, Italy, May 1992. IEEE Computer Society Press.

[66] Corson S. and Macker J. Mobile Ad Hoc NETworking (MANET): routing protocol performance issues and evaluation considerations. Technical Report RFC 2501. Internet draft, draft-ietf-manet-issues-01.txt.

[67] P. Sinha, R. Sivakumar, and V. Bharghaven. Cedar: a core-extraction distributed ad hoc routing algorithm. *IEEE INFOCOM*, March 1999.

[68] J. Stankovic. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 1995.

[69] Hamdy A. Taha. *"Operations Research: An Introduction"*. Prentice Hall, 1997.

[70] K. Tindell and A. Burns. Guaranteeing Message Latencies on Controller Area Network (CAN). In *Proceedings of the 1st International CAN Conference*, 1994.

[71] P. Veríssimo, J. Rufino, and L. Rodrigues. Enforcing real-time behaviour of LAN-based protocols. In *Proceedings of the 10th IFAC Workshop on Distributed Computer Control Systems*, Semmering, Austria, September 1991.

[72] Hu Zhou and S. Singh. Content Base Multicast (CBM) in ad hoc networks. In *Workshop on Mobile Ad Hoc Networking and Computing (MobiHoc)*, August 2000.

[73] K.M. Zuberi and K.G. Shin. Scheduling messages on controller area network for real-time cim applications. *IEEE Transactions On Robotics And Automation*, 13(2):310–314, April 1997.