Analysis of QoS in the Meteor MW

Author:
Andreas Ziethén

Supervisors:
Martin Törngren
Detlef Scholle
Barbro Claesson
Tahir Naseer Qureshi

August 19, 2008
Examsarbete MMK2008:49 MDA 329
Analys av QoS i mellanvaran Meteor
Andreas Ziethén

Sammanfattning

Detta examensarbete har utförts i samarbete med ENEA. Delar av demonstrationsplattformen SHAPE som ENEA har utvecklat för DySCAS-projektet har använts i implementationsdelen av detta examensarbete.


I examensarbetet har den tidigare nämnda QoSM designats och delvis implementerats. En applikationsmodell av en typisk Med Tech-produkt användes för att härleda kraven på QoSM. Existerande mellanvaror som HADES och ARMADA analyserades också för att identifiera nödvändiga krav på Meteor. Dessa krav användes för att designa en QoSM.


Intressanta områden för framtida arbete identifierades så som vidareutveckling av QoSM med olika QoS-nivåer, andra schemaläggningsalgoritmer och koppling av schemaläggning till schemaläggning av resurser utöver nätverket.

by: Andreas Ziethén
aziethen@gmail.com

Kungliga Tekniska Höskolan
Department of Mechatronics
ENEA
Abstract

This thesis is closely related to the Dynamically Self-Configuring Automotive System (DySCAS) project. DySCAS is a middleware for electronics in an automotive system. The thesis work has been performed in cooperation with ENEA. Parts of the demonstration platform Self configurable High Availability and Policy based platform for Embedded system (SHAPE) developed for the DySCAS project by ENEA have been used in the implementation phase of this master thesis.

The goal of this thesis was to evaluate how the Meteor MW, an offspring of the DySCAS middleware, should handle QoS and to design a Quality of Service Manager (QoSM) for the Meteor MW that fulfilled hard real-time requirements. The thesis also had the objective of analysing and identifying the requirements on such a QoSM from a QoS perspective. Specifically the requirements to guarantee real-time support for applications running on the MW. Lastly this thesis had the goal of evaluating other existing MW having support for real-time requirements.

In this thesis the aforementioned QoSM was designed and partially implemented. An application model of a typical Med Tech system was used to derive requirements necessary for the QoSM. Existing MWs such as HADES and ARMADA were also analyzed to help determine the necessary requirements on the designed MW. These requirements were used to design the QoSM.

The resulting QoSM has support for both real-time requirements and high bandwidth communication. Its main functionality is scheduling of network traffic on an Ethernet bus. The scheduling algorithm is interchangeable but only one algorithm has been implemented in this thesis. The implemented algorithm is priority based.

Interesting areas for future work were also identified such as further development of the QoSM. This development includes QoS levels, different scheduling algorithms and linking the scheduling to other resources outside the network.
Preface

This master thesis is the last part of my education at KTH. It has been a long journey that started back in the year 2000.

During the thesis work I have received a lot of help and feedback from many different people. I would like to thank a few of them specifically here. Firstly my supervisor Tahir Naseer Qureshi for your help with my report. Your help with both the linguistics part as well as the contents of the report have been valuable. Secondly I’d like to thank the DySCAS group at KTH for your input on my work. It has been insightful and has forced me to prepare my work to present it. Barbro Claesson and Detlef Scholle at Enea have also been great support during the work and helped me organize the work better than I would have done on my own. Mikael Wånggren at ENEA has been great support with his in-depth knowledge of SHAPE.

I would also like to thank my fellow thesis workers for their input and help during the entire thesis work.
## Contents

1 Introduction .................................................. 15
  1.1 Objective .................................................. 15
  1.2 Outline .................................................... 15
  1.3 Method ..................................................... 16
  1.4 Limitations ................................................ 16

2 Theory .......................................................... 17
  2.1 Application requirements and theoretical basis .......... 17
    2.1.1 Gantry Controller .................................... 17
    2.1.2 Motion Controller .................................... 17
    2.1.3 Panel Controller ..................................... 18
    2.1.4 X-Ray Controller .................................... 18
    2.1.5 Image Reconstructor .................................. 19
    2.1.6 External Image Viewer ................................ 19
    2.1.7 Signals ................................................ 19
  2.2 Requirements .............................................. 19
    2.2.1 Resource Monitoring .................................. 19
    2.2.2 Data latency ......................................... 19
    2.2.3 Timeliness ........................................... 19
  2.3 Real-time Systems .......................................... 20
  2.4 Network Architecture ....................................... 20
    2.4.1 Ethernet .............................................. 20
    2.4.2 CAN .................................................. 21
    2.4.3 USB ................................................ 22
  2.5 QoS ........................................................ 22
    2.5.1 QoS Specification .................................... 22
    2.5.2 QoS Parameters ..................................... 23
    2.5.3 QoS Mapping ......................................... 23
    2.5.4 Cost of Service ..................................... 24
    2.5.5 Resource Reservation ................................. 24

by: Andreas Ziethén  
aziethen@gmail.com  
Kungliga Tekniska Högskolan  
Department of Mechatronics  
ENEA
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.6</td>
<td>QoS API</td>
<td>24</td>
</tr>
<tr>
<td>2.5.7</td>
<td>Admission Control</td>
<td>24</td>
</tr>
<tr>
<td>2.6</td>
<td>Scheduling</td>
<td>25</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Earliest Deadline First</td>
<td>25</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Least Laxity First</td>
<td>25</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Rate Monotonic Scheduling</td>
<td>26</td>
</tr>
<tr>
<td>2.6.4</td>
<td>Maximum Urgency First</td>
<td>26</td>
</tr>
<tr>
<td>2.6.5</td>
<td>Time Token</td>
<td>26</td>
</tr>
<tr>
<td>2.6.6</td>
<td>Time Division Multiple Access</td>
<td>26</td>
</tr>
<tr>
<td>2.6.7</td>
<td>Harmonization</td>
<td>26</td>
</tr>
<tr>
<td>2.7</td>
<td>Resource Management</td>
<td>27</td>
</tr>
<tr>
<td>2.7.1</td>
<td>HeiRAT</td>
<td>27</td>
</tr>
<tr>
<td>2.7.2</td>
<td>MASI Architecture</td>
<td>27</td>
</tr>
<tr>
<td>2.7.3</td>
<td>QoS-A</td>
<td>27</td>
</tr>
<tr>
<td>2.8</td>
<td>Middleware</td>
<td>28</td>
</tr>
<tr>
<td>2.8.1</td>
<td>TAO</td>
<td>29</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Hades</td>
<td>29</td>
</tr>
<tr>
<td>2.8.3</td>
<td>ARMADA</td>
<td>30</td>
</tr>
<tr>
<td>2.8.4</td>
<td>Kokyu</td>
<td>30</td>
</tr>
<tr>
<td>2.8.5</td>
<td>Distributed Resource Controller</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>Design</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>Problem Analysis</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Requirements</td>
<td>32</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Resource Monitoring</td>
<td>32</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Data latency</td>
<td>32</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Gateway Transparency</td>
<td>33</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Bandwidth Specification</td>
<td>33</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Jitter Specification</td>
<td>33</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Data Dependency</td>
<td>33</td>
</tr>
<tr>
<td>3.2.7</td>
<td>Other Requirements</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>QoSM</td>
<td>34</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Priority Levels</td>
<td>35</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Global QoSM</td>
<td>36</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Local QoSM</td>
<td>38</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Token</td>
<td>38</td>
</tr>
<tr>
<td>3.3.5</td>
<td>API</td>
<td>39</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Multiple Network Buses</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>Analysis of the proposed design</td>
<td>40</td>
</tr>
</tbody>
</table>
### 4 Implementation

4.1 OSE .................................................. 45
4.2 SHAPE ............................................... 45
4.3 Simulated Device ................................. 45
    4.3.1 Gantry Controller .......................... 46
    4.3.2 Motion Controller .......................... 46
    4.3.3 Panel Controller ............................ 46
    4.3.4 X-Ray Controller ........................... 46
    4.3.5 Image Reconstructor ......................... 46
    4.3.6 External Image Viewer ....................... 47
    4.3.7 QoSM ........................................ 47

### 5 Results

### 6 Conclusions

### 7 Future Work

7.0.8 Connection Management ..................... 59
7.0.9 Scheduling Algorithms ....................... 59
7.0.10 Cost of Service .............................. 59
7.0.11 CPU Scheduling .............................. 59
7.0.12 Preemptive Scheduling ...................... 60
7.0.13 Run-time Prioritized Rescheduling ........... 60
7.0.14 QoS Levels .................................. 60

### A Acronyms

by: Andreas Ziethén
aziethen@gmail.com
Kungliga Tekniska Högskolan
Department of Mechatronics
ENEA
List of Figures

2.1 Application Model ............................................................... 18
2.2 Ethernet Protocol ............................................................... 20
2.3 Ethernet Timing ................................................................. 20
2.4 Data Timing Ethernet ........................................................... 21
2.5 Control Timing Ethernet ......................................................... 21
2.6 CAN Protocol ................................................................. 21
2.7 CAN Timing ................................................................. 22
2.8 CAN Identifier ................................................................. 22
2.9 ACE QoS Class Design[12] ......................................................... 25
2.11 Connection Request[7] ......................................................... 28
2.12 TAO real-time ORB end-system architecture[26] ......................... 29

3.1 Dedicated Ethernet ............................................................... 32
3.2 QoSM ................................................................. 34
3.3 QoSM Interconnections .......................................................... 35
3.4 Priority Levels ................................................................. 35
3.5 Example Schedule .............................................................. 38
3.6 Token ................................................................. 39
3.7 Request Signal .............................................................. 39
3.8 Response Signal .............................................................. 40
3.9 Flow chart describing the Request Manager .................................. 41
3.10 Flow chart describing the scheduler ........................................ 42
3.11 Flow chart describing the Token Manager .................................. 43
3.12 Flow chart describing the local QoSM ...................................... 44

4.1 Demonstration platform ......................................................... 46
4.2 Implemented QoSM ............................................................ 47
4.3 Schematic figure of SHAPE ..................................................... 48
4.4 Motion controller flow chart. .................................................. 49
4.5  Panel controller flow chart. .................................................. 50
4.6  X-Ray controller flow chart. .................................................. 51
4.7  Image Reconstructor flow chart. ........................................... 52
4.8  External Image Viewer flow chart. ....................................... 53
List of Tables

3.1 Requirements .................................................. 34
Chapter 1

Introduction

Product development in the medical technology (Med Tech) environment is a lengthy process. The process of implementing and integrating new products is usually very complex and can take several years to complete. It is also very difficult to make small changes such as upgrading of software. This is due to safety regulations in the Med Tech environment and the complexity of systems.

In a distributed system the middleware (MW) is the software layer that allows applications to communicate with each other. Using a middleware in the Med Tech environment can shorten both development time and cost since only the upgrades need to be developed instead of an entirely new product. In the Med Tech environment, systems can have safety critical functions which raise the need for supporting real-time requirements in such a middleware. This thesis looks specifically at the resulting quality of service (QoS) needs of applications running on such a middleware. QoS is the concept of providing guarantees to applications.

1.1 Objective

The main goal of this thesis is to propose a design of a QoS Manager (QoSM) for the Meteor middleware and then implement this on a simulated Meteor middleware product. This manager should ensure that applications with real-time requirements can run on the middleware. Furthermore the specific requirements on a middleware from a quality-of-service (QoS) point-of-view that exist in a Med Tech environment have been examined. Middleware usage in the Med Tech environment as well as middleware support for real-time requirements in general is also examined.

Specifically this thesis will answer the following three questions:

- What are the existing MW solutions suitable for the Med Tech environment?
- What are the QoS requirements on a middleware in a Med Tech environment?
- How should a QoS manager be implemented in the Meteor middleware?

1.2 Outline

This report starts with a brief introduction of the problem addressed in this thesis and continues describing the theory that is relevant to the questions discussed. After this
chapter three describes how the theory is used to design the middleware and QoS manager. This is followed by another chapter describing how things were actually implemented. Then an evaluation of the result of the goals set out for this thesis is discussed in the chapter five. Thereafter chapter six describes conclusions resulting from the thesis work. Finally future work that should be performed in related areas of this thesis is discussed in the last chapter.

1.3 Method

The thesis work was divided into three separate phases. First a literature study was conducted where reports and books pertaining to the objective of this thesis were studied. In the second phase the requirements and QoSM were designed. This was done by examination of a typical Med Tech application as well as through a study of requirements on similar middleware, specifically DySCAS. The resulting requirements were used to design the QoSM. Lastly an implementation phase tested the designed QoSM.

The DySCAS group at KTH has been extensively involved in this thesis and have provided a lot of input regarding the design of the requirements and QoSM.

1.4 Limitations

The most important limitation of this thesis is the time constraint. There were only 20 weeks to finish this thesis which is why the specific device discussed in the thesis has limited functionality. The QoSM only deals with scheduling and bandwidth issues.
Chapter 2

Theory

This chapter describes the theory necessary for this master Thesis. The first section describes the target application followed by different requirements of the simulated device in this thesis. Section 3 describes real-time systems in general. Thereafter some basic QoS concepts are described. Basic scheduling theory is discussed in the section 5 which is followed by the description of different resource management architectures. This chapter concludes with a section describing other middleware and their implementation of QoS in networks.

2.1 Application requirements and theoretical basis

The application discussed in the thesis is a model of a med tech device used to control an x-ray device and store the image data received from it. It consists of six basic functional blocks. These communicate with each other in various ways. Data is sent to and from blocks and it is this data flow that is interesting to analyze from a QoS perspective. Figure 2.1 describes the functionality of the application. The arrows represent traffic between functional blocks. The type of arrow reflects the amount of bandwidth necessary for the communication where a thicker arrow indicates that the data flow consists of larger data sets.

2.1.1 Gantry Controller

The gantry controller’s (GC) mission is to control the gantry movement and to store its position. It receives control signals from the motion controller. It is also possible for the motion controller (MotC) to query the GC of its current position. The GC sends the gantry position to the image reconstructor (ImRC) when queried for it by the motion controller.

2.1.2 Motion Controller

With many different nodes in a system running the same application it is necessary to control the communication between the nodes. This is done by the MotC. It acts as an administrator of the system. Its main function is sending control signals to the other functional blocks at regular intervals. The control signals sent by the MotC are of negligible data size compared to the high bandwidth communication in the application. The
motion controller decides when the image reconstructor should provide the image viewer with image data. The MotC also queries the GC for the gantry position.

### 2.1.3 Panel Controller

In the system there is a need for acquiring data from the outside world. The panel controller (PanC) provides the system with this data. It is periodically ordered to release this data to the image reconstructor by the motion controller. The data size of the images sent from the PanC is much larger than the control signals typically in the range Mbit/s. Data is sent periodically six times per second.

### 2.1.4 X-Ray Controller

The x-ray controller (XRC) communicates with the outside world when requested to do so by the motion controller. It triggers the process which provides the panel controller with image data. It takes 40 ms before the data becomes available in the PanC.
2.1.5 Image Reconstructor

Before the image data from the panel controller can be viewed in the external image viewer (EIV) it passes through the image reconstructor where it is processed. The data sent from the ImRC is substantially larger than the data sent from the PanC.

2.1.6 External Image Viewer

When the image reconstructor is ordered to provide the EIV with image data it receives this data and displays it to the user of the EIV. This data requires very high bandwidth.

2.1.7 Signals

There are five fundamental types of signals in the application. Control signals that activate a functional block, a status signal identifying the XRC’s current state, an information signal detailing the current position of the gantry node and two types of image data. These signals are discussed in section 2.4.

2.2 Requirements

This section describes selected DySCAS requirements[13] that are related to QoS. There are other requirements as well but these are not of interest to this thesis.

2.2.1 Resource Monitoring

"The system resource usage shall be monitored."

To be able to guarantee that deadlines are met it is necessary to monitor resources such as the bandwidth. It is impossible to detect a failure in a resource if resources are not monitored. This is also a requirement to be able to administrate system resources such as network buses.

2.2.2 Data latency

"The latency to deliver data from provider to subscriber must be less than a specified time."

When two nodes in an embedded system need to communicate it is necessary to be able to guarantee that data is not delayed longer than the application running on the nodes allows. This requires monitoring and administration of the network in the system.

2.2.3 Timeliness

"The services of the DySCAS middleware shall be able to produce a result within a given time limit."

In a system with hard real time constraints it is necessary to guarantee that a result is ready to be used when it is needed. This is necessary even in services that do not have hard real time constraints since the service depending on the result may have such contraints. It must be possible for the user to define a maximum acceptable delay.
2.3 Real-time Systems

It is hard to define exactly what a real-time system[23] is and there exist a lot of variety in its definitions. In this thesis a system is considered to be real time if at least some of the executions in it have time-based deadlines that will affect the system if they are not met. Real-time deadlines can be either hard or soft. Hard deadlines must always be met or the system will in some way fail, whereas soft deadlines can be missed occasionally but missing it will affect the quality of the system’s performance.

Real-time systems can either be single processor or multi processor systems. If the multi processor system is distributed it is referred to as a distributed real-time system. Many of the techniques used in multitasking systems can be applied to distributed systems, e.g. scheduling. In a distributed system each processor can be treated as a task in a multitasking system[19].

2.4 Network Architecture

There are many different kinds of networks, but this thesis only deals with Ethernet and CAN which are common in Med Tech devices.

2.4.1 Ethernet

Ethernet[18] is a protocol for communication on a shared bus. The basic functionality of the protocol is called carrier sense multiple access with collision detection (CSMA/CD). When a sender wants to use the ethernet bus it must first listen to make sure the bus is not currently in use by someone else. If the bus is not in use the sender starts sending data. If two senders try to send data at the same time a collision occurs and the data sent is discarded and resent. Before resending a delay of $0 \leq r < 2^k$ time slots is required where $k = \min(n, 10)$ and $r$ is a uniformly distributed integer and $n$ is a retransmission counter. A time slot is the time required to send a frame. There are many different standards for ethernet supporting different levels of bandwidth. Typically it runs at 10Mbps or 100 Mbps, but it can also run much faster at 10 Gbps. Figure 2.2 describes the ethernet data format.

![Ethernet Protocol](image)

Figure 2.2: Ethernet Protocol

Between two frames the protocol requires an inter-packet gap of 96 bit times. The packet gap allows the network interfaces time to prepare for the next packet. This makes the timing for the data flow look as in figure 2.3.

![Ethernet Timing](image)

Figure 2.3: Ethernet Timing
When the image data is transmitted on the ethernet bus it looks as in figure 2.4. The protocol overhead added is dependent on the instantiation layer of the middleware, described in 4.2. In total, the overhead and the image data contained in an ethernet frame is limited to 1500 bytes of data.

![Figure 2.4: Data Timing Ethernet](image)

The control signal is translated from its CAN format to the ethernet format described in figure 2.5 by the gateway if the receiving node is on the ethernet bus. This routing is handled by the gateway and data intended for the CAN bus is not sent to the ethernet bus.

![Figure 2.5: Control Timing Ethernet](image)

### 2.4.2 CAN

CAN[16] is also a protocol for communication on a shared bus. Its basic protocol is carrier sense multiple access with bitwise arbitration (CSMA/BA), similar to that which is used in ethernet but with arbitration. Figure 2.6 describes the CAN data frame. If two senders try to use the CAN bus simultaneously their identifiers are used to arbitrate the communication. Each bit of the identifier is sent on the bus and when a sender with lower priority notices a bit on the bus that differs from its own it stops sending. At the end of the arbitration period only the highest priority sender will still be sending. At most CAN supports 1 Mbps data throughput which is significantly less than ethernet. Between two frames the protocol requires an inter-packet gap of 3 bits. This makes the timing for the data flow look as in figure 2.7.

![Figure 2.6: CAN Protocol](image)

The identifier used in this system is described in figure 2.8.
The first bit of the identifier, priority, determines if the signal is long or short. Long signals are larger than 8 bytes and must be split into several small CAN data frames. The first byte of the data field contains the size of the signal for large signals. After this the signal ID identifies the type of signal. This field is used to identify what kind of data is in the frame. The following 4 fields identify the sending and receiving node and process. Lastly the LE bit defines if the frame is in little endian or big endian format.

### 2.4.3 USB

USB uses four types of transfers\cite{9}. These are control transfers, isochronous transfers, interrupt transfers and bulk transfers. Control transfers are used for bursty non-periodic command status operations. Isochronous transfers are used for periodic continuous communication. Interrupt transfers are used for low frequency and bounded latency communications. Bulk transfers are used for non-periodic large transfers that can use any available bandwidth but does not have delay or bandwidth requirements. Bulk transfers are only allowed when no other transfer types are sent. Basically bulk transfers make use of left-over bandwidth.

### 2.5 QoS

Real-time requirements of a distributed multimedia system are discussed in \cite{5}. QoS is the concept of providing guarantees to applications. E.g. an application may require a maximum delay on execution. Without proper QoS it is not possible to provide end-to-end guarantees for applications. This section describes QoS from different perspectives including specification and parameters as well as reservation of resources and admission control.

#### 2.5.1 QoS Specification

Specification of QoS requirements can be done on different layers of an architecture. Either on a low level where delays or other characteristics of a communication are defined or on a higher level through more esoteric concepts such as QoS levels where an application may be able to run in different modes depending on the available resources.
Load Description

Part of the QoS specification in a network environment is the description of the intended work load. This can be done either through QoS characteristics[5], through some high level esoteric description of the resources needed or by direct description of the resources actually needed. E.g. in the high level description case a periodic transmission could be described as a specific amount of data and a period of the transmission intervals. In the direct description all the transmissions would be individually described.

2.5.2 QoS Parameters

When defining a desired QoS level it is necessary to have quantifiable and measurable parameters. This subsection defines some QoS parameters[21] taken from a group communication perspective.

Throughput

Throughput is a measure of data rate, typically data per time. In this thesis throughput is measured as bits per second (bps). In communication systems there are local and global limitations on throughput. E.g. a node in a system may be able to generate data at a higher rate than the communication links in the system can deliver. This imposes a throughput limit on the generating node. Limiting the generation of data is an example of congestion control.

Transit Delay

Transit delay is the time between data being sent and received by the intended destination. This is the time data spends in the network system handling the communication.

Jitter

Jitter[8] is the difference between the longest and the shortest delay in a system. The concept of jitter requires that data is somehow related. I.e. that the same sender and receiver are involved in the communication and that there is some contextual relation between the data sent.

Loss

Loss is a measure of the communication failures of a data transmission. This sometimes means that data needs to be resent and sometimes losing data is acceptable.

2.5.3 QoS Mapping

An important aspect of QoS is QoS mapping. This is the process of interpreting application level QoS requirements and mapping them to the underlying hardware architecture. Without proper QoS mapping it is not possible to provide end-to-end guarantees for applications. For instance a deadline specified by an application can be mapped to underlying network resources. These resources can be reserved to meet the application requests.
2.5.4 Cost of Service

Another important aspect of QoS that Aurrecoechea[5] mentions is the cost of service. If there is no price for requesting high performance the user has no reason to specify anything other than the highest level of service. The idea is to punish an application that incorrectly requests too much bandwidth by e.g. providing lower bandwidth subsequently. This gives the application programmer an incentive to correctly estimate the bandwidth necessary for the application.

2.5.5 Resource Reservation

To guarantee the availability of resources when an application needs to use them, resource reservations[6] can be made. For instance if an application is going to use a network bus in an administrated system it could register its transmission with an administrator beforehand. Another way of reserving resources is making reservations on the intended path beforehand such as in the RSVP[30] protocol. Campbell[6] also mentions that it is necessary to manage resources to ensure that QoS is sustained. Using resource reservation has the downside of creating extra delays for a communication, i.e. the time to set up the reservation.

2.5.6 QoS API

The QoS Application Programmer’s Interface (API) is used for describing the QoS requirements of an application. The purpose of the API is to make the QoS protocols easier to use for the programmer. An example of a QoS API is described in [24]. It is based on profiles and sessions. Before an application can use a resource it must first bind itself to a session. When the application requests a session it provides a profile detailing its actual QoS requirements. The API also has support for managing profiles and requesting them from a profile server.

Another API is described in [12]. It is object orientated and built for the common object request broker architecture (CORBA) platform. This is also a session-based API but built for use in a middleware. QoS sessions are created in the QoS session factory and later bound to a socket. This relieves the application of determining what protocol is used for the communication. The API is shown in figure 2.9.

Kachroo[12] mentions the challenges of designing a QoS API. The API should provide portability and be platform independent so the user can create an application regardless of the underlying hardware. Providing extensibility is also important so the API can be expanded with future QoS implementations. Parameters for specifying QoS requirements should be identified. It is also important that the user can query the system for current QoS of a session.

2.5.7 Admission Control

If resource reservation is used as part of the QoS architecture then some form of admission control is necessary to handle the reservations[5]. If several applications want to use the same resource simultaneously then only one can be granted access to it. This means the applications are responsible for the error handling if resources cannot be granted when they are requested. The benefit of using admission control is that applications can handle errors prematurely and avoid potential disasters.
2.6 Scheduling

This section describes different scheduling algorithms that can be implemented to guarantee real-time constraints. Even though these are basically scheduling algorithms for process tasks, the theory behind the algorithms is also interesting for network scheduling. It is important to distinguish between network based QoS and end-to-end application QoS, where the former only applies to the network structure and the latter concerns the real-time requirements that are specified in an application.

2.6.1 Earliest Deadline First

EDF[20] is an optimal scheduling algorithm. The idea of EDF scheduling is to let the process with the shortest deadline execute first. This means the load of the scheduled resource will be 100% before the scheduling algorithm fails. EDF requires that all tasks are periodic with constant period and that all tasks are independent of each other. For instance if there are two tasks A and B then task B can commence regardless of whether task A has completed or just been interrupted.

2.6.2 Least Laxity First

LLF[23] scheduling is very similar to EDF scheduling. The difference between LLF and EDF is that LLF scheduling uses the process laxity instead of the deadline for calculations. The laxity of the task is the estimated time left after the process has completed before its deadline. This requires knowledge of the execution time of a task. A task’s laxity is not updated while it is being executed so this can lead to a scheduling deadlock where
context switches take up most of the execution time.

2.6.3 Rate Monotonic Scheduling

Rate Monotonic (RM)[23] scheduling is based on a simple principle. If a static schedule for periodic tasks exists it can be implemented by letting the task with the shortest period run first. This is proven in [20]. RM scheduling requires that all tasks are periodic and as such is not possible to use in a dynamic environment. The RM algorithm also requires that all tasks are independent of each other. Liu and Layland proved that for a large set of tasks the maximum guaranteed processor utilization allowed by this algorithm is \( \ln 2 \).

2.6.4 Maximum Urgency First

Maximum Urgency First[27] (MUF) is a combination of several other scheduling algorithms including RM, LLF and EDF. MUF uses a combination of three separate priority levels. These are criticality priority, dynamic priority and user priority. The tasks with critical priority are the tasks that are guaranteed to always be handled by the scheduler regardless of utilization. Dynamic priority is closely related to LLF laxity. Tasks are ordered in laxity order to define the dynamic priority. Lastly a user priority may be defined to every task. Whenever a new task enters the ready-queue the queue is rescheduled. When the scheduling is done the scheduler first looks at the criticality priority. Critical tasks are handled first. If several tasks share the critical priority the dynamic priority is used to select a winner. If two tasks also have the same dynamic priority the user priority is used to select the task to run. If it is still not possible to select a unique winner the tasks are run on a first-come first-serve basis.

2.6.5 Time Token

A time token[25] functions much like a semaphore. Before a process is allowed to send data it must receive a token permitting it to do so. The token has a hold time, which is the time a process is allowed to keep it, and a rotation time which is the time until all nodes have held it. If the process holding the token crashes the token disappears and the network becomes locked. This is usually solved using a silence time. If a node detects that the network bus is unused for longer than a predetermined time a new token is created to replace the lost token.

2.6.6 Time Division Multiple Access

In the TDMA[17] algorithm the transmitting time is divided into small slots. These are then allocated to different nodes in the system and they are only allowed to send data during their time slots. The distribution of time slots can be done in different ways to prioritize certain nodes or simply to allocate more bandwidth to specific nodes. TDMA differs from a time token in that nothing is sent to allow the process to send data, but rather clock synchronization ensures that nodes stay within their defined time slots.

2.6.7 Harmonization

Harmonization of periodic tasks is discussed in [15]. Periodic tasks that are not harmonized suffer from limited utilization. By adjusting the frame size to multiples of each other it is possible to reach a utilization of 1. The shortest period that is common for all the
tasks involved in the harmonization is referred to as the least common multiple (LCM). Using this period provides a significantly better utilization than \( \ln 2 \) which is the case in RM scheduling without harmonization.

## 2.7 Resource Management

Resource management in a distributed environment is discussed in [6] and [22]. The basic principle is to keep track of resources in a distributed system so that applications can make requests to the manager which then admits or rejects the request and reserves appropriate resources for the application. The following subsections describe current resource management frameworks, namely the Heidelberg Resource Administration Technique (HeiRAT), the MASI Application QoS Manager (AQOSM) and the Lancaster QoS Architecture (QoS-A).

### 2.7.1 HeiRAT

HeiRAT[28] is a framework for handling QoS guarantees in a distributed multimedia system. It takes the entire system into consideration, not just the network, to provide end-to-end guarantees for applications. First there is a start-up phase where applications supply their QoS requests and requirements. The system then performs a throughput test and QoS calculation. This is referred to as the negotiation phase. If the requirements can be met the request is accepted and reservations are made. Otherwise the request is rejected and the application receives a message. When the QoS calculation is performed for a request that requires several nodes the request is propagated as far through the system as the request can be met. If the throughput test fails before reaching the final destination the error message is propagated back through the system and previously registered resources are unregistered. After reservation is done a resource scheduling is performed to ensure that all QoS requirements are fulfilled.

HeiRAT provides support for both guaranteed and statistical requirements. The former is used by applications that have hard real-time requirements and the latter by applications with soft requirements.

### 2.7.2 MASI Architecture

The MASI architecture[10] is a layer based approach to QoS management. An application communicates with the top level QoS manager called the AQoS Manager providing the requirements of the application. The AQoS Manager then calculates a communication profile for the application requirements and sends this to the connection manager called ACM. The ACM then forwards the request to the lower level layers of the communication management. The profile progresses to the network level where a QoS result is calculated which is then sent back up through the different levels and finally to the application from the AQoS Manager. Profiles contain information about parameters specific for a type of application such as jitter, loss and delay. Figure 2.10 (a) describes the initialization process and figure 2.10 (b) describes profiles used in the architecture.

### 2.7.3 QoS-A

The QoS-A[7] uses a concept called flows to handle network traffic. A flow is a description of what a communication will look like including frame size and frame rate as well
as a description of the bursty behavior of a communication. Before a flow commences a connection request is sent specifying the sender and receiver as well as the flow. The request is interpreted by the QoS mapping service and redirected to the connection manager of the specific resource which performs resource allocation and admission testing. The connection manager communicates with the system resource managers to ensure that all resources are available. If any of the managers reject the request the initial connection request is rejected. Otherwise resources are reserved and the request is granted. Figure 2.11 describes the connection request.

2.8 Middleware

There are many different middlewares in the market, but this thesis only deals with MW’s that have support for real-time requirements or are in other ways fundamental to Meteor.
This section only deals with the aspects of the middlewares that relate to QoS in networks.

2.8.1 TAO

TAO[26] is an implementation of CORBA. The real-time guarantees of the network in TAO are provided by several components. The base of it is a high-speed network adapter. On top of this there is a run-time scheduler that ensures that applications meet their real-time demands. This can use different scheduling algorithms for doing so, e.g. rate monotonic scheduling. To ensure that the scheduler can supply applications with their required resources the admission controller accepts or refuses real-time requests from applications. If a request is accepted it is guaranteed to be scheduled on time. Figure 2.12 describes the TAO real-time ORB end-system architecture.

![Figure 2.12: TAO real-time ORB end-system architecture][26]

2.8.2 Hades

The HADES[4] middleware supports three different protocols for multicasting. These are time-bounded basic multicast protocol, time-bounded atomic multicast protocol and time-bounded causal multicast protocol. These are all similar protocols but the latter two are fault-tolerant. TDMA is the strategy used in the protocols for communication. The only difference between the atomic and the causal protocol is that the causal delivers messages in temporal order. TDMA is described in section 2.6.6.

2.8.3 ARMADA

ARMADA[11] uses a communication service called RTCAST[3] to support multicasting. It performs the multicasting by passing a token around the system permitting only the holder to broadcast. The holder then passes the token to the next user. RTCAST guarantees real-time requirements through another protocol called ACSA. It keeps a register of all periodic messages that have real-time requirements as well as aperiodic messages that have not been sent already. Before it allows another application with real time guarantees

---

by: Andreas Ziethén
aziethen@gmail.com
Kungliga Tekniska Högskolan
Department of Mechatronics
ENEA
to communicate, it performs schedulability analysis to determine whether it is possible to guarantee the new application requirements.

2.8.4 Kokyu

Kokyu[14] is a middleware scheduling framework designed to provide flexible scheduling for middlewares, e.g. it supports TAO. It is possible to let the framework decide on an optimal scheduling algorithm for an application based on the knowledge of periodicity in the application. This relieves the application programmer from trying to optimize the scheduling algorithm for a specific application and platform.

2.8.5 Distributed Resource Controller

Distributed Resource Controller (DRC)[29] is a middleware for mapping QoS specifications end-to-end. The MW is responsible for mapping application level requirements to available network resources. An application sends a QoS request to the DRC Service Manager (DM). The DM then forwards the request to the appropriate DRC Resource Agent (DA) which queries the resource layer for available resources. If resources are available the request is mapped to the specific resource layer QoS specification and resources are reserved. The result of the request is then sent back through the middleware layer to the application.
Chapter 3

Design

This chapter describes the design work performed in this master thesis. The first section discusses the problem. The second section describes the requirements of the Meteor middleware. Lastly the design of the QoSM and its components are described.

3.1 Problem Analysis

In section 2.1 a typical Med Tech application that has timing and bandwidth requirements was described. The model itself does not specify a hardware architecture and neither should the middleware be hardware specific. However it is of interest to not make it impossible to tailor the hardware to the specific application. E.g. the application discussed in this thesis could run with all functional blocks acting as ECUs on one shared bus. It would also be possible to use more than one network bus i.e. the blocks with high bandwidth requirements could have a dedicated network connection specifically for this communication while other blocks make use of e.g. a CAN bus. This is illustrated in figure 3.1.

The use of several network buses in general and two different types of buses specifically introduces another problem. It should be possible for the application programmer to construct an application without any knowledge of underlying hardware architecture like but at the same time to implement hardware optimizations where applicable. This also means that the middleware needs to keep track of where a specific block of the application is executed. It is important that the middleware manages to be generalized enough to serve its purpose but at the same time be specific enough that it can be used for optimization purposes. This will affect the levels of details in the following section on requirements.

It is important that the middleware allows the system designer freedom to choose between different ways of implementing and solving a specific problem. The middleware must also provide a suitable interface for the programmer such that its functionality can be utilized efficiently without requiring tailor-made optimizations on hardware level for each application.

The middleware support for QoS should also have dynamic properties that make it possible to expand it to a dynamic environment in the future. This allows the user of the MW more freedom when designing a system. Another benefit of making a more dynamic MW is that this allows the MW to be ported to other areas than the Med Tech environment with similar but not identical requirements.
3.2 Requirements

To verify that the Meteor middleware functions as desired and fulfills the needed requirements of the application programmer it is necessary to specify requirements on the middleware. These are largely based on the DySCAS requirements described in section 2.2 and the application model in section 2.1. The previous section 3.1 also discusses issues worth considering outside the model in section 2.1.

3.2.1 Resource Monitoring

"The system resource usage shall be monitored."

To be able to guarantee that deadlines are met it is necessary to monitor resources such as the bandwidth. If resources are not monitored it is impossible to detect a failure in a resource. This is also a requirement to be able to administrate system resources such as network buses. Resource monitoring should also keep track of system specific parameters such as power and translate application level requirements to system requirements. This enables the middleware to perform resource optimizations on the system level.

3.2.2 Data latency

"The latency to deliver data from provider to subscriber must be less than a specified time."

When two nodes in an embedded system need to communicate it is necessary to be able to guarantee that data is not delayed longer than allowed by the application running on the nodes. This requires monitoring and administration of the network in the system.
3.2.3 Gateway Transparency

"The gateway must transparently connect network buses of different types."

In a distributed system with several network buses connected through gateways it is necessary to be able to send messages between nodes on different buses. It is also necessary to be able to separate parts of the network from this transparency. This can be done by a gateway. Specifically the gateway must transform message headers between different protocols without corrupting the data inside the message. This relieves the application programmer from knowing what the underlying network structure looks like. The gateway must also restrict access to dedicated network buses.

3.2.4 Bandwidth Specification

"The necessary bandwidth for communication must be possible to specify on application level."

Applications may have bandwidth requirements that are soft rather than hard. E.g. an application may wish to display live feed from a camera but it is not necessary for the quality to be perfect. It is desirable to specify a bandwidth for this communication without requiring that all of the data is delivered within a specified time as in Req-3 described in subsection 3.2.2.

3.2.5 Jitter Specification

"The jitter of a communication must not exceed the amount specified by an application."

Communications may have jitter requirements even though the entire communication does not have a delivery requirement. E.g. an application may wish to display video data to the user and requires a specified jitter to do this. It should be possible for the application programmer to define a maximum allowed time for jitter in the communication.

3.2.6 Data Dependency

"The Meteor middleware must allow an application to specify timing dependencies between separate communications."

Applications may need several data transfers that have different timing requirements depending on each other, i.e. the second transfer needs to be sent 100 ms after the first has been delivered. This creates a data dependency that the QoSM needs to support.

3.2.7 Other Requirements

There are other requirements which are not handled in detail in this thesis but are still relevant to the Meteor middleware. These are largely based on the DySCAS requirements specified in [13] and list in 3.1.
CHAPTER 3. DESIGN

### Detection of node malfunction
"The Meteor middleware shall detect if a node is not functioning properly."

### Application execution control
"The system shall be able to prevent an application from executing."

### Application constraints adherence
"The system must respect any constraints imposed on the software allocation by the designer, e.g. bandwidth allocation."

### Available startup ECUs
"The system must be able to detect available ECUs during start-up."

### Setting up remote communication
"The Meteor middleware shall be able to establish a communication link with a remote entity or device."

### Generate error codes
"The Meteor middleware shall generate error codes for malfunctioning components."

### Switching on and off hardware
"The Meteor middleware shall be able to switch on and off hardware."

### Timeliness
"The services of the Meteor middleware shall be able to produce a result within a given time limit."

### Network Architecture
"The Meteor middleware must keep track of the network architecture in the system."

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detection of node malfunction</td>
<td>&quot;The Meteor middleware shall detect if a node is not functioning properly.&quot;</td>
</tr>
<tr>
<td>Application execution control</td>
<td>&quot;The system shall be able to prevent an application from executing.&quot;</td>
</tr>
<tr>
<td>Application constraints adherence</td>
<td>&quot;The system must respect any constraints imposed on the software allocation by the designer, e.g. bandwidth allocation.&quot;</td>
</tr>
<tr>
<td>Available startup ECUs</td>
<td>&quot;The system must be able to detect available ECUs during start-up.&quot;</td>
</tr>
<tr>
<td>Setting up remote communication</td>
<td>&quot;The Meteor middleware shall be able to establish a communication link with a remote entity or device.&quot;</td>
</tr>
<tr>
<td>Generate error codes</td>
<td>&quot;The Meteor middleware shall generate error codes for malfunctioning components.&quot;</td>
</tr>
<tr>
<td>Switching on and off hardware</td>
<td>&quot;The Meteor middleware shall be able to switch on and off hardware.&quot;</td>
</tr>
<tr>
<td>Timeliness</td>
<td>&quot;The services of the Meteor middleware shall be able to produce a result within a given time limit.&quot;</td>
</tr>
<tr>
<td>Network Architecture</td>
<td>&quot;The Meteor middleware must keep track of the network architecture in the system.&quot;</td>
</tr>
</tbody>
</table>

Table 3.1: Requirements

### 3.3 QoSM

The QoSM is the service responsible for providing QoS guarantees in the Meteor MW. It also monitors system resource usage and calculates necessary schedules for the bandwidth in the system. To do this it keeps track of resources available in the system. The components of the QoSM are described in this section as well as the API for communicating with the QoSM. Figure 3.3 describes how the different components are interconnected. Every node has a QoSM, either a local or the global. There can only be one Global Quality of Service Manager (GQoSM) in the middleware. The applications running on a node communicate with the local QoSM which communicates with the GQoSM through the node’s network interface.

Figure 3.2 describes the GQoSM.

![Figure 3.2: QoSM](image)

The design of the QoSM is suitable for an ethernet architecture, but not for CAN. This is due to the data size of the administrative functions in the QoSM. This is discussed in detail in section 3.3.2.
3.3.1 Priority Levels

There are 64 different priority levels in the QoSM. The range is from 0 to 63. The lowest level is the highest priority. Priority levels are divided into three different groups. Level 0 is only for use within the request manager (ReqM). When a request has been accepted its priority is set to 0 meaning it will always have priority over unscheduled requests.

Levels 1 to 31 are reserved for the middleware services. It is necessary for middleware services to also register their transfer requests and to ensure that middleware services can be prioritized among each other there is more than one level available for them. The last group of priority levels 32 to 63 are reserved for application level services. These can be used freely by the application designer. Priority levels are schematically depicted in figure 3.4.

The local QoSM is responsible for supervising requests made by applications and the middleware to ensure that priority level restrictions are adhered to. This provides supervision of the system to the middleware.
3.3.2 Global QoSM

The GQoSM runs only on one network node. It is responsible for administrating the network traffic on the local bus. It receives bandwidth requests and accepts or denies them. It consists of three separate processes which are described in detail in this section. They are the ReqM, network scheduler and token manager.

ReqM

The ReqM keeps track of all the applications’ requests for network bandwidth including the sender and receiver of a transfer. It continuously checks for incoming requests. These are stored in a list of unscheduled requests. Unscheduled requests are forwarded to the scheduler upon receiving a heartbeat.

If the ReqM receives a returned request from the scheduler it checks if the request was scheduled and prepares a response message for the requesting application. If the request was accepted its priority level is decreased to guarantee level 0 and it is added to the unscheduled requests list. Prioritizing accepted requests ensures that applications meet their bandwidth requirements as discussed in 3.2.4.

An overview of the ReqM is shown in figure 3.9.

Scheduler

The scheduler calculates the network usage schedule. It is responsible for the requirements of data latency, data dependency and jitter. It provides data latency guarantees by checking the maximum deadline for a request when scheduling is done. The specified deadline is converted into allowed time slots taking into account the time since the request was made. Similarly the minimum latency is used to specify data dependency guarantees. Lastly jitter specification is adhered to through allocation of allowed time slots in proximity of the time slot marked by the deadline.

To ensure that hard real-time requirements are fulfilled, the scheduler uses a TDMA approach for network administration. A solution based on collision detection and retransmission could lead to missed deadlines and unwanted program behavior therefore the TDMA solution is superior.

Scheduling is based on dynamic heartbeat periods where the schedule for one period is calculated during the previous period. This makes it possible to admit new requests during run-time. It is possible to implement different scheduling algorithms in the scheduler. Scheduling techniques were described in section 2.6 including MUF and EDF. The scheduler uses a variant of MUF scheduling where only priority and deadlines are considered.

Before the actual scheduling is done the scheduler needs to calculate the scope of the schedule. This is done dynamically to improve the utilization of the system by harmonizing the schedule as described in section 2.6.7. The schedule is not always perfectly harmonized as the deadlines of requests may limit the maximum scope of the schedule. Also if there are few requests the scope may be greater than the LCM of the requests to ensure that scheduling overhead does not lower utilization significantly. In this case the schedule is extended to a multiple of the LCM above the minimum heartbeat period. To help the system designer bound the scope of the schedule there are two system parameters available to set the boundaries of the schedule. This can also be used to ensure that scheduling of the next period has time to complete before it starts.

As described in 3.3.2 the scheduler does not immediately receive requests when an application sends them. They first go through the LQoSM and wait for the end of the current
heartbeat period before progressing to the ReqM. Then the ReqM waits for its update signal before sending them to the scheduler. The ReqM receives an update signal halfway through a heartbeat. The scheduler waits for the heartbeat before calculating the schedule. Heartbeats for the scheduler and update signals for the ReqM are phase shifted since requests do not exist in the request manager while they are being scheduled.

Following this the schedule progresses to the token manager and is sent with the next heartbeat. This means that in total a request needs to wait a maximum of two heartbeat periods before it is scheduled. At minimum the request needs to wait one full heartbeat plus the registration time. Therefore it is necessary to alter the maximum and minimum allowed heartbeat period in the system based on the required response time from the scheduling sub-system.

Scheduling Algorithm

The scheduling starts with the creation of instances of periodic requests whose period is shorter than the scope of the schedule or requests with deadlines inside the scope. The jitter specification is used to define the allowable time slots for these instantiated requests. If there are requests with periods greater than the scope of the schedule their deadlines are adjusted so they will be properly scheduled in subsequent periods of the schedule. After the instances have been created they are sorted by priority first and deadline last. This is done to ensure that prioritized requests are first scheduled.

After this the scheduler creates the basic schedule with administrative overhead in it. This includes time for sending the schedule and also for handling requests and request responses. Then the list of request instances is handled and each instance is inserted into the schedule. If it is not possible to insert an instance into the schedule this is because a higher priority request has already been scheduled in all allowable time slots. The scheduler will try to move the blocking slots. If they cannot be moved the scheduler will register the instance that failed, remove all previously scheduled instances of that request and return the request as failed.

The scheduler will only try to move blocking slots one level deep. It will not try to move slots that are blocking the blocking slots. This is a design choice to keep the scheduling time deterministic. It would be possible to connect the allowed time for calculating the schedule with the minimum heartbeat period in the system. This could provide end-to-end guarantees for applications with timing deadlines.

Once all instantiated requests have been scheduled remaining unused slots are left as bulk slots similar to bulk transfers in the USB protocol described in 2.4.3. These can be used by an application that has a lot of data to send without deadlines. There can only be one bulk service in a system since otherwise there might be collisions on the network between different nodes and the underlying network controllers may try to retransmit the data outside the bulk time slots. It would be possible to implement a token passing system for the bulk time slots.

Non-periodic requests that should be scheduled in subsequent heartbeat periods are checked against the existing schedule to perform admission control of the requests. If they are allowed their accepted status is set to accepted.

An example of what a schedule looks like is presented in figure 3.5.

When all instances have been handled the schedule calculation is complete. The schedule is then sent to the token manager. The functionality of the scheduler is described in figure 3.10.
CHAPTER 3. DESIGN

Figure 3.5: Example Schedule

Token Manager

The token manager receives the schedule from the scheduler. The schedule is converted to a token list. If two subsequent time slots belong to the same application the size of the token is increased and only one token is kept in the token list. This reduces the amount of data used in the token list. When the token manager receives an end of schedule signal it knows that the entire schedule is in the token list and waits for the administration cycle allowing it to distribute the tokens to the applications. The token manager also keeps track of when the ReqM should send its request responses and sends a token to the ReqM when it is time.

Since the scope of the schedule determines the number of time slots in a schedule and thus the number of tokens it may be necessary to alter the minimum and maximum size of the schedule’s scope depending on the type of network bus in the system. In case of CAN network tokens can be split into two parts due to the large size of tokens as compared to a CAN data frame. This is why this design is better suited for an ethernet architecture. It is however implementable on a CAN architecture as well.

The functionality of the token manager is described in figure 3.11.

3.3.3 Local QoSM

Every application node has a local QoSM. It has three responsibilities. Firstly it handles local request registrations and ensures that requests are only sent on the network to the GQoSM during the registration phase. This also includes translating delays specified in ms to time slots used in scheduling. Secondly it receives the schedule and keeps the tokens for the local applications and services. Lastly it uses the tokens to request data from the local applications which it keeps in queues until it is time to send the data. The LQoSM creates a queue for every service that makes a bandwidth request. The queue is used to ensure data is ready to be sent in the allocated time slot. By sending data through the local QoSM the local resource monitoring requirement is handled.

The functionality of the local QoSM is described in figure 3.12.

3.3.4 Token

A token contains information about its data size. What service it belongs to and its offset in time slots to the heartbeat containing it.
The token is described in figure 3.6.

![Figure 3.6: Token](image)

### 3.3.5 API

The purpose of the API is to establish a method for the application programmer to specify the bandwidth requirements of an application. It is also necessary for the API to support confirmation of requests so the application can plan its actions accordingly. If a request is denied it may be necessary to send a new request later or to simply lower the amount of bandwidth requested.

**Transfer Request**

```c
request_transfer(U32 MinDelay, U32 MaxDelay, U16 Jitter, U16 Size, U16 Period, U8 Prio, U16 Instances)
```

Transfer requests are sent by an application in a setup phase when determining the amount of bandwidth the system can provide it with. If a request is denied the application can send another request for less bandwidth. This provides the application with a form of negotiation.

When a transfer request is sent to the QoS the application specifies the maximum delay until the transfer must complete as well as the maximum delay until it can start. Time limits are specified in ms on application level. The reason for specifying a minimum delay is to enable applications to pre-negotiate the needed bandwidth. Applications can also specify a maximum allowed jitter of a periodic transfer. Included in the transfer specification is the size in kilobytes of the transfer and the period of future identical transfers. If a transfer is not periodic the period is set to 0. Lastly a request holds a priority number allowing the application programmer to specify the relative importance of communications in a system.

A request consists of 22 bytes of data. Since an Ethernet frame can hold up to 1500 bytes of data a total of 68 requests would fit in one frame. This means a node will usually not need more than one ethernet frame to send its requests to the ReqM. A CAN bus however would need four frames for each request making it much slower.

The request is described in figure 3.7.

![Figure 3.7: Request Signal](image)
Send Transfer Response

send_transfer_response(U8 DestinationService, boolean Accepted)

This function is only used by the QoSM. When the QoSM receives a transfer request it tries to calculate a schedule including the request. After this the application is notified of the result using the send transfer response function. A response consists of six bytes of data. The response is described in figure 3.8.

<table>
<thead>
<tr>
<th>SIG NO</th>
<th>1 bit Accepted</th>
<th>1 byte Owner</th>
</tr>
</thead>
</table>

Figure 3.8: Response Signal

Send Token

send_token(U8 DestinationService, struct Token)

This function is called by the LQoSM when it sends a token to a service.

3.3.6 Multiple Network Buses

The QoSM described in this thesis only discusses the use of one network bus. It would be possible to use the same basic structure when other network buses are connected through a gateway. The gateway would then register for transfers when data is sent between the buses. Every network would need to implement its own QoSM to ensure that system wide guarantees could be made. This would ensure the GW transparency. The GW could register for periodic transfers to achieve faster response times. It could also put several CAN frames in the same ethernet frame acting as a buffer. The frame would then be broadcasted and analyzed by each node on the ethernet bus to find data intended for the specific node.

3.4 Analysis of the proposed design

The design presented in this chapter has some interesting features. It performs admission control where applications that have been allowed will never be removed. This has both positive and negative effects. The benefit is that applications are guaranteed to receive the requested bandwidth for their entire execution once their request has been accepted. This provides reliability in the system. The downside is that higher prioritized requests that arrive later may be rejected in spite of their higher priority.

Another interesting aspect is the timing of responses from the admission control. The system parameters allow the system designer to control the maximum response time of any request in the system. These parameters also give the system designer freedom to control the overhead in the system by specifying the minimum heartbeat period.

The design does not provide any penalties for requesting more bandwidth than necessary and this may result in a high amount of wasted bandwidth when applications request bandwidth that is left unused.
Request Manager

Figure 3.9: Flow chart describing the Request Manager
Figure 3.10: Flow chart describing the scheduler
Figure 3.11: Flow chart describing the Token Manager
Figure 3.12: Flow chart describing the local QoSM
Chapter 4

Implementation

The global quality of service manager (GQoSM) was implemented on the master node of the system. The system consists of five nodes in total. These are the motion controller (MotC), Panel Controller (PanC) x-ray controller (XRC), external image viewer (EIV) and image reconstructor (ImRC) as described in section 2.1. The gantry controller (GC) is implemented as part of the MotC due to time restrictions in the thesis work.

4.1 OSE

Operating System Embedded (OSE) is an operating system created by ENEA. It is based on a message passing system rather than a shared memory architecture. This feature makes it ideal for implementing a middleware. This implementation uses OSE version 5.3.

4.2 SHAPE

The platform used for the implementation was originally created as part of the DySCAS[2] project. It consists of services that communicate through message passing. The architecture is depicted in figure 4.3.

4.3 Simulated Device

The simulated device is quite general and does not have any real functionality, although it does send some dummy data between nodes to test the underlying architecture. The dummy data sent is controlled by the QoSM designed in this thesis. The system acts as a representative of a typical device in the Med Tech environment. It consists of three ccerboard nodes running OSE and two linux nodes running on standard PCs. All nodes are connected to an Ethernet bus. The system is depicted in figure 4.1. Originally the system should have included a CAN bus as well but the development of the gateway connecting the CAN and Ethernet buses was not finished on time.
4.3.1 Gantry Controller

The GC was not implemented as a stand-alone application but rather its functionality is simulated by the MotC.

4.3.2 Motion Controller

The MotC runs on a standard PC with the operating system Linux. The MotC acts as a time keeper in the system and coordinates the interactions between all the application services connected to the system. It also synchronizes the startup in the system. The functionality of the MotC is graphically presented in figure 4.4.

4.3.3 Panel Controller

The PanC runs on a cerfboard 255[1] with the operating system OSE. When the PanC starts up it waits for the start command from the MotC. Upon receiving this it sends its bandwidth request to the LQoSM. After this it waits for a response from the LQoSM before proceeding. If the request was denied it sends a new request. When a request is accepted it proceeds to its functional part. It waits for the MotC to send the control signal which triggers the sending of data to the ImRC. Before sending data it waits for a token from the LQoSM. The functionality of the PanC is graphically presented in figure 4.5.

4.3.4 X-Ray Controller

The XRC runs on a cerfboard 255 with the operating system OSE. The XRC only waits for signals and receives them. It never sends anything on the network bus. The functionality of the XRC is graphically presented in figure 4.6.

4.3.5 Image Reconstructor

The ImRC runs on a cerfboard 255 with the operating system OSE. The ImRC starts by waiting for a start command from the MotC. When this is received it registers its transfers with the LQoSM and waits for a response. Just like the PanC it sends new registrations if the requests are denied. When the request has been accepted it proceeds to its functional part. This includes sending data to the EIV if there is data to send and a
token for sending. Otherwise it checks if there is data to receive from the PanC. If none of this is true it waits until some of it is. The ImRC runs as the bulk service of the system. The functionality of the ImRC is graphically presented in figure 4.7.

4.3.6 External Image Viewer

The EIV runs on a standard PC with the operating system Linux. The EIV receives data from the ImRC. When this is done it checks if it has received a displayable image. If it has it displays it to the user. Otherwise it just waits for more data from the ImRC. The functionality of the EIV is graphically presented in figure 4.8.

4.3.7 QoSM

The QoSM was implemented close to how it was designed in section 3.3. Application nodes implement the LQoSM that gathers local requests and sends them to the ReqM in the GQoSM. The ReqM forwards the requests to the scheduler which calculates the schedule and forwards it to the token manager. Requests are returned to the ReqM which checks if they have expired. The token manager does not send the tokens to the LQoSM but rather sends the heartbeat and the tokens itself when it is time for the next heartbeat. Figure 4.2 depicts the implemented QoSM.

Figure 4.2: Implemented QoSM.
Figure 4.3: Schematic figure of SHAPE
Figure 4.4: Motion controller flow chart.
Panel Controller

Figure 4.5: Panel controller flow chart.
X-Ray Controller

Figure 4.6: X-Ray controller flow chart.
Image Reconstructor

Start

Wait for start command

Register Transfers

Success?

No

Yes

Signaling waiting?

No

Yes

Token?

No

Yes

Is there data to send?

No

Yes

Send Data and decrease data count

Yes

Send Data and increase data count

No

Gantry Position?

Yes

Ignore Signal

No

Data?

Yes

No

Discard Signal

Figure 4.7: Image Reconstructor flow chart.
**External Image Viewer**

![Flow chart of External Image Viewer](image)

- **Start**
- **Wait for start command**
  - **Is there data to display?**
    - Yes: **Display Data**
    - No: **Receive Data**
      - **Did a transfer finish?**
        - No: **Wait for start command**
        - Yes: **Display Data**
  - No: **Receive Data**

- **Is all data received?**
  - Yes: **Display Data**
  - No: **Wait for start command**

Figure 4.8: External Image Viewer flow chart.
Chapter 5

Results

This master thesis set out to accomplish three distinct goals as described in section 1.1. Of these goals the main objective was to design a QoSM. This objective has been accomplished and described in section 3.3. The designed QoSM provides support for hard real-time requirements in the middleware. It performs admission control and scheduling of network traffic.

Another goal of this thesis was to identify the requirements on a middleware from a QoS perspective. These requirements are described in section 3.2 which provided the basis for the QoSM design. The resulting requirements concern specifications of resource usage, monitoring of resources and timing requirements in the system.

The last goal of this thesis was to examine different middleware for the Med Tech environment. This was done in section 2.8. Several middlewares with real-time support were found but none were specific for the Med Tech environment. The examined middlewares are HADES, ARMADA, TAO, Kokyu and DRC.

During the implementation phase of this thesis the proposed design of a QoSM was implemented. Although, this was not fully accomplished, the major functionalities of QoS i.e. admission control and scheduling have been implemented on the demonstration platform. The implemented QoSM is described in section 4.3.7.

The designed QoSM has some limitations. For instance it does not handle data dependency requirements between different requests. The design is also limited in the negotiation possibilities it offers applications. Negotiation can be done by sending requests for high bandwidth first and requesting lower bandwidth if the first request fails. It is however not possible to specify a range of interesting QoS levels and let the QoSM choose the level the system can provide.
CHAPTER 5. RESULTS

THIS PAGE INTENTIONALLY LEFT BLANK
Chapter 6

Conclusions

The QoS that was designed in this thesis is able to handle the application model it was designed for. However, the limited time and the lack of a complete implementation made this hard to test. The project should have been smaller and more time should have been spent on achieving measurable results. The specified objectives should have been more detailed and less abstract. A lot of interesting future projects are presented in chapter 7. It would be very interesting to see how the QoS performs in conjunction with the future work projects. The identification of requirements was a good way to structure the design of the QoS. This also presented a measurable result for one of the objectives of the thesis. The earlier implementation should have been examined more closely at an early stage so the current implementation phase could have been better planned.
Chapter 7

Future Work

During the thesis work a lot of interesting ideas have emerged.

7.0.8 Connection Management

It would be possible to develop a subsystem of the middleware that monitors the physical network topology of the system. By connecting this subsystem to the QoSM it would be possible to provide QoS to applications running on different network buses.

7.0.9 Scheduling Algorithms

The QoSM designed in this thesis implements a specific scheduling algorithm. It would be interesting to implement other algorithms and compare them to each other. Specifically from a utility based perspective. Also it would be possible to have the scheduler implement several algorithms at once and let the choice of algorithm be policy controlled. E.g. when the system has applications with real-time requirements running, the scheduler would use a scheduling algorithm that provides support for this. If no such applications are running the scheduler might even opt to not use any scheduling and instead implement a best-effort network.

7.0.10 Cost of Service

Cost of service was discussed in section 2.5.4. The designed QoSM does not implement any kind of cost of service. This is more interesting for applications with soft real-time requirements. Possibly the use of cost of service could be connected to the policy based algorithm selection discussed in section 7.0.9.

7.0.11 CPU Scheduling

The designed QoSM only performs network scheduling. It would be interesting to have it implement scheduling of other parts of the system as well. This could be implemented as part of the LQoSM. By connecting the scheduling time to the heartbeat period as discussed in 3.3.2 the QoSM would truly implement hard real-time guarantees on application level.
7.0.12 Preemptive Scheduling

The designed QoSM does not support preemptive scheduling. It would be very interesting to combine this with bandwidth guarantees for already scheduled network requests. Especially in combination with CPU scheduling this becomes a really complicated problem.

7.0.13 Run-time Prioritized Rescheduling

The designed QoSM does not support replacing lower priority requests that have been accepted with higher prioritized requests that arrive during run-time. This is a feature to be able to guarantee that an accepted request is fully accepted for its entire duration. It would however be interesting to analyze the implications on application level of allowing such scheduling replacements to occur.

7.0.14 QoS Levels

The designed QoSM requires that a specific request is explicitly defined. It would however be interesting to study the possibility of specifying a range of requests or possibly several different levels of them. This would provide a better foundation for negotiation of QoS levels in the system.
Bibliography


by: Andreas Ziethén
aziethen@gmail.com

Kungliga Tekniska Högskolan
Department of Mechatronics
ENEA
Appendix A

Acronyms

ACSA - Admission Control and Schedulability Analysis
API - Application Programmer's Interface
AQOSM - Application Quality of Service Manager
BPS - Bits per Second
CAN - Controller Area Network
CM - Connection Manager
CORBA - Common Object Request Broker Architecture
CSMA/BA - Carrier Sense Multiple Access with Bitwise Arbitration
CSMA/CD - Carrier Sense Multiple Access with Collision Detection
DA - DRC Resource Agent
DRC - Distributed Resource Controller
DM - DRC Service Manager
DySCAS - Dynamic Self Configuring Automotive Systems
ECU - Electrical Control Unit
EDF - Earliest Deadline First
EIV - External Image Viewer
GC - Gantry Controller
GQoS - Global Quality of Service Manager
HADES - Highly Available Distributed Embedded System
ImRC - Image Reconstructor
IPG - Inter-Packet Gap
LCM - Least Common Multiple
LLF - Least Laxity First

by: Andreas Ziethén
aziethen@gmail.com
Kungliga Tekniska Högskolan
Department of Mechatronics
ENEA
**APPENDIX A. ACRONYMS**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LQoSM</strong></td>
<td>Local Quality of Service Manager</td>
</tr>
<tr>
<td><strong>MotC</strong></td>
<td>Motion Controller</td>
</tr>
<tr>
<td><strong>MUF</strong></td>
<td>Maximum Urgency First</td>
</tr>
<tr>
<td><strong>MW</strong></td>
<td>Middleware</td>
</tr>
<tr>
<td><strong>OS</strong></td>
<td>Operating System</td>
</tr>
<tr>
<td><strong>OSE</strong></td>
<td>Operating System Embedded</td>
</tr>
<tr>
<td><strong>PanC</strong></td>
<td>Panel Controller</td>
</tr>
<tr>
<td><strong>QoS</strong></td>
<td>Quality of Service</td>
</tr>
<tr>
<td><strong>QoS-A</strong></td>
<td>Lancaster Quality of Service Architecture</td>
</tr>
<tr>
<td><strong>QoSM</strong></td>
<td>Quality of Service Manager</td>
</tr>
<tr>
<td><strong>ReqM</strong></td>
<td>Request Manager</td>
</tr>
<tr>
<td><strong>RM</strong></td>
<td>Rate Monotonic</td>
</tr>
<tr>
<td><strong>TAO</strong></td>
<td>The ACE Orb</td>
</tr>
<tr>
<td><strong>TDMA</strong></td>
<td>Time Division Multiple Access.</td>
</tr>
<tr>
<td><strong>XRC</strong></td>
<td>X-Ray Controller</td>
</tr>
</tbody>
</table>