Sharing Real-Time Objects

in Distributed Embedded systems

Master Thesis

Version history

Description	Version	Init	Revised	Approved	Date
Start of document	0.1.0	IHA	ERL		2014-01-29
Very early draft version sent to Brian Nielsen for comments	0.2.0	IHA	ERL		2014-09-15
Revised after comments, and design section added. Draft version sent to Brian Nielsen for comments	0.3.0	IHA,ERL			2014-12-04
Nearly final draft sent to Brian Nielsen for final comments	0.4.0	IHA,ERL			2015-03-02
Final Thesis – handed in version	1.0.0	IHA,ERL			2015-03-27

Students: Erland Larsen & Ib Havn
Supervisor: Brian Nielsen, AAU Centre of Embedded Software Systems, CISS

Abstract

With increasing intelligence in consumer products, distributed embedded systems have become more common. Sharing real time objects between nodes are one of the major challenges in cost effective systems as they often lack the resources needed for technologies like RMI and CORBA. This report describes a real industrial case of a resource constrained distributed embedded system, and suggests a middleware that could improve performance and maintainability of the product.

The contribution of this project is to implement an effective middleware and communication layer that will ease the development of distributed applications in small resource constrained embedded systems. CORBA Event Service and "The ACE ORB" (TAO) is studied, and optimisations are suggested to decrease complexity and code size. For communication services, the Local Interconnect Network (LIN) is studied as inspiration for implementation of a transparent communication layer using the standard UART embedded in most microcontrollers, and RS-485 hardware on the physical layer.

The designed and implemented middleware is divided into separate modules with very little coupling, this makes it possible to replace the communication service to match other systems communication technologies and bus structures. The modularity of the middleware makes it event possible to use the middleware on a standalone node without the communication service part.

An API is provided, and a simple example of a distributed application is programmed on four 8-bit Atmel microcontrollers connected by a RS-485 bus.

The demo system shows that it is relatively simple to implement a distributed embedded system with the provided API, and the latency of the communication service is predictable.

A low coupling to hardware makes it easy to port to another platform and/or another network technology.

Acknowledgements

Our master study has been made possible with support from our employer VIA University College and mainly the great support from Head of ICT Engineering Jens Cramer Alkjærsig. We would also like to thank our colleagues in Department of ICT Engineering for support and flexibility.

We are grateful for the very careful guidance we have received through the project from our supervisor Brian Nielsen (bnielsen@cs.aau.dk), AAU Centre of Embedded Software Systems, CISS. Brian has introduced us to the world of research methods and to the form in which a research work should be presented.

Stephan Erbs Korsholm has kindly helped us with input to the problem formulation.

Our dear colleague Mona Wendel Andersen has done a great job correcting our English language.

Not least, thanks to our families who have supported us in the process and have had to do without us both mentally and physically for long periods.

Contents

1 Introduction

Intelligent control systems in e.g. automotive, apparatus, modern house appliances, autonomic robots etc. encapsulate more and more complexity. At the same time, requirements for safety, availability, user friendliness, fast response, maintenance capability etc. are growing.

These control systems are often implemented using a number of resource constrained specialised Electronics Control Units (ECU) such as intelligent sensors, actuators, processing units taking care of the business logic and user interfaces (UI). These control units are typically built around resource constrained Micro Controller Units (MCU) together forming a distributed real-time embedded (DRE) system. The MCUs used in these control units vary in size and performance depending on the functionality they are responsible for.

Jurgen Hubbert, in charge of the Mercedes-Benz passenger car division, publicly stated in 2003:"*The industry is fighting to solve problems that are coming from electronics and companies that introduce new technologies face additional risks. We have experienced blackouts on our cockpit management and navigation command system and there have been problems with telephone connections and seat heating.*" Alberto Sangiovanni-Vincentelli concluded: *"I believe that this sorry state is the rule for the leading OEMs, it is not the exception in today's environment. The source of these problems is clearly the increased complexity but also the difficulty of the OEMs in managing the integration and maintenance process with subsystems that come from different suppliers who use different design methods, different software architecture, different hardware platforms, different (and often proprietary) Real-Time Operating Systems"* [1].

Resource constrained ECU are typical based on 8/16 bits MCUs, without memory managements units (MMU) and with very small memories like RAM < 16 kB and Flash < 128 kB. These MCUs vary in IO-capability depending on the job they are selected to perform [2]. MCUs used for automotive ECUs are typically equipped with hardware implementations of controller area network (CAN[1]), which is widely used as communication network between ECUs. Development tools for resource constrained MCUs are often limited compared to what is available in development tools used to desktop application.

A DRE system faces **real-time challenges** both in the single ECU, which must live up to real-time requirements for the functions it handles and also in the interaction with the other nodes in the system. In an automotive ECU controlling the engine there are a number of real-time requirements e.g. open the fuel valves in a specific time and period after a sensor has detected the position of pistons and values from lambda sensors (environment pollution control), ignite the spark plugs in a narrow time interval after the fuel is injected in the cylinder. These are typical **hard real-time requirements**. On system level there are real-time challenges involving the ECUs forming the DRE system. E.g. the ECUs that are responsible for the ABS brake system measure the rotation of the wheel, and this information is sent to the ECU controlling speed pilot, which figures out if it is necessary to adjust the speed by sending information to the engine ECU to change the power of the engine to keep the set speed. If the car is equipped with an automatic gearbox, then the engine ECU can send a request to the ECU controlling the gearbox to shift the gear up or down. In this very simplified example, there are a number of real-time requirements, which the involved ECUs must fulfil in cooperation. These **distributed real-time challenges** set focus on the real-time capability not only the ECU's but also the network connecting the ECUs.

The car industry is not only focusing on more and more powerful ECUs, but also increasingly in very resource constrained and cost effective systems, of which the Local Interconnect Network (LIN) bus [3] is a good example. A central door locking system is an example of a DRE

[1] CAN is an ISO standard (ISO 11898-1:2003).

that uses this bus. Each door has a MCU to control the locking mechanism and with the ability to communicate with other systems doors become more intelligent. E.g. to avoid burglars from opening the boot and steel passengers' bags when stopping for red light, the boot will lock automatically when the car starts driving, and remain locked until one of the front doors has been opened.

The communication system for such small systems must be efficient since no dedicated communication controller is present, so all link layer functions must be implemented in software.

The distributed nature of DRE systems gives a good modularisation of the application. The downside is that designing and implementing firmware for distributed systems are more complex and demanding than for centralised systems. To overcome the increased complexity, a number of middleware implementations are described in literature and used in industry. One of the difficulties found in implementation of DRE systems is the **distribution of the state of the system**.

Resource constrained DRE systems as described above, lead to specific **requirements for middleware (MW), and possibly operating systems (OS)** that can be used in these systems. These requirements are: Small program memory and RAM footprint, real-time capabilities, efficiency, flexibility on CPU architecture, network topology etc.

Middleware like CORBA, CORBA Event Service, Real-time CORBA, Remote Method Invocation (Java/RMI) and similar middleware in combination with Ethernet or similar are used in many distributed system implementations. These implementations are demanding many resources from the hardware platform they are running on and scaling them down to resource constrained hardware platforms are in many cases impossible.

The main aims for this thesis is to focus on very resource constrained hardware platforms based on 8-bit MCUs with limited resources of data memory (RAM $\leq$ 4 kb) and program memory (Flash $\leq$ 64 kb), at the same time the thesis will try to be independent of a particular communication topology. Thus; the focus on very resource constrained hardware platforms the middleware will be designed and implemented without the need for an operating system. With this focus a middleware will be designed, documented and a prototype will be implemented to test the capabilities of the developed middleware on a small distributed application.

Though the focus is on very resource constrained hardware platforms, it will be possible to upscale the middleware to run on more resource intensive platforms, and integrate it with small and cheap platforms where it is appropriate.

The thesis will model the distribution of real-time data objects and different underlying communication topologies, the models will be checked using UPPAAL[2].

This thesis is organised in sections where section 2.1 will describe the real industrial case that has founded the motivation for this. Section 2.3 will try to generalise the case.

The main part of the thesis describes the middleware development.

[2] UPPAAL is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata, extended with data types (bounded integers, arrays, etc.). http://www.uppaal.org/

2 Problem analysis

The problem analysis for this thesis is mainly case based, where the case is a specific existing product, which is implemented as a DRE system. The case based approach is chosen, because the purpose of the thesis is to design and implement a middleware that deals with many of the problems found in real life systems.

The particular system is selected thus we believe it is a good representation of many DRE systems based on resource constrained hardware platforms.

To give readers an insight into the structure and functionality of the product it is first described in general and later in more detail.

Next, a number of specific problems in the current implementation is found and described.

After this, the described case is generalised to cover a larger portion of DRE systems. The Problem description for this thesis is based on the generalised system.

2.1 Description of Case

The case is an existing DRE system with up to 10 nodes connected via an asynchronous serial half-duplex master based low bandwidth communication bus. Each node is a specialised ECU based on a resource constrained MCU. The application of the system is implemented as a distributed application deployed to different ECUs.

2.1.1 Hoist system

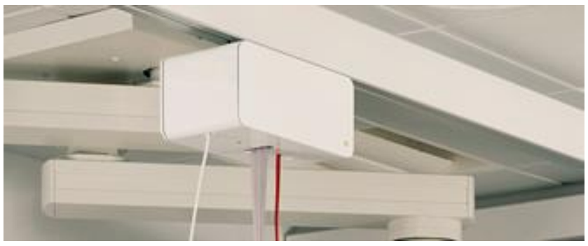


Figure 2-1 Hoist System

Figure 2-2 Hoist System in use at hospital

The motivation for the project is that the firmware in an existing battery operated hoist system has met the limit for adapting future functionalities, and a number of issues have been discovered, which cannot be solved with the current design and implementation.

The communication between the nodes in the system is mainly based on sporadic messages, which are results of events coming from the Remote Control or different status and alarm messages sent between the nodes.

Each of the ECUs/nodes in the hoist is responsible for handling different parts of the application. The hardware of the different node types is designed specially to take care of the nodes functions. Figure 2-3 shows a block-diagram of the hoist system.

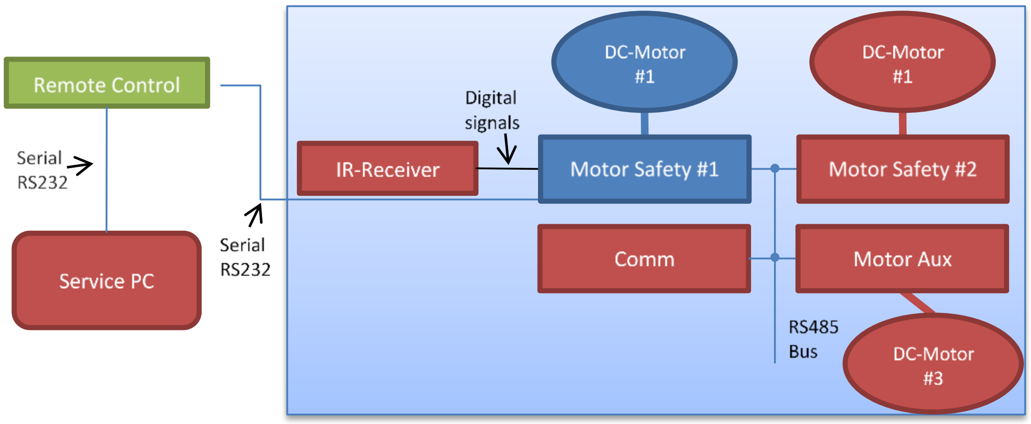


Figure 2-3 Overall Block diagram of Hoist system

The overall functionality of the different nodes in the hoist system are described below.

2.1.1.1 Motor Safety

The smallest configuration of a hoist system has a single DC-Motor (#1) controlled by a Motor Safety (#1) ECU. Hoist systems to be used for heavy persons are equipped with an extra DC-Motor (#2) controlled by its own Motor Safety (#2) ECU. Both Motor Safety ECUs are nodes in the system.

In a configuration with two DC-motors each are equipped with an electrically controlled disc brake, and as the motor, the corresponding ECU controls it. The two DC-motors are connected to the same physical shaft. Thus, the disc brakes and the DC-Motors must be controlled as synchronously as possible by the two ECUs controlling them.

2.1.1.2 Comm (Option)

A communication ECU can be included in the hoist system. This ECU is used to calculate and store a number of statistical runtime data together with a log of data for all hoists done with the system. It also interfaces two load cells used to measure the weight of the hoisted persons.

2.1.1.3 Motor Aux (Option)

This option is a simple version of a Motor Safety ECU that controls a smaller DC-Motor (#3) that can move the hoist system horizontally on a railing system mounted on the ceiling.

2.1.1.4 Remote Control

The normal cable based hand held remote control communicates with the hoist system via an asynchrony RS232 full duplex serial line, which is separate from the serial line used to the communication between the nodes.

2.1.1.5 IR-Receiver (Option)

This option can replace the basic functionality of the cable based remote control.

The IR remote control has no impact on our project and is not described further.

2.1.1.6 Service PC

A Service Engineer can connect a Service PC to the remote control (RS232). This gives the possibility to read out statistical data, logs etc. and to update the firmware in the different ECUs in the Hoist System.

2.1.1.7 Communication Network

The physical layer of the communication network is a half-duplex RS485 communication bus, with a single master (Motor Safety #1). The master sends out a token to the nodes one by one. The nodes can then reply with a single message/telegram, or a NAK character to tell it has nothing to send. The token is not distributed equally to the nodes; some of the nodes gets the token more often than others, e.g. Motor Safety #2 and Motor Aux gets it every second time, and the Comm-module gets it every fifth time. This is an attempt to ensure that the nodes that take part in the control loop (controlling the DC-Motors synchronically) can get faster access to the communication bus.

2.1.2 Standards

The firmware in the Hoist System must comply with *Medical device software - Software life-cycle processes (DS/EN 62304:2006)* [4]. The system is a Programmable Electrical Medical System (PEMS). To classify the system, risk analyses must be done for the software design, and it will inherit the highest risk of the subsystems.

The possible classifications are:

Class A: No injury or damage to health is possible

Class B: Non-serious injury is possible

Class C: Death or serious injury is possible

The classification has been done for the existing case system by the manufacturer, and has a Software Safety classification of Class B: *Non-serious injury is possible.* A redesign of the software will not change this as long as it complies with the existing requirements.

The development lifecycle is illustrated as a V-model in Figure 2-4 [4]. A development process fully in compliance with this model is out of scope for this thesis.

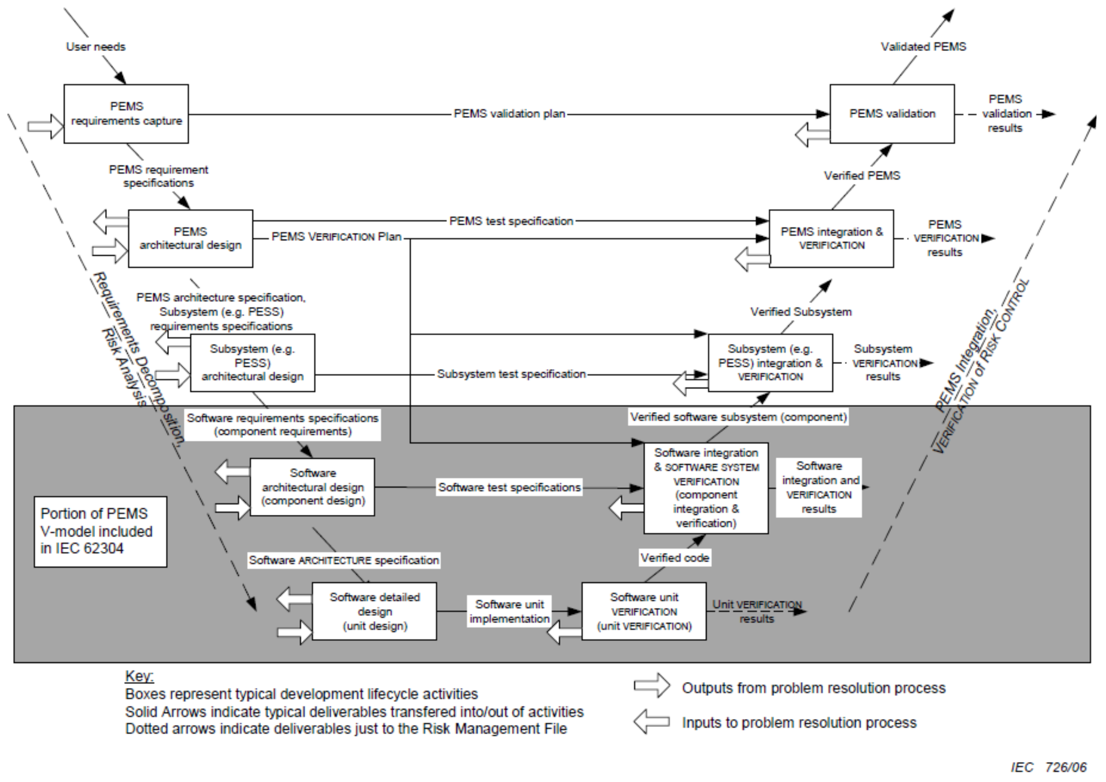

User needs
Validated PEMS
PEMS requirements capture
PEMS validation plan
PEMS validation
PEMS validation results
PEMS requirement specifications
Verified PEMS
PEMS architectural design
PEMS test specification
PEMS VERIFICATION Plan
PEMS integration & VERIFICATION
PEMS VERIFICATION results
PEMS architecture specification, Subsystem (e.g. PESS) requirements specifications
Verified Subsystem
Subsystem (e.g. PESS) architectural design
Subsystem test specification
Subsystem (e.g. PESS) integration & VERIFICATION
Subsystem VERIFICATION results
Requirements Decomposition, Risk Analysis
PEMS Integration, VERIFICATION of RISK CONTROL
Software requirements specifications (component requirements)
Verified software subsystem (component)
Software architectural design (component design)
Software test specifications
Software integration & SOFTWARE SYSTEM VERIFICATION (component integration & verification)
Software integration and VERIFICATION results
Portion of PEMS V-model included in IEC 62304
Software ARCHITECTURE specification
Verified code
Software detailed design (unit design)
Software unit implementation
Software unit VERIFICATION (unit VERIFICATION)
Unit VERIFICATION results
Key:
Boxes represent typical development lifecycle activities
Solid Arrows indicate typical deliverables transfered into/out of activities
Dotted arrows indicate deliverables just to the Risk Management File
Outputs from problem resolution process
Inputs to problem resolution process
IEC 726/06

Figure 2-4 Development lifecycle

2.2 Recognised Problems

Modular Design: An analysis of the firmware's source code has revealed that the firmware implementation lacks focus on layered and distributed design and without focus on adapting future functionality. An example is that the application layer of the firmware in the different nodes knows details about how information are being sent between nodes. The firmware has a very high coupling in the design and lacks design.

Time Synchronisation: At first, the control problem in the hoist system seems very simple. When an operator activates an up/down/left/right button on the Remote Control the motor(s) must start up with an acceleration ramp, and when the operator release the button the motor(s) must stop with a deceleration ramp. The real control problem is that in a hoist with two DC motors on the same shaft, and a drive motor to move the hoist horizontally, the movement of these motors has to be synchronised in time. Each of the motors is controlled from separate nodes/ECUs (Motor Safety #1, Motor Safety # 2, Motor Aux) on the communication bus. The disc brakes and the DC Motors must be controlled synchronously by the separate ECUs controlling them. One of the big issues with the current implementation is this time synchronisation.

The Master (Motor Safety #1) build up a kind of global state of the current operational state of the hoist, part of this state is then send to the different nodes when they need the information. An operational state can be that the DC Motors must be supplied with a specific Pulse Width Modulation (PWM) value and a specific direction or that the DC Motors have to be stopped, etc.

The problem with token passing priority is that it is the nodes that are prioritised and not the most important messages or data in the system.

Many of above-mentioned issues found in the Hoist System relates to the data distribution between the ECUs.

An example of this data distribution happens when a user presses the up key on the remote control to hoist a person; this will result in internal communication between the nodes in the Hoist system. This communication is described in Figure 2-5. The master/slave based communication is working with tokens that are send out from the master in a kind of prioritised round robin scheme. The token are sent more often to some of the nodes, than it is to others. This is done because some nodes implement parts of the application that has shorter deadlines than other parts thus, it is more important to have a fast communication channel to these.

See Figure 2-5. At (A) the Master Node adds an unimportant message to slave SN1 in its internal send queue. At (B) the user presses the up key on the remote control, the Master Node receives this information, and starts up the DC motor, connected to the node, with a speed of 500. At the same time it queues an important message to slave SN1 to do the same to the motor connected to that node. At this point the motor connected to the Master Node is started. At (C) the Master Node sends the token to the next slave in sequence (SN3), which has nothing to send back, and therefore replies with a NAK.

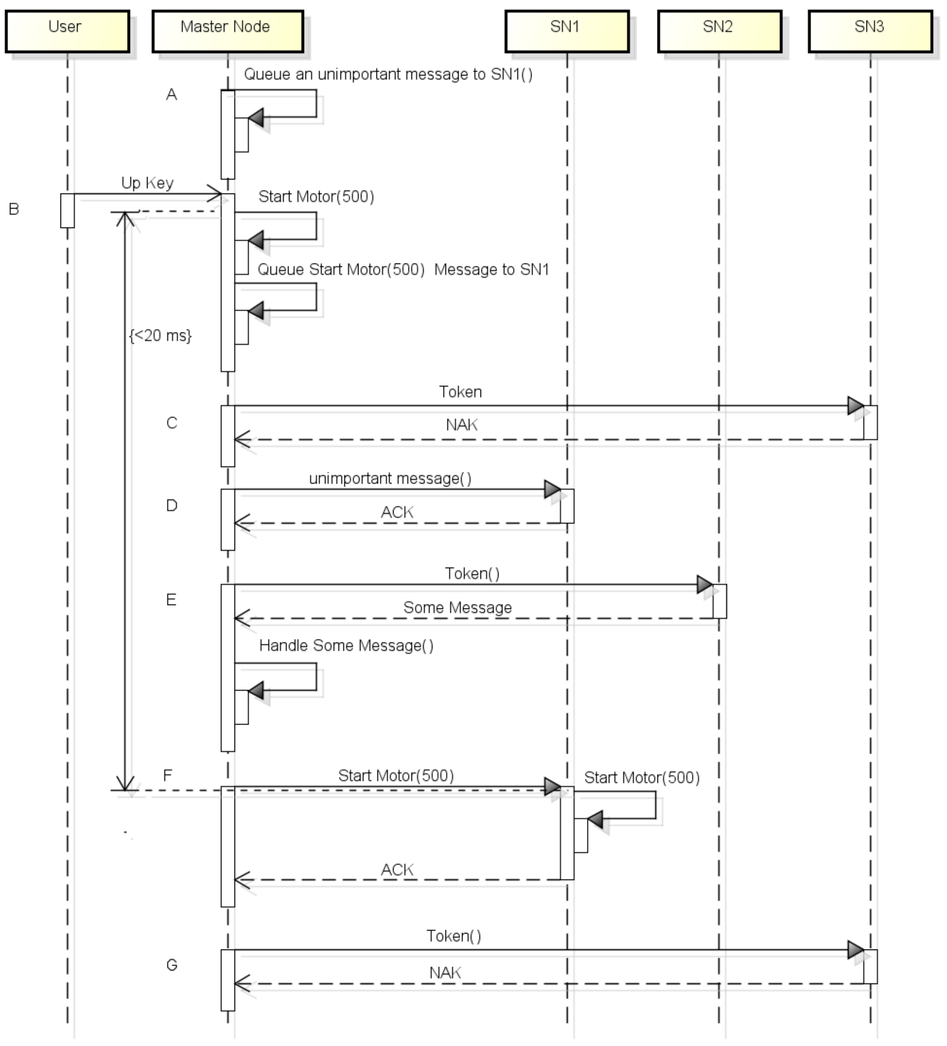


Figure 2-5 Example of some worst-case communication sequence

At (D) it is SN1's turn to get the token and because the Master Node has a message (the unimportant message queued at (A)) to SN1 it sends that as part of the token, SN1 has nothing to send back, so it just replies with an ACK to confirm that the message is received correct. At (E) the Master Node sends the token to the next slave in sequence (SN2), which has some message that it sends back to the Master Node who handles the message. At (F) it is finally time to send the important Start Motor (500) message to SN1. At this time the motor connected to slave SN1 is started. In the ideal world this should have been done at exactly the same time as the motor connected to the Master Node was started. As seen in the sequence diagram the delay between starting the two motors should be less than 20 ms. At (G) it starts all over again.

The sequence diagram shows only a small part of how the two DC motors are controlled. Actually they are started up and stopped using acceleration and deceleration ramps, where the Pulse Width Modulation (PWM) value are increased and decreased during start and stop. These PWM values are controlled of the Master Node and sent to the slave node (SN1) in the same way as described in Figure 2-5.

2.2.1.1 Conclusion on Recognised Problems

The node that wants to send data must know which nodes must receive it (high-coupling).

Tokens are sent in a kind of prioritised order from the master.

There is no prioritised send and receive queue in the nodes.

Each node implements its own Transport Layer as part of the application code.

The communication is Master/Slave based; thus it is very important that the Master is running (single-point-of-failure).

2.3 Generalisation

This section generalises the case described in section 2.1, and tries to find some points where a general middleware could have helped design, implementation and test of distributed real-time embedded systems.

In the present case, some general key problems can be identified:

- **High maintenance costs**. It is complex and time consuming to implement new features into the system as well as debugging. There is a high coupling between the different functional elements in the system. An example is that on application level, the nodes must know which other node can deliver a specific service (e.g. motor power for hoisting).
- **High Latency.** As described in section 2.2, a high-priority message can be delayed by low-priority in the outgoing queue. That makes it difficult to have real time knowledge of the global state without an unacceptable coarse time scale, which leads to that the performance of the system suffers. The main problem here is the round-robin priority of nodes rather than messages.
  Another latency related problem is the non-efficient communication bus and protocol. The RS-485 bus has no support for real time traffic, but is still used in many embedded systems because of low cost and relative high immunity to noise. A CAN bus would certainly have been better, but since this project is concerned about preserving the present hardware, focus will be on the token-based point-to-point protocol.
  Again, the high coupling is a problem. The applications must have knowledge about the underlying network to address the receiver of a message. If a message is to be sent to more than one receiver, it cannot deliver to all simultaneously. Multiple copies must be generated, queued and wait for the token round-robin.
- **Difficult certification process.** Lack of documentation and unstructured design makes it costly and time consuming to verify that the firmware is in compliance with relevant requirements.

Typically, projects start out with simple systems but as time goes, new features and patches make them complex. The hoist system is a typical example where the nodes have too tight coupling.

In the industry, there are many examples of systems like this. They could benefit from a light-weight middleware to take away the complexity of distributing objects. A middleware that has a small footprint, is easy to implement into a new system, decouples the business logic from the distributed hardware and software design, coded in C and, preferably, open source would probably have a good chance of being adapted into new designs.

A lot of the work that has been done in the past to describe such distributed real time systems design (e.g. Real Time CORBA), has the implementation and verification done on powerful off-the-shelf hardware since real industrial products have not been available for the project. An example is found in [5].

This project takes off from a current product but will redesign the software completely. The goal is to implement a demo version of the middleware that leaves enough resources to make the implementation of the business logic possible.

If it is successful in this scale, it would also be possible to implement it in the systems that use more powerful controllers but generally the focus here is on a class of systems that has the following in common:

- Short range distributed nodes, typically within the same box.
- Resource constrained nodes
- Certifiable systems
- Small number of nodes (<30)

Our vision is that this work could lead to an Open Source standard for how to implement efficient middleware for a DRE. This gives many benefits, like:

- Faster to implement new products or new features to existing ones.
- Less complex applications, which leads to fewer bugs, better stability and safer systems.
- Easier to test and debug.
- Faster response on the distribution of software objects, because the middleware is designed with small code footprint and efficiency as a design goal.
- Standardized middleware increases reuse and easier integration with OEM products.
- Easier certification when middleware is already known by certification authorities.
- Possibility for nodes to connect and disconnect dynamically.

3 Problem Formulation

Based on the analyses of the Hoist System and general experience with embedded solutions, it seems that many distributed embedded systems need to share the state of a number of fairly small Real-Time Objects (typically containing a few bytes, integers, words etc.) in an easy and maintainable way.

For computer systems with high CPU power, high bandwidth communication, plenty of memory etc. a number of high-level programming and middleware technologies exists for handling problems and challenges in distributed software systems (e.g. CORBA, RMI, WEB-services). The resource requirements of these technologies has prevented the use of the technology on resource constrained embedded environments. Since problems relating to distribution of services and data are equally relevant for resource constrained distributed embedded systems it is likely that ideas found in these high-level technologies and concepts can aid in the efficient development of resource constrained embedded distributed systems as well.

The thesis of this work is:

High-level object sharing concepts and middleware technologies can indeed be implemented on resource constrained embedded systems as well to achieve more maintainable systems and easier development.

A main challenge is to keep the implementation sufficiently lean in terms of memory and computational requirements while at the same time maintain a high degree of functionality.

To substantiate our thesis this work describes and designs a lean, sufficiently complete and efficient implementation for sharing real time objects on resource constraint distributed embedded systems. We apply the design and solution to a complex existing embedded system and arrive at a solution that solves many of the issues identified in our analysis of the software of the Hoist System. We demonstrate how the embedded implementation can run on an ATMEGA64 embedded 8 bit micro-controller with just 4 kB of RAM and 64 kB bytes of code memory.

We argue that our solution is sufficiently general to be applied to other embedded systems domains as well.

The problems we give special focus:

- How can a middleware for resource constrained distributed systems be designed and implemented.
- How can network communication become more efficient, using the resources of the existing communication hardware?
- What is the performance of the system?
- Can we avoid the use of a Real Time Operating System which is resource consuming?

The thesis will be answered by:

- Proposing a general method for sharing Real-Time Objects in resource constraint DRE systems.
- Describing a middleware to make the application programming independent of the inner workings of Real-Time Object sharing method.
- Measure the Code space and RAM usage for a given CPU architecture and compiler.
- Measure the latency and jitter when updating Real-Time Object in different situations.
- Model check (e.g. UPPAAL) of the middleware and communication protocol.

- Implementation of a C-library (API) for the middleware and a small demo application.
- Test and documentation of the C-library.
- Evaluate our implementation quantitatively and qualitatively.

3.1 Delimitation of Problem

The focus will not be on creating a middleware that can fulfil ISO 62304: Medical device software – Software life cycle processes, but the work will try not to conflict with this standard.

The design of communication stack will focus on standard UARTs since this component usually is present in small microcontrollers. This will ensure a solution that can run in very price sensitive systems. If the system is less dependent on cost, or need better performance, a fast Real Time bus could be attached later (e.g. CAN-bus). This would, of course demand changes to the link layer.

Model-checking with UPPALL will only be done if time allows it.

4 Theory - Related designs/architectures

In this section we have chosen to describe designs/architectures that are highly used in the industry today. The designs/architectures described here are selected from our research of related works as the best matching to our problem. The designs/architectures we are focusing on is CORBA, CORBA Event Service and the LIN protocol.

CORBA and CORBA Event Services are not in any aspect designed to fit into resource constrained distributed embedded systems, but they has some of the aspects that we would like to implement in our middleware.

4.1 CORBA

The principle of RPC is that an application in a client machine makes a procedure call, the procedure call together with its arguments are marshalled into a buffer that are sent across the network to a server machine where the actual procedure is implemented. The server un-marshals the buffer and executes the procedure and marshals the return value(s) into a buffer and via the network send it back to the client machine. The client machine un-marshals the buffer and send the return value back to the original calling application. Normally this is implemented as a synchronous procedure call, meaning that the calling client is blocked until it receives the return values from the server.

The main requirements for CORBA are to implement RPC with the following goals in mind:

- Object Orientation
- Location transparency
- Independence of programming languages
- Internet Inter-ORB Protocol

Object Orientation Remote procedures/operations are grouped into interfaces, like it is known from Java and C++ classes, an instance of such an interface is a CORBA object, which resides in a server. In CORBA each CORBA object has a unique id – the CORBA Object reference, which must be used every time the object is used.

Location transparency makes it easier to use CORBA objects, which will be used the same way independent of the location of the CORBA Object. The object can be located on the same machine as the calling client, or on another machine, seen from the calling client the syntax will be the same.

Independence of programming languages. CORBA is designed to work with multiple programming languages at the same time. Clients can be implemented using different programming languages as well as servers can be implemented with different programming languages. The independence of the programming languages are due to the CORBA *Interface Definition Language* (IDL), which is used to define the interface to a CORBA object independent of the programming language. The IDL is an independent language, that can be compiled into a specific programming language using a language specific IDL-compiler.

Internet Inter-ORB Protocol (IIOP) is a standard for the mediations of CORBA invocations over the TCP/IP transport layer. If an ORB implementation conforms to this standard, different vendors of ORB-implementations can be used together in a CORBA based distributed application.

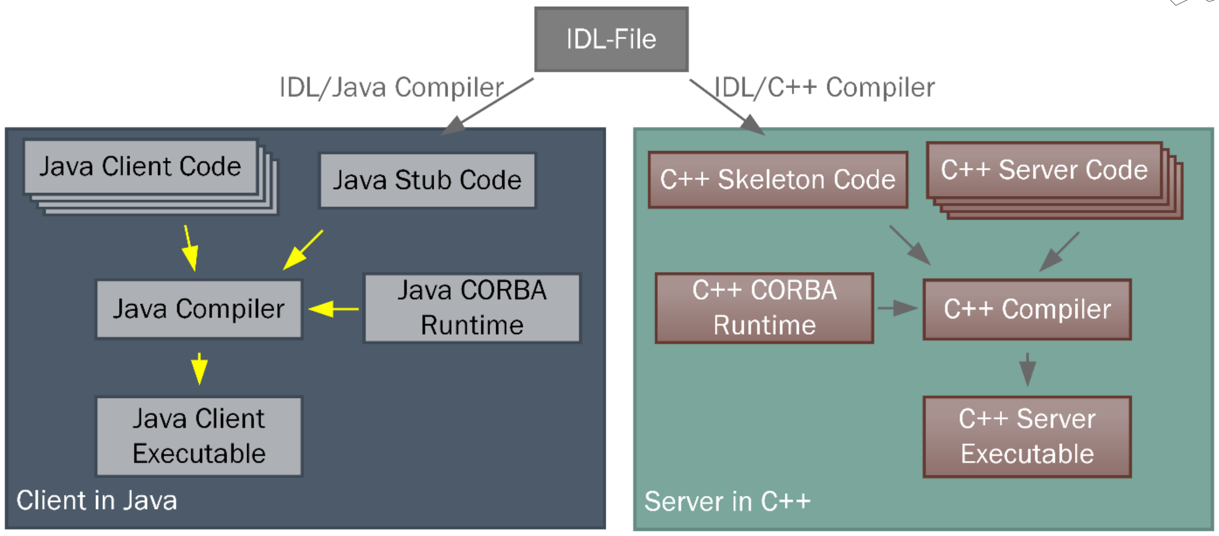


Figure 4-1 Creation flow of a CORBA Java Client and a C++ Host

Figure 4-1 shows how a common IDL file is compiled with language dependent IDL-compilers into code stubs for clients and code skeleton for servers matching the programming language the client and the host will be implemented in. These stubs are compiled together with the application code for the client and the host, and a language specific CORBA runtime code, into executables for client and server.

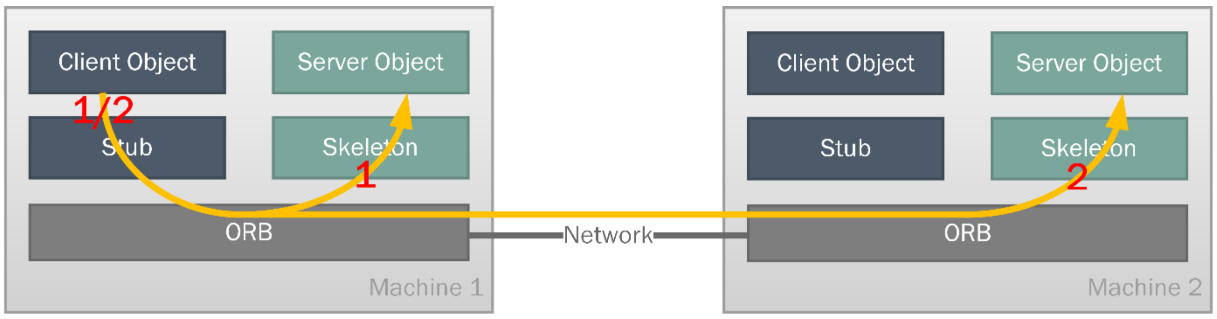


Figure 4-2 A client object calls a server object on same machine and on another machine

Figure 4-2 shows how a client object in one machine invokes an operation in a server object on the same machine (1), and on another machine (2). Seen from the client object it will not know if the server object is on the same machine or it is on another machine, it just calls the operation in the stub, which could be seen as a proxy for the object. The ORB knows where the server object is located, and if it is on the local machine (1), it passes the operation call to the local server object via the server skeleton. If the server object is located on another machine (2), the ORB marshals the operation call together with its arguments, and send it via the network to the ORB on the remote machine which un-marshals the operation call and invokes the server object via the server skeleton code. If the operation has return values, they will be sent back to the calling client object via the server skeleton, the ORB, the network, the ORB, the stub code and finally to the client object who called the operation. Invocations of operations are synchronous calls, so the client object is blocked until it receives the return values if any.

Each CORBA object has a unique system global reference, which are originated by the servers. These references are obtained in different ways. The most common ways are (a) that the server writes the references in a file that the clients have access to, (b) using CORBA's Interoperable Naming Service that maintain a central catalogue of object references and locations and finally (c) CORBA Trading Service who builds on the naming service, but is more flexible. The client object does not need to know where the server objects are located, but they need to know the server object's reference.

Standard CORBA is a strict client to server synchronous request invocation model, where clients send request to a remote object on a specific server/destination and the client blocks until it has received a response from the remote object, or an exception if the client doesn't respond.

4.2 CORBA Event Service

In a control system based on CORBA clients will typically be polling servers for different information, if the information change frequently, the polling frequency needs to be quite high. If more clients are interested in the same information, all the clients must separately poll the servers. This polling scheme can result in high utilisation of CPU and not least the network that connects clients and servers.

CORBA Event Service [6] is a service that instead of polling uses a call-back scheme, where clients are called every time the information, they are interested in, is changed. In this way CORBA Event Service allows applications to use a decoupled asynchronous communication model that decouples suppliers of information from consumers of the information. This call-back scheme can as well result in high CPU and network utilisation if the information is changing rapidly. To avoid this it is often possible to set up filters that determinate when the information is changed in a way that demands an Event to be send/a call-back to be called.

The basic model of CORBA Event Service is that *suppliers* produce events, and *consumers* receive them. Both the producers and the consumers are connected to an *Event Channel*, where the suppliers and consumer only know about the Event Channel, but do not know anything about each other. Each Event Channel handles only one specific event[3]. CORBA Event Service provides two models for Event delivery a *push model* and a *pull model*. In the push model the suppliers push events to the Event Channel, and the Event Channel pushes the events to the consumers (see Figure 4-3).

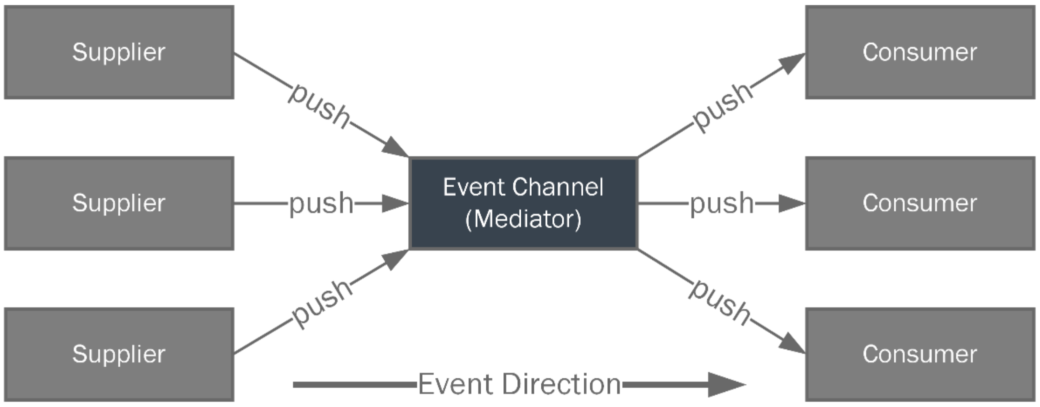


Figure 4-3 CORBA Event Service overview - Canonical Push Model

In the pull model the consumers pull the Event Service for events, and the Event Channel pulls the suppliers for events (see Figure 4-4).

[3] An example of an event can be the temperature in a room, another event could be the humidity in the room. These two types of events will be connected to two separate Event Channels.

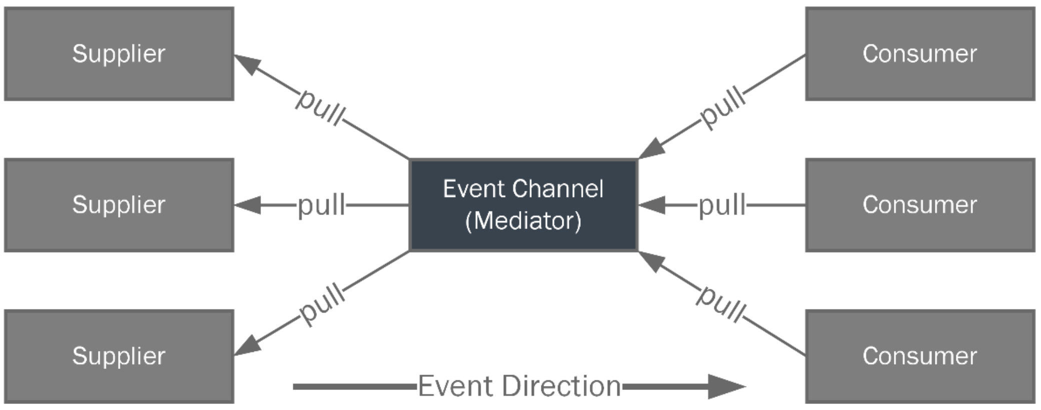


Figure 4-4 CORBA Event Service overview - Canonical Pull Model

Event Channels allow multiple consumers and multiple suppliers to connect to them. An Event Channel allows both push and pull suppliers and consumers to be connected at the same time. This leads to five different models:

1. *Canonical push model* where **all** suppliers and consumers of events use the push model, Figure 4-3.
2. *Canonical pull model* where **all** suppliers and consumers of events use the pull model, Figure 4-4.
3. *Hybrid push/pull model* where suppliers push events to the Event Channel, but consumers pull events from the Event Channel.
4. *Hybrid pull/push model* where the Event Channel pulls events from the suppliers and pushes events to the consumers.
5. *Mixed push/pull model* is a mixture of 3 and 4, where some of the suppliers will use the push model, others will use the pull model – the same is the case for the consumers.

A CORBA Lightweight Event Service [6] only implements the Canonical push model, and the rest of this description will focus on that.

When a supplier generates events of interest for the system, it will via the Event Service obtain an *Event Channel*, and will obtain a Proxy Push Consumer object. This proxy consumer is the only “consumer” that the supplier is aware of. When a consumer is interested in some events, it will via the Event Service obtain a *Proxy Push Supplier* object, this object is the only “supplier” that the consumer is aware of. The *Event Channel* works as a mediator between the proxy suppliers and consumers. The Event Channel is responsible for handling the connection between consumers and suppliers of the given events.

Seen from the consumers’ “end” of the Event Channel it acts as proxy supplier, and seen from the suppliers’ “end” it acts as a proxy consumer (see Figure 4-5). The consumers and suppliers interact only with these proxies that provide the illusion that they are interacting with the real suppliers and consumers. The Event Channel is “invisible” to consumers and suppliers, and works as a mediator between the proxies.

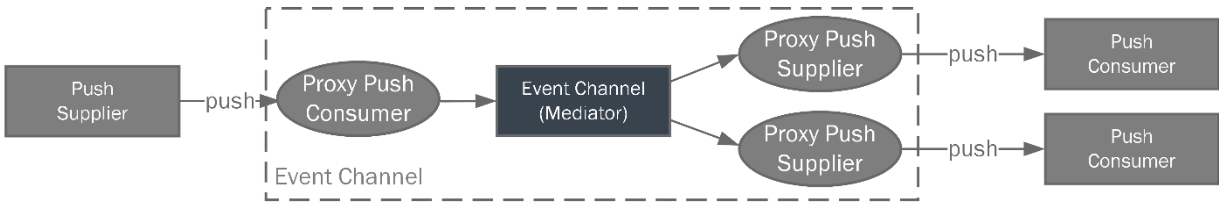


Figure 4-5 Event Service with visible Proxy Consumers and Suppliers

Each Event Channel has a unique object reference, these references are typically obtained from CORBA Naming Service.

A simplified sequence diagram showing how an Event Channel is setup between a Push Supplier and a Push Consumer can be seen in Figure 4-6. In the example the proxy supplier and consumer have already retrieved the Event Channels reference (*id* in the diagram) from the Naming Service.

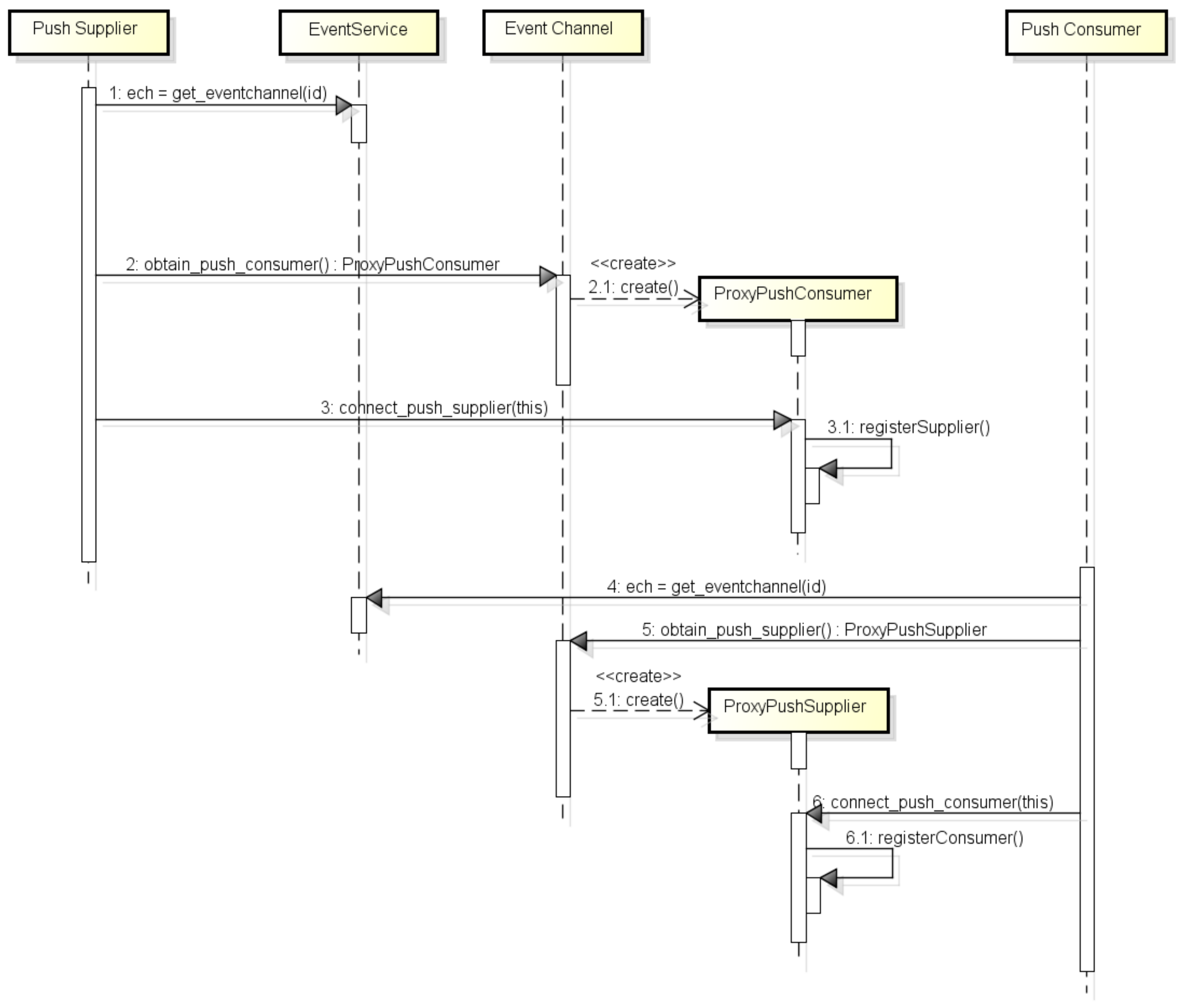

Push Supplier
EventService
Event Channel
Push Consumer
1: ech = get_eventchannel(id)
2: obtain_push_consumer() : ProxyPushConsumer
<<create>>
2.1: create()
ProxyPushConsumer
3: connect_push_supplier(this)
3.1: registerSupplier()
4: ech = get_eventchannel(id)
5: obtain_push_supplier() : ProxyPushSupplier
<<create>>
5.1: create()
ProxyPushSupplier
6: connect_push_consumer(this)
6.1: registerConsumer()

Figure 4-6 Simpliefied example showning how a push supplier and a push consumer set up an Event Channel in CORBA Event Service

After the sequence shown in Figure 4-6 is finished, the *Push Supplier* can call *push(data)* on the *ProxyPushConsumer* object, and the *EventChannel* will push this event on to the *ProxyPushSupplier*, that will push it to the *Push Consumer* object. The *EventChannel* decouples the *Push Supplier* completely from the *Push Consumer*.

Events in CORBA Event Service are data, with the CORBA type *Any*, meaning that it can contain all kinds of information, they are typically implemented as a *struct*[4] construction.

The Event Channel uses the ORB to mediate the events between the proxy consumers and proxy suppliers.

In distributed systems the Event Service can be implemented as a federated Event Channel. An example of this can be seen in the TAO real-time (RT) Events Service [7] [8] where a local Event Service is found in each Host (see Figure 4-7). The distributed Event Channels is then

[4] As known from C and C++.

connected by Event Channel gateways. The Event Channel gateways are implemented using an Observer pattern, and will only be observers on a nodes local Event Service, if there are consumers for the specific events on some other nodes in the system. Using federated Event Channels lowers the ORB and network utilisation when suppliers and consumers are located on different hosts.

The TAO real-time Events service adds filtering and priority based dispatching of events to the standard CORBA event service. Filtering makes it possible for consumers to set up different filters on events which allows the consumers to receive only the events they are interested in. This filtering can even be set up across events. An event filter could be that a consumer is only interested in an event, if other events also have occurred before the event of interest. This kind of filtering can of course be done in the consumer application, but by implementing it into the event service, it lowers the utilisation of the whole system. In TAO events can be given a priority, so that the most important events are handled first by the Event Channel and the Event Channel gateway.

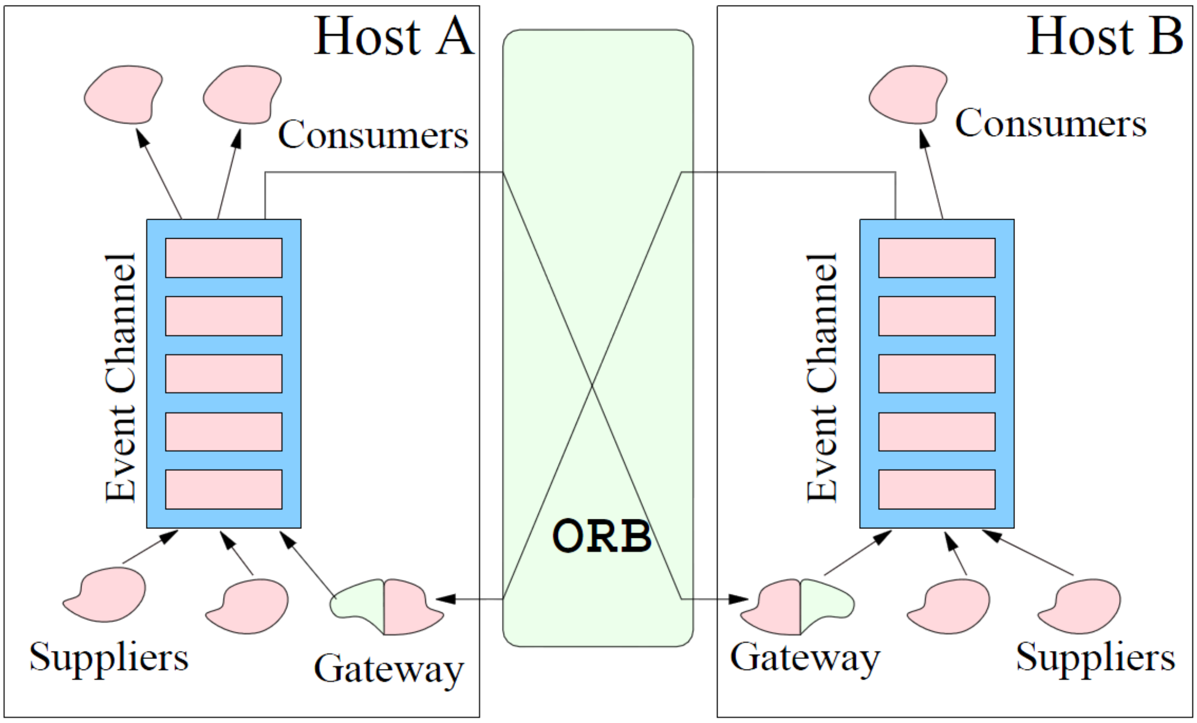


Figure 4-7 A Federated Event Channel Configuration as shown in [7].

The Event Channel Gateway on a host will only connect to remote gateways if there is at least one local consumer interested in the Event Channel. Moreover, if multiple local consumers are interested in the same event only one message has to be sent between the nodes, and the local Event Channel will do the local distribution to consumers.

4.3 Local Interconnect Network (LIN)

A LIN cluster consists of a single Master unit and several Slaves. In a hierarchical network, e.g. an automobile, the master acts as a gateway between the CAN and LIN bus, thus enabling a diagnostic tester to communicate with the LIN nodes. The specification describes a Transport Layer API for that purpose.

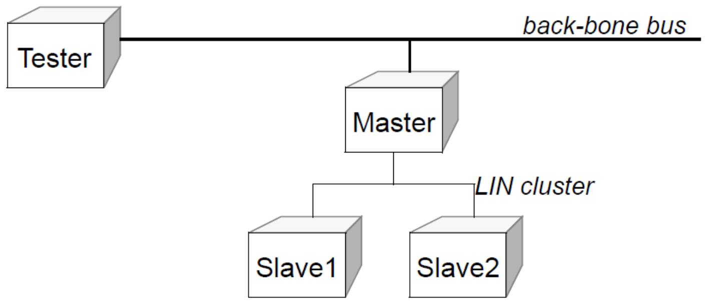


Figure 4-8 Typical setup for a LIN cluster using the Transport Layer API

The asynchronous communication protocol on the LIN bus makes it possible to use a low cost UART and a simple hardware interface. The transmission speed is up to 20 kbit/s, and the delays are deterministic.

Applications in the Master and Slave nodes communicate using a Signal based interaction API. All nodes in the cluster have a slave task, but the Master node also has a Master task. The Master task holds a schedule of when and which signal frames shall be transferred on the bus. Different signal frames have different signal IDs, called PID (Protected Identified).

The Master sends out a header identifying the signal to be transferred, and the node producing that signal responds with a response frame.

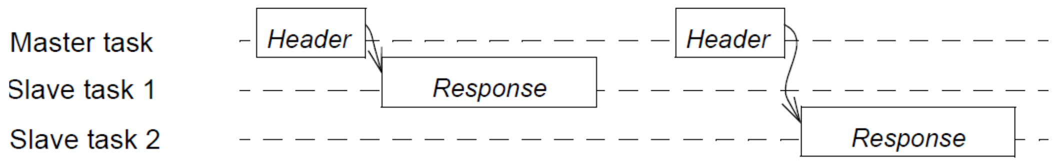


Figure 4-9 Header and Response frames

4.3.1 Messages

Two types of messages exists; signal and diagnostic messages.

Signals are the data, either scalar values between 1 and 16 bit, or byte arrays between 1 and 8 bytes. Each signal has only one publisher but more nodes may subscribe to it. More data fields from the same node can be packed into one Signal.

A frame is a combination of the Header (transferred by the Master task) and the Response (transferred by the publishing slave task).

The Header has 3 fields; Break, Sync and the PID (Protected Identifier).

The Break field is 13 nominal bit time of dominant value. It is used for signaling the start of a new frame. It is the only field that does not begin with a start bit and end with a stop bit surrounding a data byte.

A Sync field of value 0x55 follows the Break field. This can be used by slave nodes to synchronize the RX/TX clock.

The PID field specifies the Frame Type. 6 bit are available for different identifiers and two bit are used for parity control of the Frame Type. Values from 0 to 59 are used of specifying signals. Values from 60 to 63 are used for other purposes (diagnostic and configuration etc.).

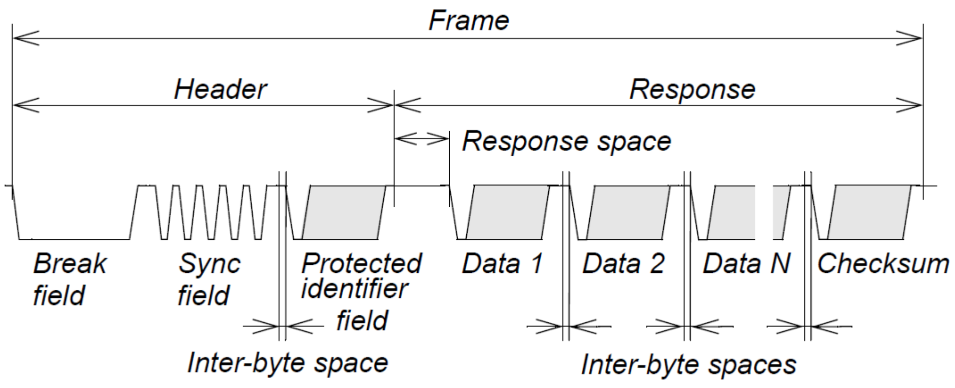


Figure 4-10 Frame structure

When a Slave task, receiving the header, recognizes a PID specifying a signal that it supplies, it must send a response on the bus. The Response Frame is max. 8 Bytes of data followed by a one byte checksum.

4.3.2 Checksum

The checksum is a 1-complement sum. For systems compatible with LIN rev. 1.x, the sum is calculated over all data bytes. In LIN rev. 2.x, the PID is also included in the checksum. It is then called an Enhanced Checksum.

It is obtained by integer addition of all the bytes in the frame. When an 8-bit checksum is calculated, all words added must also be 8-bit. Carry out of MSB is wrapped around and added to the sum. Finally when the sum is calculated the 1-complement value of it is appended the frame as the checksum.

Verification is then simple. All bytes in the received frame is added in the same way, including the attached checksum. If the result gives anything different from 0xFF an error has been detected (the sum of a value and its complement gives all 1-bits).

An example of an enhanced checksum calculation of a 4 Byte message is shown below.

Sending host:

Frame			
Field	Value (hex)	Value (Bin.)	
PID	0x80	10000000	
Type	0xC1	11000001	
		101000001	PID + Type (carry out of MSB)
Data1	0x12	00010010	
		01010100	PID + Type + carry + Data1
Data2	0x34	00110100	
		10001000	PID + Type + carry + Data1 + Data2
Data3	0x56	01010110	
		11011110	PID + Type + carry + Data1 + Data2 + Data3
Data4	0x78	01111000	
		101010110	PID + Type + carry + Data1 + Data2 + Data3 + Data4 (carry out of MSB)
Checksum	0x00	00000000	
		01010111	PID + Type + carry + Data1 + Data2 + Data3 + Data4 + carry + Checksum
	0xA8	10101000	1-complement

Frame transmitted:	
PID	0x80
Type	0xC1
Data1	0x12
Data2	0x34
Data3	0x56
Data4	0x78
Checksum	0xA8

Verification in receiving host:

Sum of PID + … + Data4		01010111	(as above)
Checksum in frame	0xA8	10101000	
Sum		11111111	No error!

Table 1 Calculation and verification of checksum.

4.3.3 Frame transfer

The normal way of transferring data is in unconditional frames. The Master can request a signal from a Slave or it can send a signal produced by the Master itself to one or more slaves. Figure 4-11 also shows an example of the Master requesting a signal be put on the bus produced by Slave 1 and destined for Slave 2. It all depends on the context of the PID, which has been decided in advance by the designers of the cluster. The schedule, controlled by the Master, decides the transmission of the specified signals.

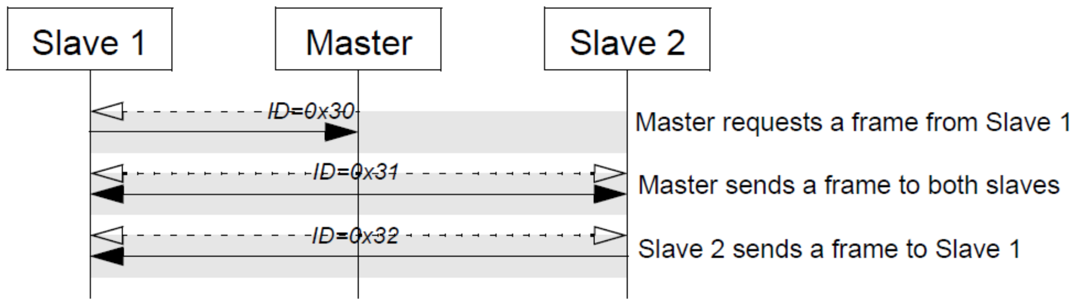

Slave 1
Master
Slave 2
ID=0x30
Master requests a frame from Slave 1
ID=0x31
Master sends a frame to both slaves
ID=0x32
Slave 2 sends a frame to Slave 1

Figure 4-11 Unconditional frame transfer

4.3.4 Event-triggered frames

To increase the responsiveness for seldom occurring events, the Master can also switch to a schedule for event-triggered frames. All Slaves subscribing to the event-triggered frame must respond if they have an updated signal for that frame, otherwise remain silent. This could lead to more than one node putting a response on the bus at the same time, so collisions can occur.

The Master node is responsible for resolving the collision by switching to a collision resolving scheduling table. Figure 4-12 shows an example of a collision of event-triggered frames.

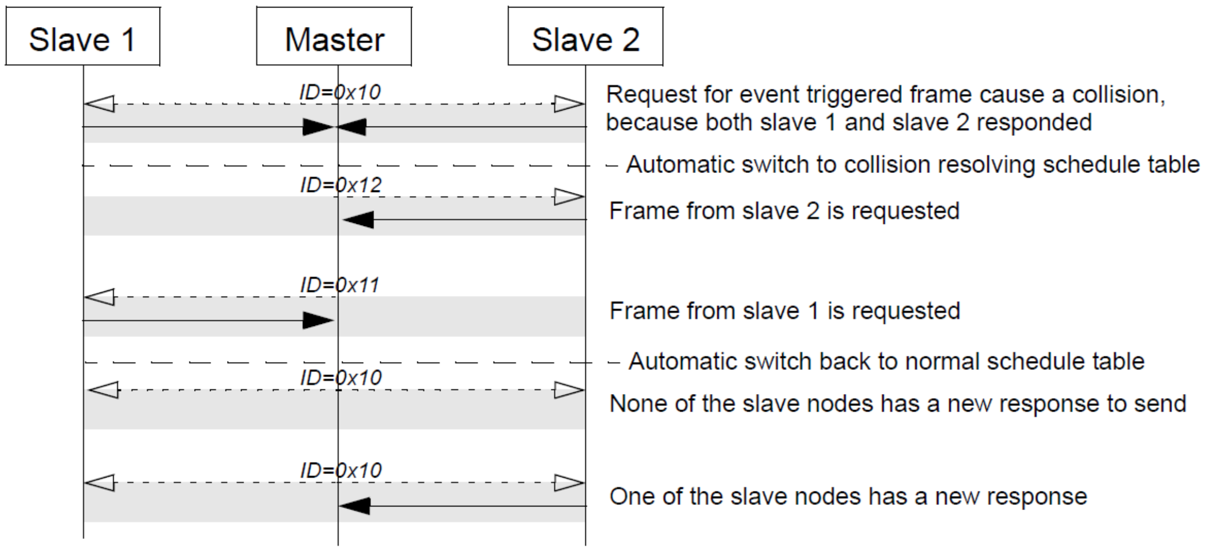

Slave 1
Master
Slave 2
ID=0x10
Request for event triggered frame cause a collision, because both slave 1 and slave 2 responded
Automatic switch to collision resolving schedule table
ID=0x12
Frame from slave 2 is requested
ID=0x11
Frame from slave 1 is requested
Automatic switch back to normal schedule table
ID=0x10
None of the slave nodes has a new response to send
ID=0x10
One of the slave nodes has a new response

Figure 4-12 Event-triggered frames

4.3.5 Sporadic frames

Some dynamic behaviour can also be achieved with Sporadic Frames. The Master is the only publisher of sporadic frames. If some event updates a signal on the Master that is associated with a sporadic frame, the Master can queue it according to priority and transmit it in the next frame slot reserved for sporadic frames.

All sporadic frames are associated with the same frame slot, and are transmitted according to priority.

5 Methods

Computer science is mainly characterised as an empirical discipline as it was already stated in 1975 in an article titled “Computer Science as Empirical Inquiry: Symbols and Search” by Allen Newell and Herbert A. Simon [9]. One citation from this article match perfectly the method we are using in this project:

“We build computers and programs for many reasons. We build them to serve society and as tools for carrying out the economic tasks of society. But as basic scientists we build machines and programs as a way of discovering new phenomena and analyzing phenomena we already know about.”

Our project is based on the case we have described earlier and our professional experience with embedded systems, both stand-alone and distributed. We have identified some problems that are common to distributed embedded systems running on small resource constrained hardware platforms. We will mainly use a case based **Experimental Research Method** to analyse and design and implement a middleware that can handle the main part of the identified problems. From experiments with the new middleware implementation we will obtain results that can lead to both a quantitative and a qualitative evaluation and we expect to demonstrate that the design and implementation really possess the proposed advantages.

We will carefully study related work and already existing designs/implementations to learn what already has been done to solve the identified problems.

5.1 Alternative Methods

We have taken other methods into consideration. In this section we will briefly describe these, and argue why we have chosen the Experimental and empirical approach.

A **Literature Study** [10] to analyse a number of already published sources that focuses on the identified problems we have described earlier. This method will give an insight in and a classification of what has been published in the area of our problem like what has worked, and what has not worked - but it will not lead to a design and an implementation we can evaluate on basis of the problems found in our case study.

The **Simulation Method** will typically be used where it is not possible to do experiments in the real problem domain, or where the systems in focus are too complex to make it feasible to set them up for real experiments. Our problem domain is not very complicated or difficult to do real experiments on.

Formal Modeling could be used to design a set of models of the problems found, and then it would be possible to test and verify different approaches to solve the problems found. This method could be a very good supplement to the experimental approach. Before we did the delimitations to our problem formulation it was planned to setup a number of formal models and do validation and verification test with UPPAAL.

6 Design

This section forms the main part of the thesis.

6.1 Overall Design Goals

Our design goals come from the case study and from *ISO/IEC 9126 Software engineering — Product quality* standard [11]. This standard categorises software quality into six characteristics (functionality, reliability, usability, efficiency, maintainability and portability). In our design process we have tried to have these six characteristics as focus point. Below, the means used to address the software quality characteristics is described.

Decoupling in *space*, *time* and *synchronization* of publishers and subscribers of RTOs. Space decoupling means that publishers and subscribers do not need to know each other, but only what RTOs they can produce or are interested in using. Time decoupling means that publishers and subscribers do not need to participate in the interaction of RTOs at the same time. Synchronisation decoupling means that the publisher is not blocked when updating a RTO [12]. Decoupling increases the scalability of the application by removing dependencies between publishers and subscribers. These decoupling's address the *maintainability* and *usability* software quality characteristics. The decoupling makes it easier to implement and maintain applications that shares data (RTOs) and events on these data.

Quality of Service (QoS) like handling of time-outs, node breakdown etc. must be part of the middleware. Handling of time-outs when RTOs are not updated in due time. If a node breaks down, then all the RTOs published by this node is set to an invalid status and all its subscribers are notified. These build in QoS address the *reliability*, *usability* and *functionality* software quality characteristics. Especially fault tolerance and recoverability is made easy for the application programmer to handle; thus the status of RTOs is automatically kept up-to-date by the middleware.

Focus will be on DRE Systems using Event based Execution without any need for a real-time operating system. The middleware will be designed in well-defined layers with clear interfaces between the layers – addressing the *testability* (part of *maintainability*) software quality characteristics. The middleware will be implemented in standard C. The middleware can be divided into two main parts: the Event Service and the Communication Service (COM-Service). The Event Service part will be designed independently of the hosting platform and the communication technology used. A simple interface between the Event Service and the Communication service will be defined. All these means address the *portability* software quality characteristics.

A number of optimisations will be done in the implementation of the Event Service, compared to standard CORBA Event Service. These optimisations can be done because of the target resource constrained platforms where not all functionality of CORBA Event Service is needed. These optimisations addres the *efficiency* software quality characteristics.

6.1.1 What to achieve

The overall purpose of the design can be explained from the model described in [13]. Figure 6-1 shows the model.

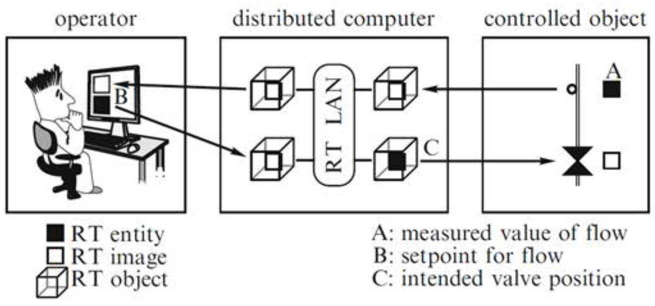


Figure 6-1 Real-Time entities (*RT-Entity*), Real -Time images (*RT-Image*), Real-Time objects (*RT-Object*)

RT-Entity is the origin of real-time state variable. Eg. *A* in the figure illustrates the actual flow in a pipe. *B* illustrates a set point for a valve on a user interface (UI). *C* is the current calculated *soll-wert* for the valve. Besides the dynamic value, the *RT entity* also has a number of static or semi static values like name, value domain, type, update period etc. In our design we call these data for meta-data.

RT-Image is the picture of an *RT Entry* seen in the software. Eg. the *RT image* coming from the flow-sensor and found in the *Distributed computer* and at the *UI* is an instant picture of the flow in the tube. The *RT image* in the valve is an instant picture of the *RT-entity* with the calculated *soll-wert* for the valve.

RT-Object (RTO) is just a container in the system that holds an *RT image* or an *RT entity*.

By looking at this model the design has to give answers to how can *RT entities* and *RT images* be established and how can they be distributed efficiently between the nodes/processes in very resource limited DRE systems.

6.1.2 Example scenarios

The middleware (MW) must provide services to the application for establishing and sharing *RT Objects* across different nodes in the system. It is based on a Producer (P)/Consumer (C) pattern, where processes that produce *RT-Objects* of interest for other processes are producers, and processes that want to use these *RT-Objects* are consumers. One design goal is that producers do not know which consumers (if any) are using the data, and the consumers do not know which producers have generated the data (*space* decoupling).

A consumer will only ask the MW for information and updates for *RT-Objects* of interest for the process. The consumer process must be notified by the MW when one of the *RT-Objects* of interest is being changed. Meaning that when an *RT-Object* is produced/updated by a producer all consumers of the *RT-Object* must be notified by an asynchronies event, like it is found in a traditional Observer pattern [14].

The MW must be designed as a distributed service that will run part of the software on each node in the system.

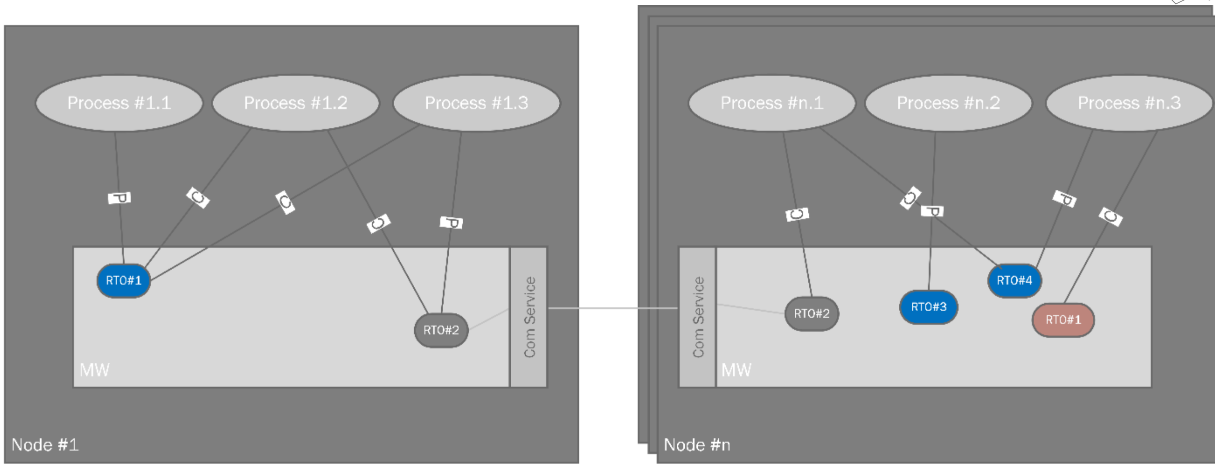


Figure 6-2 Snapshot at time *t*.

Figure 6-2 shows a scenario at time *t* with a number of nodes that all have the MW installed. In the scenario *Node #1* has three processes where *Process #1.1* produces (*P*) *RT-Object RTO#1*, *Process #1.3* produces *RT-Object RTO#2* and consumes (C) *RTO#1*, *Process #1.2* consumes both *RT-Object RTO#1* and *RTO#2*. *Node #n* has three processes where *Process #n.1* consumes *RT-Object RTO#2* and *RTO#4*, *Process #n.2* produces *RTO#3*, *Process #n.3* produces *RTO#4*, and has just told the MW that it would like to consume *RT-Object RTO#1*. *RT-Objects RTO#1* is in this moment only available in the MW in *Node #1*, the MW in *Node #n* must then establish a connection to *RTO#1* in the MW in *Node #1*, so that the new situation at time *t*+1 will be like in Figure 6-3.

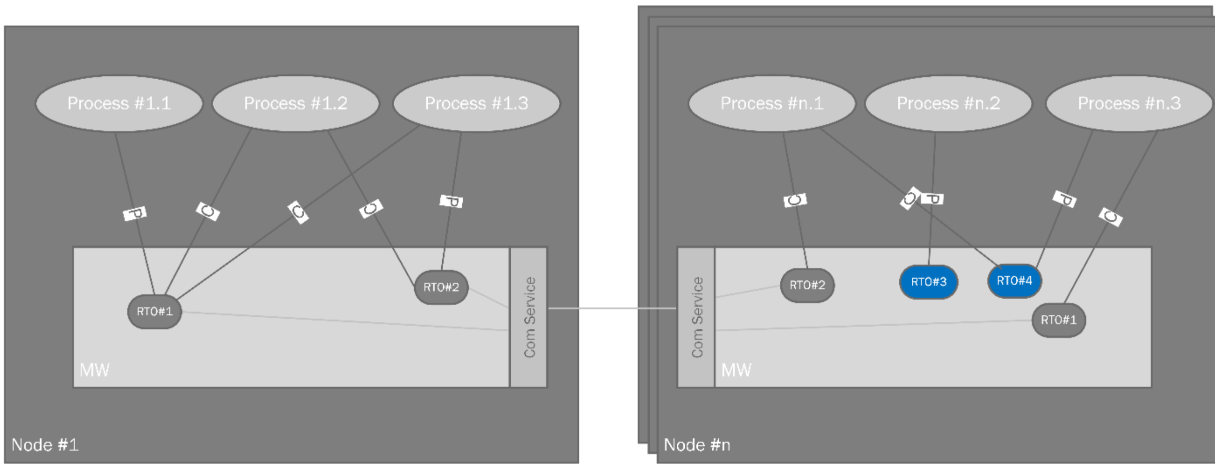


Figure 6-3 Snapshot at time t+1 of DDL

The processes that produce RTOs should not be aware of who is consuming them and on which nodes the consumers are running. And the same counts for consumers, who should not know who is producing RTOs and where they are located.

6.2 Design inspiration

The overall design is highly inspired by CORBA Event Service [6] (see 4.2). The goal is to implement parts of CORBA Event Service and parts of the Ace ORB (TAO) [7] [15] project additions to the Event Service, but always have in mind that this thesis is about implementing a middleware on very resource constrained platforms.

6.3 Overall Block Diagram

To give a better understanding of the following sections an overall block diagram can be seen in Figure 6-4.

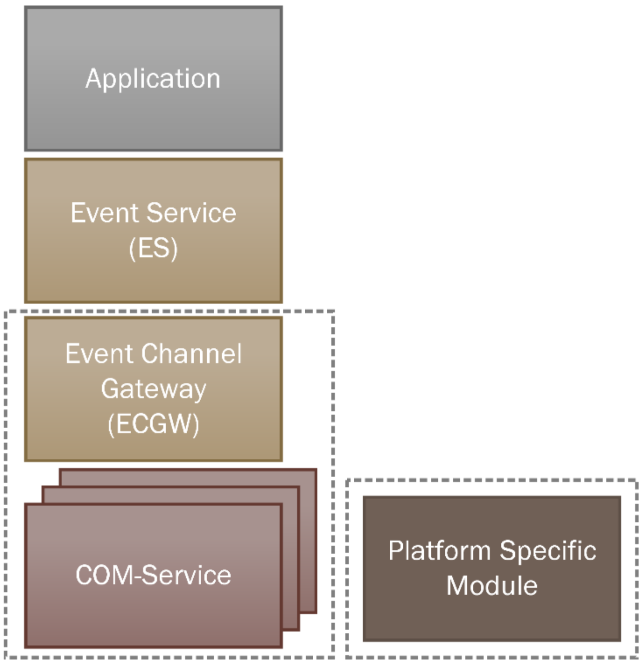


Figure 6-4 Overall Block Diagram

In a simple single node system with no need for real-time handling of RTOs, only the Event Service is necessary. If the real-time handling of RTOs is needed, then a very small Platform Specific module is needed.

If the Event Service should be used in a distributed system (multiple nodes), the Event Service Gateway and the COM-Service will be needed together with a very small Platform Specific module.

The COM-Service is shown as different modules, but only one module will be used in a system. The COM-Service module will be a specific implementation depending on the communication bus-topology used on the platform.

In all configurations the application code only needs to know about the Event Service.

6.4 Event Service Design

The example scenario (see 6.1.2) can more or less be mapped directly to the standard CORBA Event Service, but some modifications will be necessary to implement it efficiently on resource constrained platforms.

The way CORBA Event Service set up Event Channels, and make it possible to attach Suppliers and Consumers to the Event Channel, has inspired our design. Our design will be different in many aspects due to the focus on efficiency and the fact that not all the functionality found in CORBA Event Services are needed in small DRE-systems. Some of the functionality will not be implemented, and some will be changed to better target resource constrained hardware platforms and a few extra functions will be added.

CORBA Event Service use an Object Request Broker (ORB) to communicate events in the system. On the resource constrained platforms we are targeting it would be impossible to implement an ORB. Instead of the ORB a simple Communication Service (see 6.6) will be implemented.

A federated Event Service will be found on each node in the DRE-system. Figure 6-5 shows the federated Event Service on three different nodes. Node 1 has a Push Supplier, Node 2 has a Push Consumer, and Node 3 has two Push Consumers of the same object.

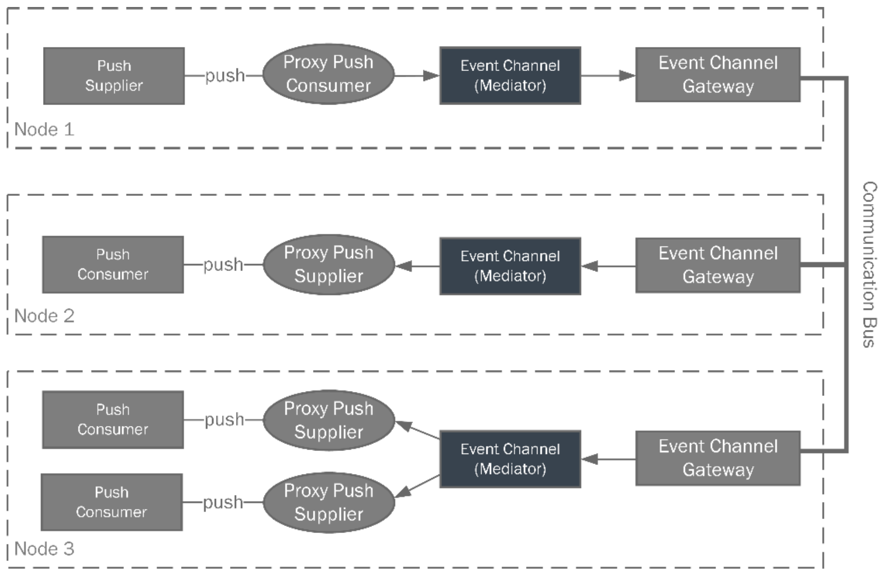


Figure 6-5 Distributed Event Service

Each node in the DRE will contain a copy of the basic Event Service together with an Event Channel Gateway that will connect the federated Event Services together, see Figure 6-6. The Event Channel Gateway will via the Communication Service be adapted to the specific communication bus technology found in the DRE-System to be used. The interface that the Event Channel Gateway must implement toward the Event Service will be the same independent of the communication bus technology, this interface will be described in 6.6.

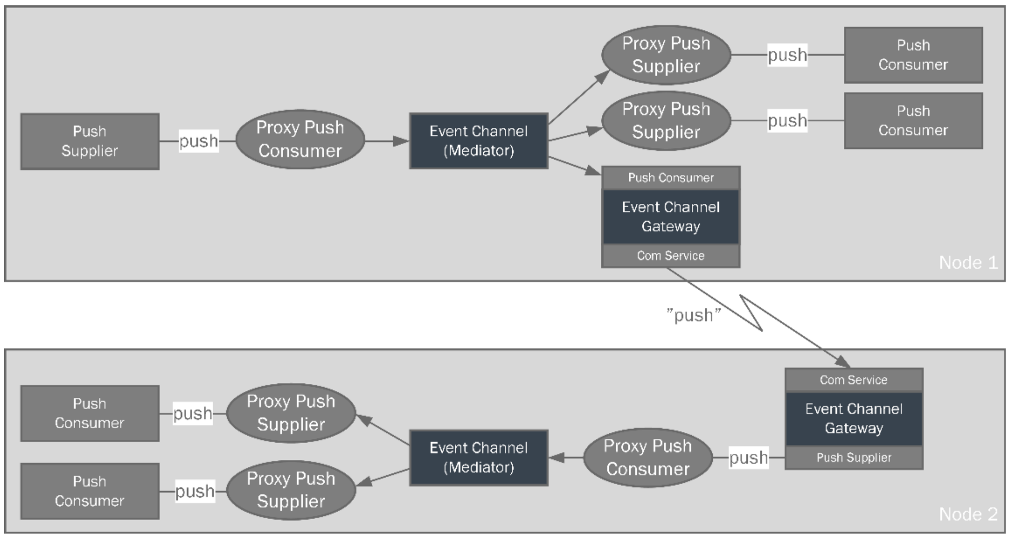


Figure 6-6 Event Channel Gateway added to Distributed Event Service

One way to add the Event Channel Gateways without changing the Event Channel radically, could be to combine the Event Channel Gateway with a Proxy Push Supplier or Consumer. By doing this the Event Channels will just see the gateway as a normal consumer or supplier – this is very similar to what is found in the Ace ORB (TAO) projects Real-time Event Service [8].

Optimisation Step 1: As Pull Consumers will not be part of our Event Service, Proxy Consumers will not need to know if there has occurred an event that the Consumer has not yet pulled, or that a Consumer has already pulled. This means that we do not need a Proxy Supplier for each Consumer, but all Consumers on a node can share a single Proxy Supplier. See Figure 6-7.

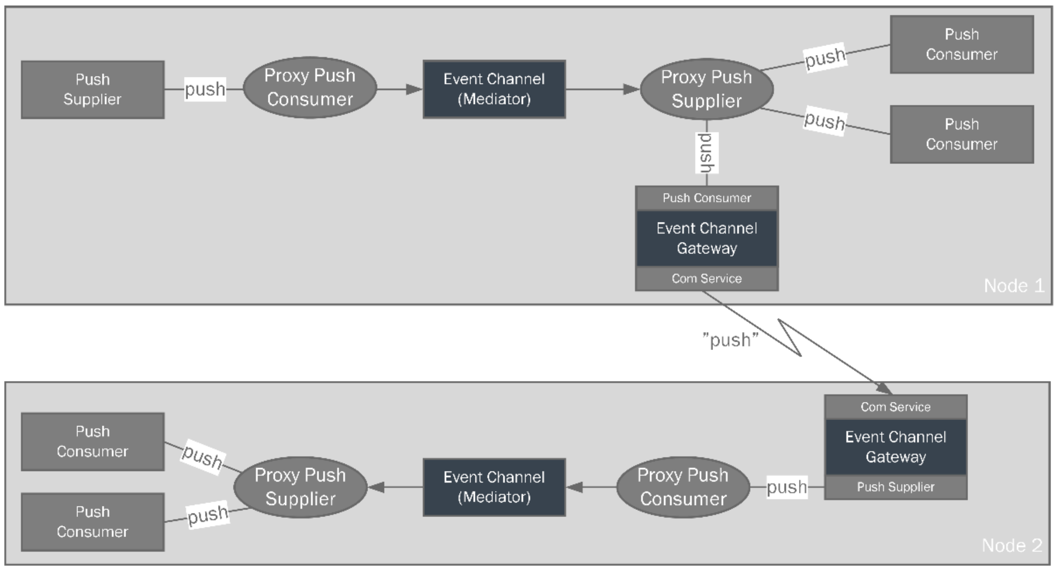


Figure 6-7 Optimized Event Service

The sequence diagram in Figure 6-8 shows in a simplified version how a supplier on one node (node1) and a consumer on another node (node2) establish and attach to an Event Channel for a given event (id). The numbers in parentheses refer to the sequence numbering in the diagram. It can be seen how the Push Supplier get an Event Channel from the Event Service (1), and obtain the Proxy Push Consumer (2) it will connect to (3). When the Push Supplier connects to the Proxy Push Consumer, the supplier will be registered on the Event Channel (3.1) and the Event Service will register that the event (id) is now supplied (3.1.1). The Event Service will via the local Gateway announce that the event (id) is now supplied (3.1.1, 3.1.1.1). The Gateways on the other nodes in the system, here node 2, will ask the local Event Service if the event (id) is consumed of some consumers at the node (3.1.1.1.1.1). In the example no consumers are yet interested in the event (id).

Later the Push Consumer on node 2 wants to connect to the event channel for the event (id). The Push Consumer gets the Event Channel from the Event Service (4), and obtains the Proxy Push Supplier (5) from the Event Channel, if the Proxy Push Supplier doesn’t exist on the node the Event Channel will create it (5.1). The Push Consumer connects to the Proxy Push Supplier as a consumer (6). The Event Channel will register the Push Consumer as consumer on the Event Channel (6.1), and tell the local Event Service that the event (id) is now consumed on the node (6.2). The Event Service announces via the local Gateway that the event (id) is now consumed (6.2.1, 6.2.1.1). The Gateways on the other nodes, here node 1, will ask the node Event Service if the event (id) is supplied by the node (6.2.1.1.1). In the example node 1 has supplied it earlier, the Gateway will now obtain a Proxy Push Supplier for the event (6.2.1.1.2), if the Proxy Push Supplier doesn’t exist, the Event Channel will create it (6.2.1.1.2.1), the Gateway connect as consumer to the Proxy Push Supplier (6.2.1.1.3), this will be registered on the Event Channel (6.2.1.1.3.1), and the Event Service will register that the event (id) is now consumed. The Gateway will announce that the event (id) is supplied by node 1.

The Gateways on the other nodes in the system, here node 2, will ask the local Event Service if the event (id) is consumed of some consumers at the node (6.2.1.1.4.1). The Gateway will obtain the Event Channel (6.2.1.1.4.2), and obtain a Proxy Push Consumer from the Event Service (6.2.1.1.4.3). The Event Channel will create it (6.2.1.1.4.3.1) and the Gateway will connect to it as a supplier (6.2.1.1.4.4). The Gateway will also register that node1 supplies the event (id) (6.2.1.1.4.5).

The distributed Event Channel is finally established.

The reason for sending the supplied messages two times (3.1.1.1.1 and 6.2.1.1.4) is that by doing this it is possible to make the distributed Event Services more stateless, it must not keep information about who can supply and who is consuming events, before they have an actual interest in knowing it.

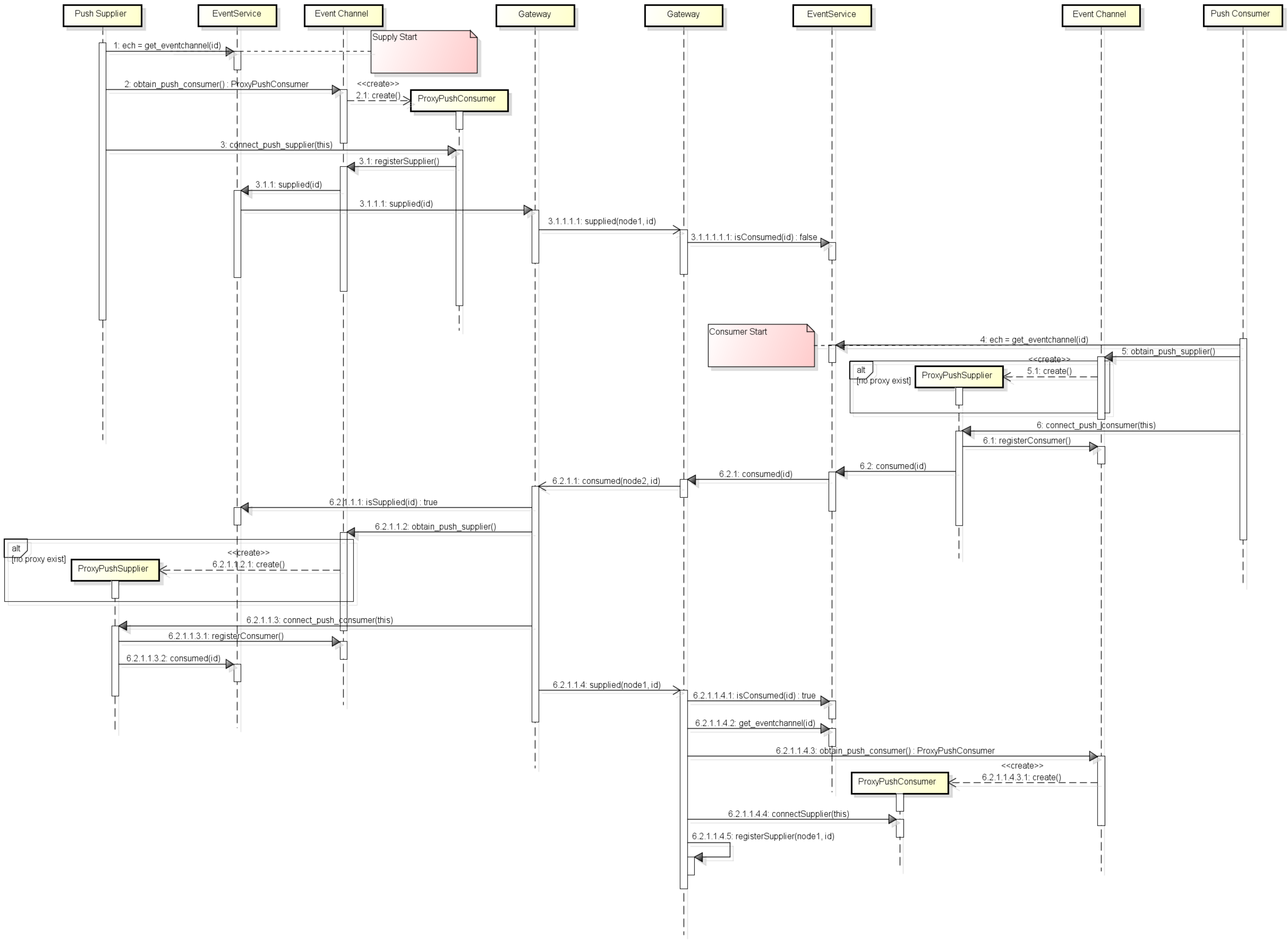

Push Supplier
EventService
Event Channel
Gateway
Gateway
EventService
Event Channel
Push Consumer
Supply Start
1: ech = get_eventchannel(id)
2: obtain_push_consumer() : ProxyPushConsumer
<<create>>
2.1: create()
ProxyPushConsumer
3: connect_push_supplier(this)
3.1: registerSupplier()
3.1.1: supplied(id)
3.1.1.1: supplied(id)
3.1.1.1.1: supplied(node1, id)
3.1.1.1.1.1: isConsumed(id) : false
Consumer Start
4: ech = get_eventchannel(id)
5: obtain_push_supplier()
<<create>>
alt
[no proxy exist]
ProxyPushSupplier
5.1: create()
6: connect_push_consumer(this)
6.1: registerConsumer()
6.2: consumed(id)
6.2.1: consumed(id)
6.2.1.1: consumed(node2, id)
6.2.1.1.1: isSupplied(id) : true
6.2.1.1.2: obtain_push_supplier()
alt
[no proxy exist]
<<create>>
6.2.1.1.2.1: create()
ProxyPushSupplier
6.2.1.1.3: connect_push_consumer(this)
6.2.1.1.3.1: registerConsumer()
6.2.1.1.3.2: consumed(id)
6.2.1.1.4: supplied(node1, id)
6.2.1.1.4.1: isConsumed(id) : true
6.2.1.1.4.2: get_eventchannel(id)
6.2.1.1.4.3: obtain_push_consumer() : ProxyPushConsumer
<<create>>
6.2.1.1.4.3.1: create()
ProxyPushConsumer
6.2.1.1.4.4: connectSupplier(this)
6.2.1.1.4.5: registerSupplier(node1, id)

Figure 6-8 Example of how the Event Service can setup a distributed Event Channel. The supplier first attach an event to the Event Channel and later the consumer attach to the same Event Channel.

How a push of an event from a local Push Supplier, in a scenario with a Push Consumer on a remote node, will be executed in the distributed Event can be seen in Figure 6-9.The Push Supplier pushes an event together with some event data to the Proxy Push Consumer (1). The Proxy Push Consumer notifies the Event Channel (1.1), that has registered the Proxy Push Suppliers

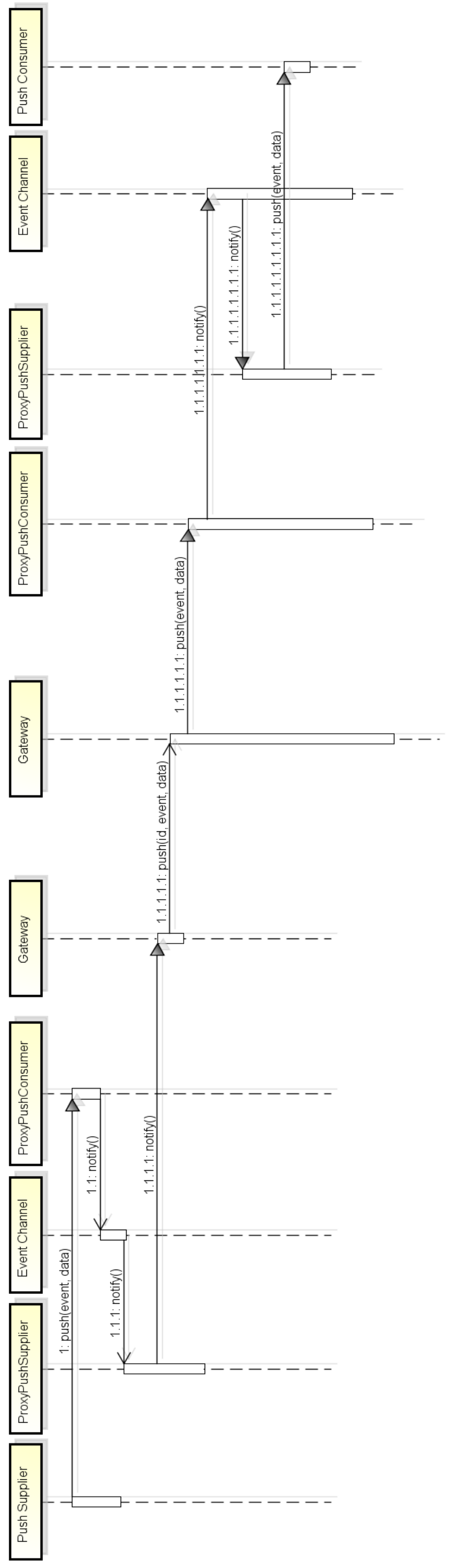


Figure 6-9 Push scenario before optimization

of this Event, it will notify the Proxy Push Supplies one by one, in this example only one (1.1.1). The Proxy Push Supplier in the example is the one that has the Gateway as consumer, the gateway is notified (1.1.1.1) and announces to the other nodes that the event has occurred (1.1.1.1.1).

The gateways on the other nodes will push it on to the Proxy Push Consumers, if any, that are connected to the specific event (1.1.1.1.1.1), that will notify the Event Channel about the event (1.1.1.1.1.1.1). The Event Channel notifies then the registered Proxy Push Consumers one by one, here only one (1.1.1.1.1.1.1.1), who again pushes the event on to the Push Consumer (1.1.1.1.1.1.1.1.1).

Optimisation Step 2: As it can be seen in Figure 6-9 there are quite many notifications inside each node to handle a push. After the first optimisation where the number of Proxy Suppliers and Consumers for each Event Channel was reduced to one in each node, it will be possible to combine the Proxy Supplier and Proxy Consumer with the Event Channel to a single Proxy for each event type/RTO. Figure 6-10 shows the result.

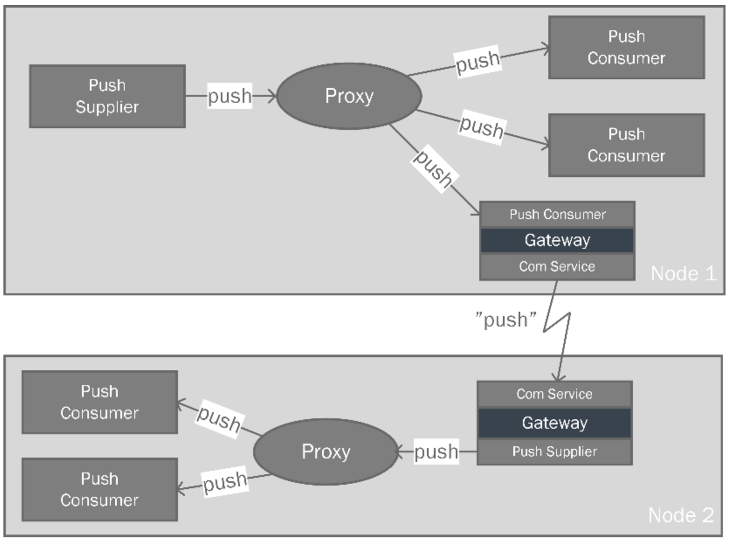


Figure 6-10 Proxy Supplier and Proxy Consumer combined with the Event Channel

After the last optimisation the sequence diagram for establishing a supplier on one node and a consumer on another node can be seen in Figure 6-11.

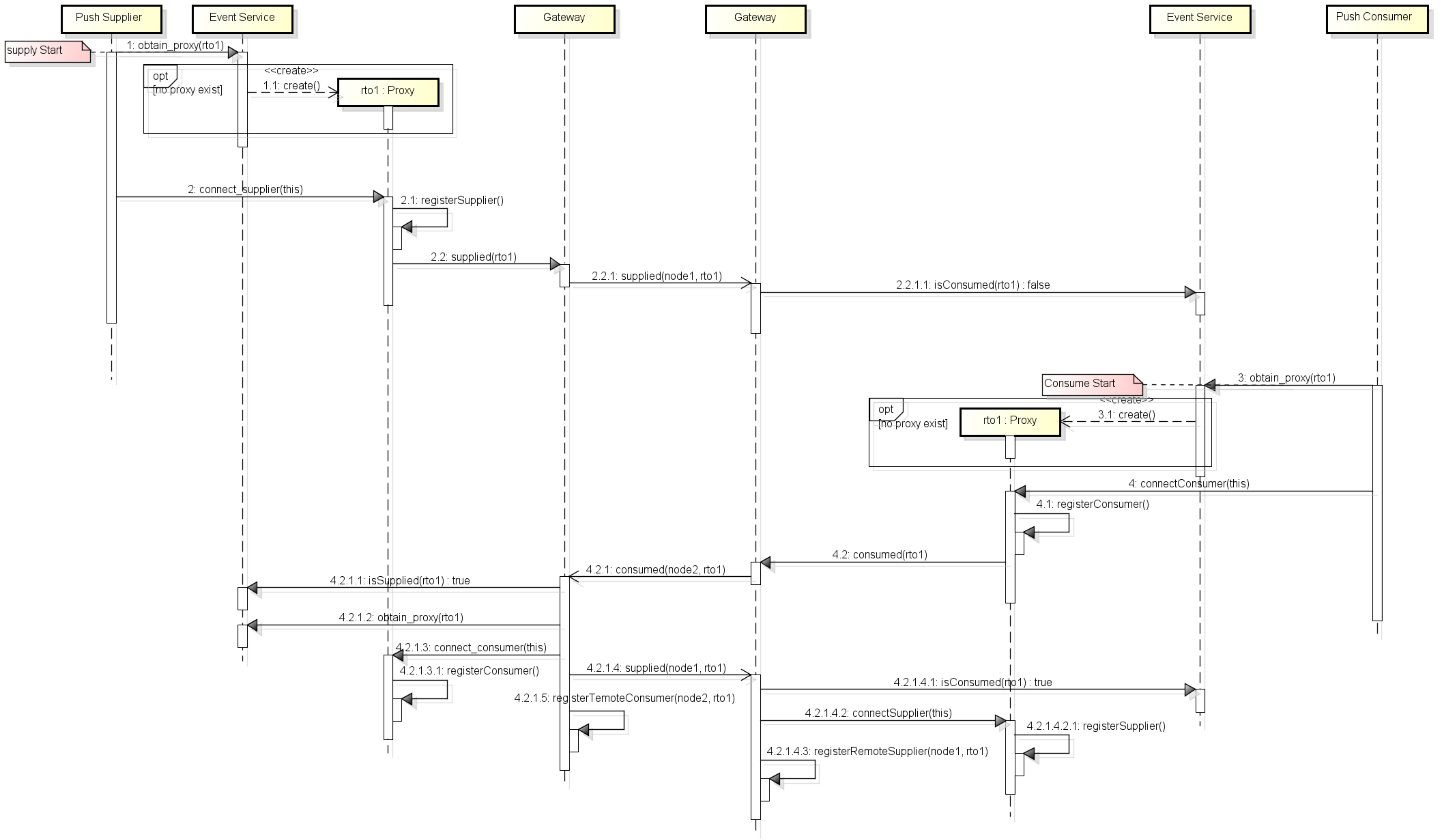

Push Supplier
Event Service
Gateway
Gateway
Event Service
Push Consumer
supply Start
1: obtain_proxy(rto1)
opt
[no proxy exist]
<<create>>
1.1: create()
rto1 : Proxy
2: connect_supplier(this)
2.1: registerSupplier()
2.2: supplied(rto1)
2.2.1: supplied(node1, rto1)
2.2.1.1: isConsumed(rto1) : false
Consume Start
3: obtain_proxy(rto1)
<<create>>
3.1: create()
rto1 : Proxy
4: connectConsumer(this)
4.1: registerConsumer()
4.2: consumed(rto1)
4.2.1: consumed(node2, rto1)
4.2.1.1: isSupplied(rto1) : true
4.2.1.2: obtain_proxy(rto1)
4.2.1.3: connect_consumer(this)
4.2.1.3.1: registerConsumer()
4.2.1.4: supplied(node1, rto1)
4.2.1.5: registerTemoteConsumer(node2, rto1)
4.2.1.4.1: isConsumed(rto1) : true
4.2.1.4.2: connectSupplier(this)
4.2.1.4.2.1: registerSupplier()
4.2.1.4.3: registerRemoteSupplier(node1, rto1)

Figure 6-11 After optimisation example of how the Event Service can setup a distributed Event Channel. The supplier first attach an event to the Event Channel and later the consumer attach to the same Event Channel

Figure 6-11 shows that setting up a distributed Event Channel, now Proxy, is almost similar to doing it before the last optimisation step though a bit more simple. But the push scenario are

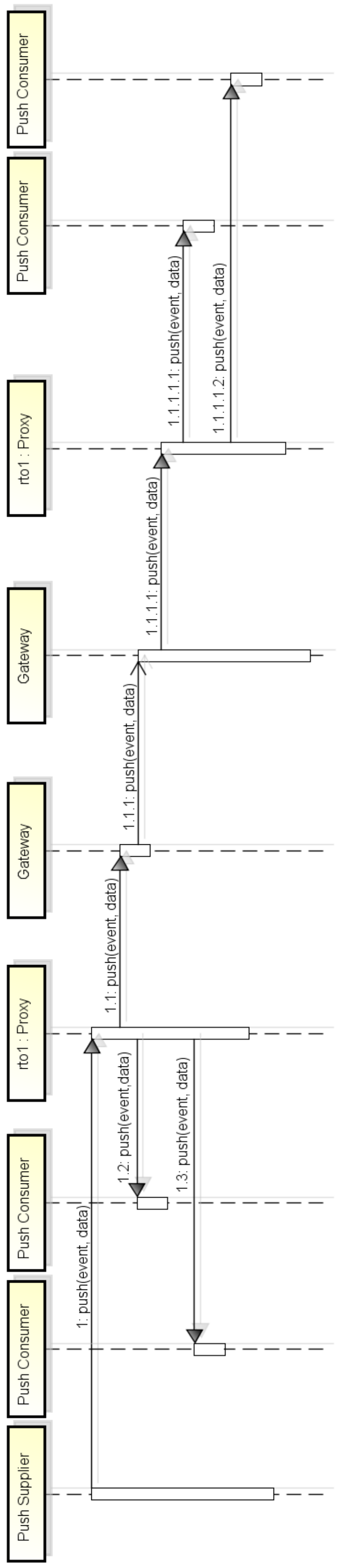


Figure 6-12 Push scenario after optimisation

much simpler as it can be seen in Figure 6-12. The push example is extended to two consumers on the same node as the supplier and two consumers on a remote node.

Optimization Step 3: The CORBA Event Service specification [6] doesn't specify priority of events. Real-time CORBA specifies priorities. In DRE-systems the bandwidth of the communication bus is often low, the consequence is that distribution of events with low importance for the application can delay distribution of events with high importance.

By giving events a priority when the event is supplied to the Event Service it is possible to distribute the events with high priority before low priority events.

Filtering of events is also an issue that can minimize the communication on the communication bus. E.g. if a supplier pushes an event, and shortly after pushes a new event to the same RTO before the first event is distributed by the gateway, then it can be argued that the first pushed event can be thrown away, and only the late pushed event need to be distributed.

Both the priority of events and the filtering of events can be designed into the Event Service. The major changes will be in the design of the Gateway.

The Gateway needs to queue events that should be distributed via the Com Service to the communication bus. By implementing two queues, a high priority and a low priority queue, instead of just one, it will be possible to first distribute high priority events before low priority events. To implement the event filtering the queues should not hold events but only event ids.

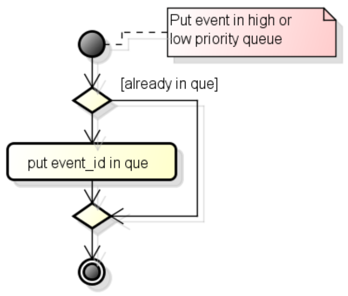


Figure 6-13 Add event to queue with filtering

When an event must be added to one of the two queues the en-queue function will only add the event id to the queue if it is not already queued (see Figure 6-13). When the Com Service is ready to distribute an event on the communication bus, it will ask the Gateway for the next event to distribute. The Gateway will return the next high priority event if any or the next low priority event if any or no events if both queues are empty (see Figure 6-14). Notice that the de-queue function only holds the id of the events and fetches the actual event (latest event) from the Proxy that handles the specific events.

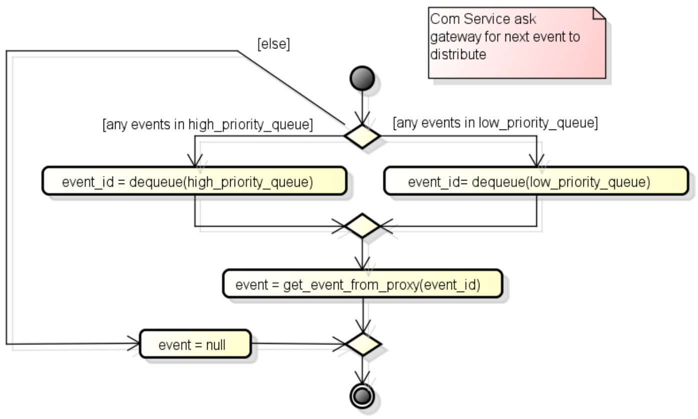

Com Service ask gateway for next event to distribute
[else]
[any events in high_priority_queue]
[any events in low_priority_queue]
event_id = dequeue(high_priority_queue)
event_id= dequeue(low_priority_queue)
event = get_event_from_proxy(event_id)
event = null

Figure 6-14 De-queuing events from the gateway queues

The result of fetching the events from the Proxy will be that it is always the newest/latest event that will be distributed, and that the Proxy can be updated many times between the events are actually distributed on the communication bus. This filtering function will also limit the necessary size of the queues.

Optimisation step 4: In many DRE-systems it is important to know if some of the RTOs are updated regularly as specified/expected. Often it complicates the design and lowers the performance of the system if it is up to the application program to check if RTOs are updated periodically. This service can be built into the Event Service as a QoS.

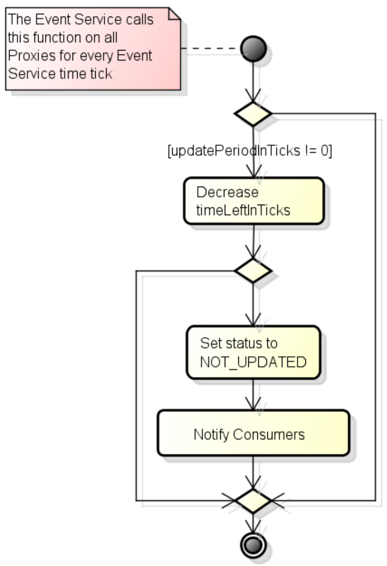

The Event Service calls this function on all Proxies for every Event Service time tick
[updatePeriodInTicks != 0]
Decrease timeLeftInTicks
Set status to NOT_UPDATED
Notify Consumers

Figure 6-15 Event Service Timer tick function

A supplier connected to an Event Proxy can specify an update period for the Proxy/RTO in question, then it is up to the Event Service to check if events are pushed regularly with a period lower than the specified update period. If the supplier of events fails to send an event in time, the Event Service will set the status of the Proxy/RTO to TIMED_OUT and push an event to all involved Consumers that the status of the RTO is now TIMED_OUT, and thus the value can't be used see Figure 6-15.

Optimisation step 5: If a node fails to communicate with the rest of the system, a QoS function can be built in to the Event Service to handle this situation. Figure 6-16 shows the local Gateway

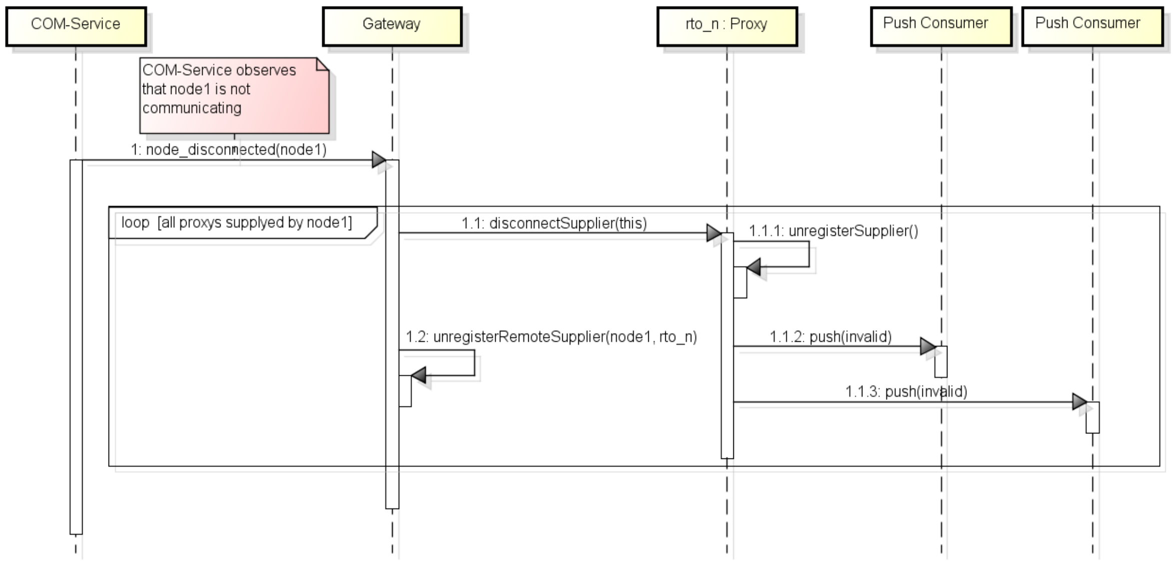

COM-Service
Gateway
rto_n : Proxy
Push Consumer
Push Consumer
COM-Service observes that node1 is not communicating
1: node_disconnected(node1)
loop [all proxys supplyed by node1]
1.1: disconnectSupplier(this)
1.1.1: unregisterSupplier()
1.2: unregisterRemoteSupplier(node1, rto_n)
1.1.2: push(invalid)
1.1.3: push(invalid)

Figure 6-16 Handling of node failure

being notified by the COM-Service, that there is no connection to a given node, and how all Proxies supplied by the failing node are notified (1.1). The Proxy then invalidates the event data and unregisters the supplier (1.1.1), and notifies all Consumers that the event data is invalid (1.1.2). Finally the Gateway unregisters the failing node as remote supplier (1.2).

After handling of the node failure, the Event Channels will be back in a state where there is no suppliers registered but only the local Consumers.

When the communication with the failing node is re-established, the Event Channel must be set up from scratch. This minimises the need for saving states in the Event Service and minimise the risk for the federated Event Service going out of sync.

It will be nearly the same scenario if a supplier voluntarily stops as supplier for an RTO see Figure 6-18.

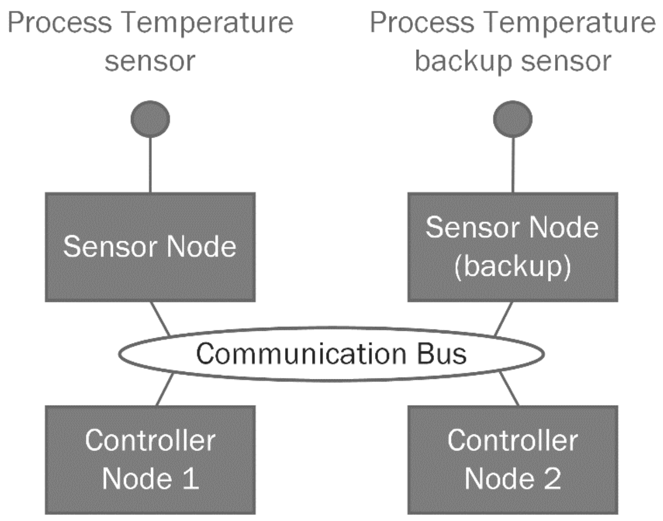


Figure 6-17 Simple example application

As it can be seen is the Event Service and the Event Service Gateway completely cleaned up. This opens up for the possibility to implement applications that are self-organising. An example could be a distributed system with four nodes where it is very important to keep track of processes temperature, see Figure 6-17. Two sensor nodes (Sensor Node and Sensor Node (backup)) could each have temperature sensor measuring the process temperature. The two Controller nodes uses this temperature to control the process. If one of the two temperature sensors fails, and one of the nodes in the system can detect the problem, then the sensor node with the failing sensor, can stop supplying the temperature value, and the backup sensor node can start supplying the temperature value.

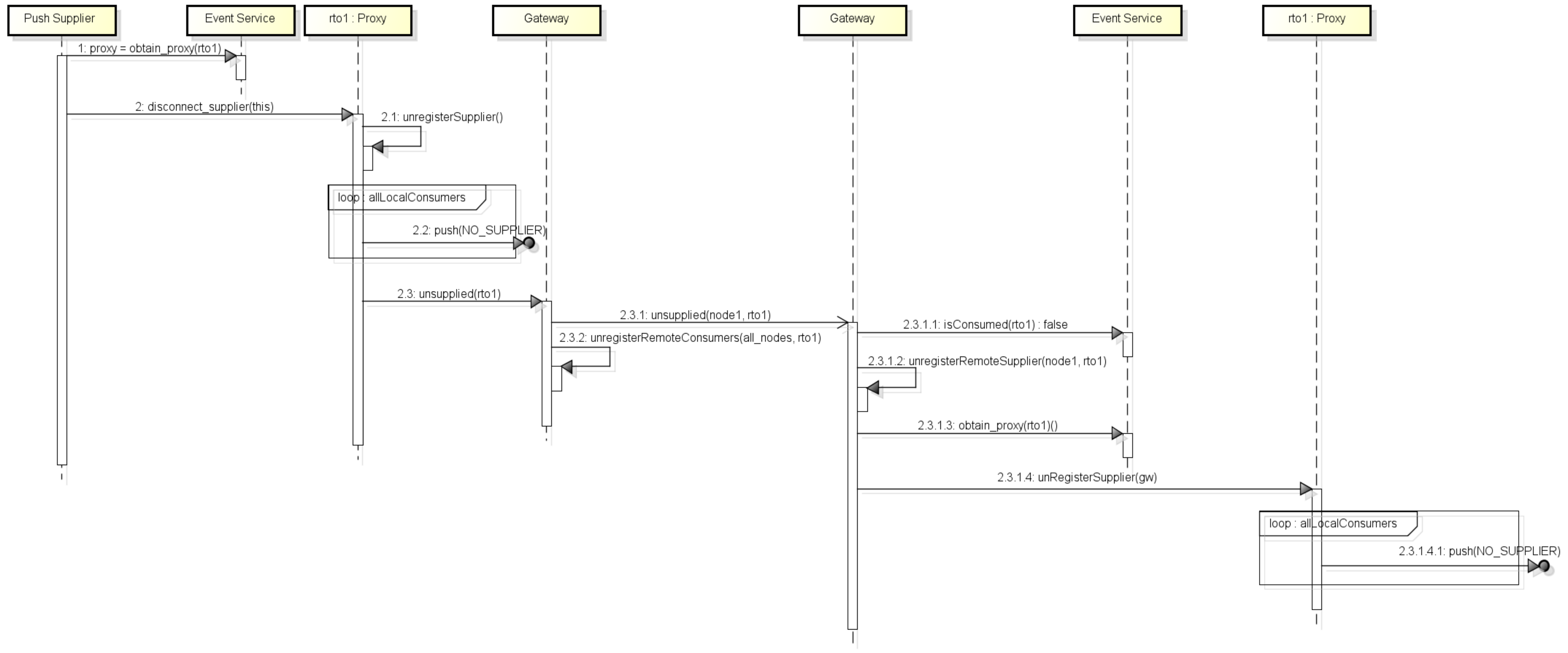

Push Supplier
Event Service
rto1 : Proxy
Gateway
Gateway
Event Service
rto1 : Proxy
1: proxy = obtain_proxy(rto1)
2: disconnect_supplier(this)
2.1: unregisterSupplier()
loop : allLocalConsumers
2.2: push(NO_SUPPLIER)
2.3: unsupplied(rto1)
2.3.1: unsupplied(node1, rto1)
2.3.2: unregisterRemoteConsumers(all_nodes, rto1)
2.3.1.1: isConsumed(rto1) : false
2.3.1.2: unregisterRemoteSupplier(node1, rto1)
2.3.1.3: obtain_proxy(rto1)()
2.3.1.4: unRegisterSupplier(gw)
loop : allLocalConsumers
2.3.1.4.1: push(NO_SUPPLIER)

Figure 6-18 A supplier voluntarily stops supplying a RTO

6.4.1 Proxy Design

The Proxies contains the attributes that can be seen in Table 2. A proxy is one of the corner stones in the Event Service, it is the proxy that holds all information about an RTO, including who are the consumers, and who can/will supply it.

Table 2 Description of Proxy contents.

Attribute	Description
RTO ID	ID of the RTO this proxy is holding
Value	A pointer to the value the RTO contains
Status	Current status of the RTO
Supplier ID	ID of the supplier of this RTO
Consumer list	List of current consumers of this RTO
Update ticks left	Update ticks left before the RTO is not valid
Meta data attribute	
Update period in ticks	The maximum number of timer ticks between the RTO must be updated. (0 => timeout is disabled)

The Proxy is implemented as an Abstract Datatype (ADT).

The Proxy uses a simple form of Observer pattern to notify the Consumers about a change in an RTO.

6.5 Event Channel Gateway Design

The Event Channel Gateway (ECGW) is the module sitting between a nodes Event Service and COM-Service. Figure 6-19 shows a simplified block diagram of the Event Channel Gateway,

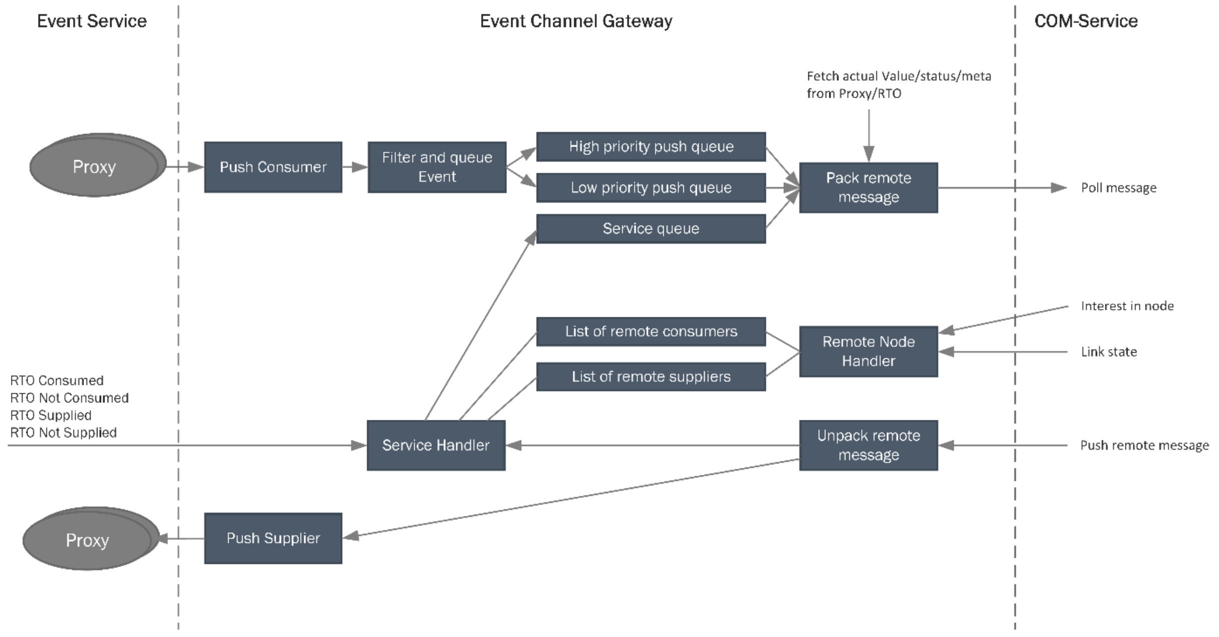


Figure 6-19 Overview of the Event Channel Gateway

this figure can be used as reference when reading the description of the Event Channel Gateway below.

6.5.1 Messages to COM-Service

Two classes of messages will be send via the COM-Service to the network. *Push messages* and *Service messages*. The format of these messages is explained in 6.5.1.1.

Push messages are messages that distributes events on RTO's. These messages contains either a new state, a new status or new meta-data of an RTO.

Service messages are messages that set up Event Channels between the COM-Service and the local Event Service.

- Supplied
- Un-Supplied
- Consumed
- Un-Consumed

Push messages have the highest priority, and will always be sent before Service messages. Furthermore the Event Service will internally prioritises push messages in high and low priority, but this is transparent to the COM-Service.

6.5.1.1 Message format

The message format sent between the Event Channel Gateway and the COM-Service can be seen in Figure 6-20.

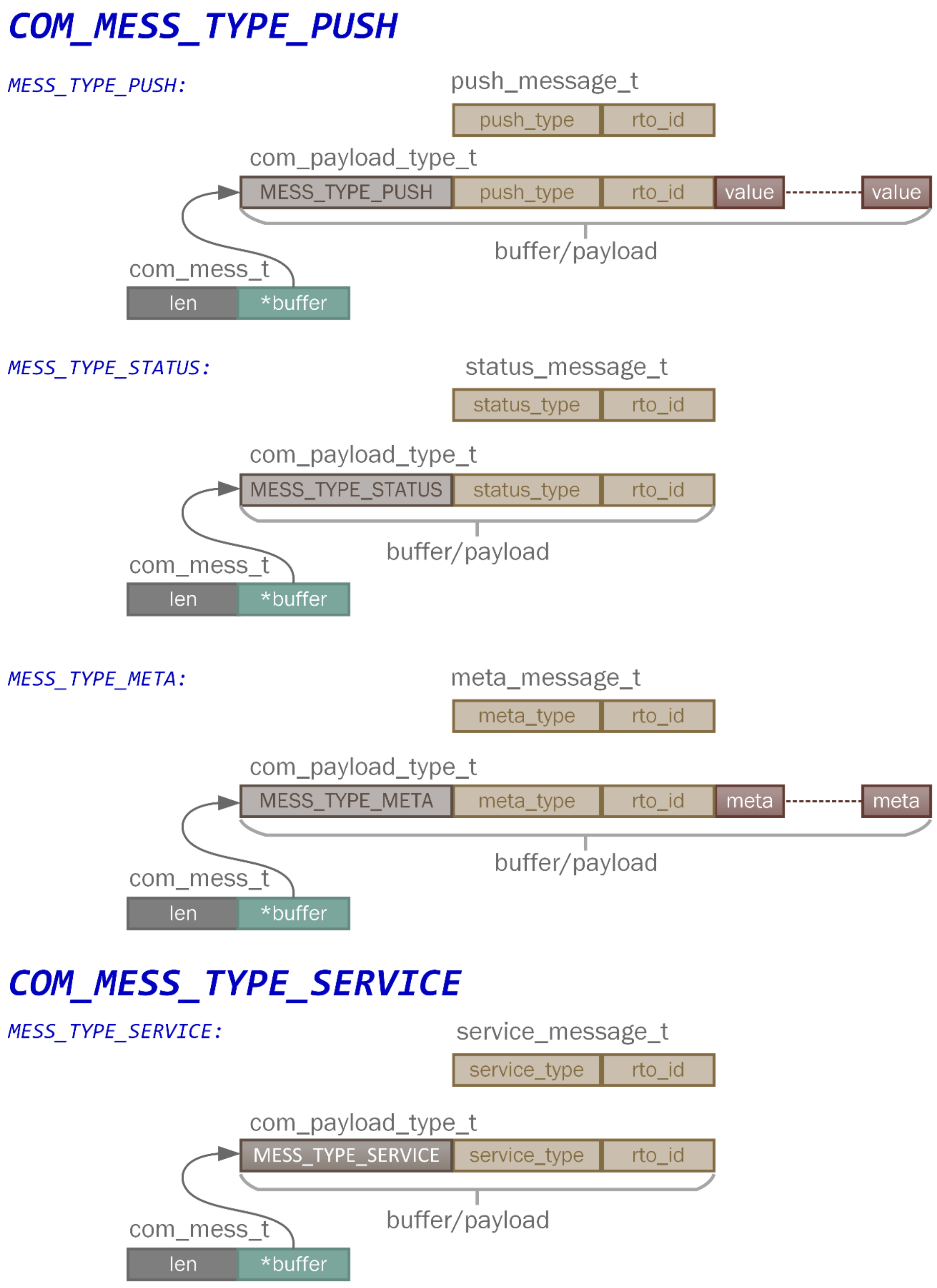


Figure 6-20 Message format between Event Channel Gateway and COM-Service. Shows also the internal format used inside the Event Channel Gateway.

Messages sent between the ECGW and the COM-Service can have two different overall types: COM_MESS_TYPE_PUSH and COM_MESS_TYPE_SERVICE.

Push messages are messages that contain value, meta-data or status changes to Proxies/RTOs. Service messages contain consume/supplied information to RTOs.

6.5.2 Interface to Communication Service

The interface (IECGW) between Event Channel Gateway (ECGW) and Communication Service is implemented in four function calls. All functions are implemented in the Event Channel Gateway, and called from the Communication Service when a corresponding event happens.

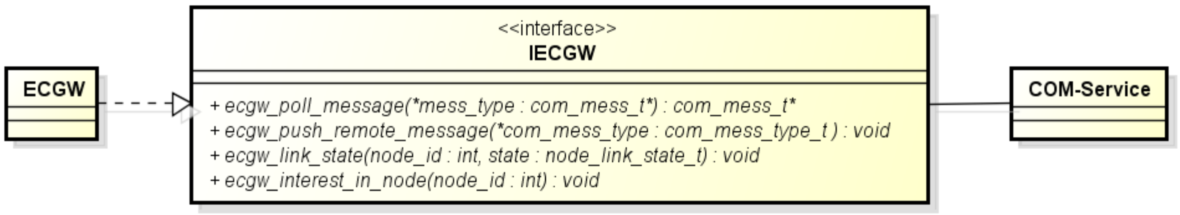

ECGW
<<interface>>
IECGW
+ ecgw_poll_message(*mess_type : com_mess_t*) : com_mess_t*
+ ecgw_push_remote_message(*com_mess_type : com_mess_type_t) : void
+ ecgw_link_state(node_id : int, state : node_link_state_t) : void
+ ecgw_interest_in_node(node_id : int) : void
COM-Service

Figure 6-21 Event Channel Gateway interface

At regular time intervals, the Communication Service polls a message queue with the *poll_message(messag_type)* call. This function will return the next message to be sent and a type specification. The type specification is used by the receiving node to filter away messages that are of no interest to it. This is an optimization to save resources in the receiving node, as it can stop processing frames that would otherwise be discarded later by the Event Channel.

Each Event Channel Gateway maintain a table of which nodes are supplying which RTOs that this node consumes. Likewise a table of which nodes are consuming RTOs that is supplied from this node.

The Communication Service can query this table with a call to *interest_in_node(node_id)*. This returns false, if the Event Channel Gateway does not consume any RTOs from the node specified as the parameter. In this case the Communication Service can discard the frame without wasting resources on RTOs that are not consumed by this node.

When an RTO has been received by a node, it is delivered to the Event Channel Gateway by a call to *push_remote_message(message_type, message).*

The Communication Service can inform the Event Channel Gateway about the connection state of a node with a call to *link_status(node_id, state)*, where state indicates CONNECTED or DISCONNECTED.

6.6 Communication Service Design

The communication service is in parts inspired by the Local Interconnect Network (LIN) standard [3], used in the automotive industry. There are several advantages of this technology; it uses ordinary UARTs that is integrated in most low cost MCUs (although a LIN line driver circuit is necessary), messages can be broadcast to more subscribers simultaneously thus saving costly bandwidth, and achieve predictable end-to-end transmission times.

One disadvantage is that it relies on a single master node to control the communication schedule. This makes it vulnerable to a single point of failure. This issue will not be resolved in this thesis, although a discovery of a malfunctioning node or communication link will be reported to the Event Service, see 6.6.4 Error handling.

There are a few reasons for not implementing the full LIN standard. In this project, we focus on preserving the original hardware configuration of the Hoist system, which use RS-485 physical layer, that can't produce the special break field used to indicate the start of a frame. Also, the Event triggered frames are excluded, as they will make communication delays non-deterministic. Some optimization is introduced to decrease the time spent on receiving frames that are of no interest to a node.

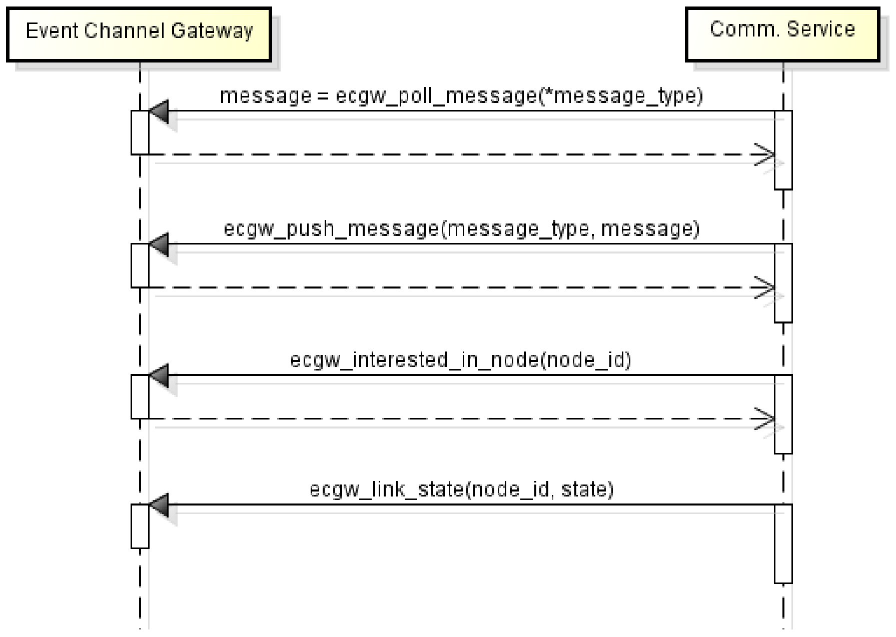

Event Channel Gateway
Comm. Service
message = ecgw_poll_message(*message_type)
ecgw_push_message(message_type, message)
ecgw_interested_in_node(node_id)
ecgw_link_state(node_id, state)

Figure 6-22 Interface between Event Channel Gateway and Communication Service

The Link layer of the communication service is tightly coupled with the physical layer, and a change of network technology on this level will also lead to changes here, so the following is specifically designed with the Hoist system in mind.

The system consists of a cluster with one Master and several slave nodes (Figure 6-23). The Master controls the communication schedule, i.e. decides when and which frame to be transferred on the bus. The slave tasks provide the data frames. This architecture is simpler and more predictable than a multiple access protocol, which could lead to collisions. The disadvantage is longer response time for sporadic events as they can only be transferred on the bus according to the schedule. To make up for this, a higher transmission rate on the bus will be allowed.

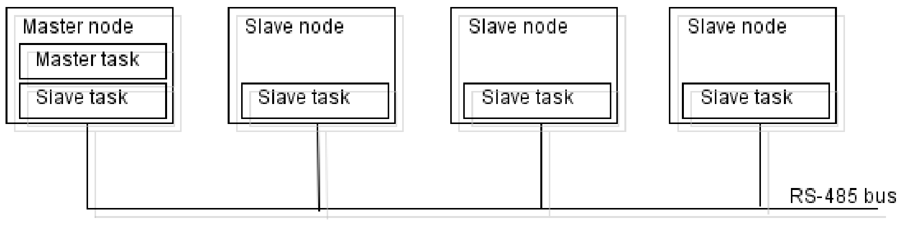


Figure 6-23 Node cluster

6.6.1 Task behaviour model

As in LIN, the Master task has a simple job; to transmit the header whenever a node is due for transmission. This is illustrated by the state machine in Figure 6-24. A *time_tick* brings the task into *scheduling* state. It will immediately leave this state again if a transmission from a slave task is still in progress. Otherwise the next node from the schedule is addressed and the header sent. The state then returns to *waiting*.

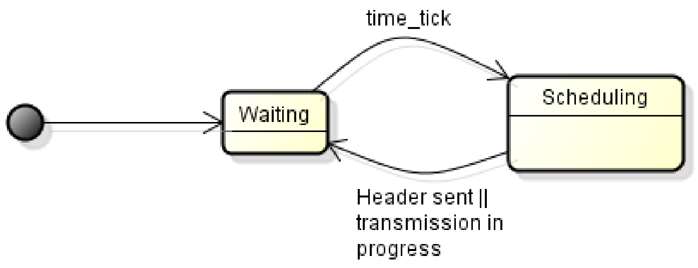


Figure 6-24 Master task state machine

The Slave task is more complex as it is responsible for reception and transmission of the response part of frames. It must synchronize on the reception of a Break field and receive the PID correct. The Break field is unique for the communication and will always be interpreted as start of frame. Should a Break field be received during reception/transmission of a response, the slave must abort the operation and discard the frame. Figure 6-25 illustrates the state machine of the frame logic in a Slave task. It has two outer states; Idle and Active. It waits in Idle until a Break field is received, and then enters Active, to process the frame. The sub-states in Active can exit with different status caused by different events during reception or transmission. The behaviour will be discussed in the following sections.

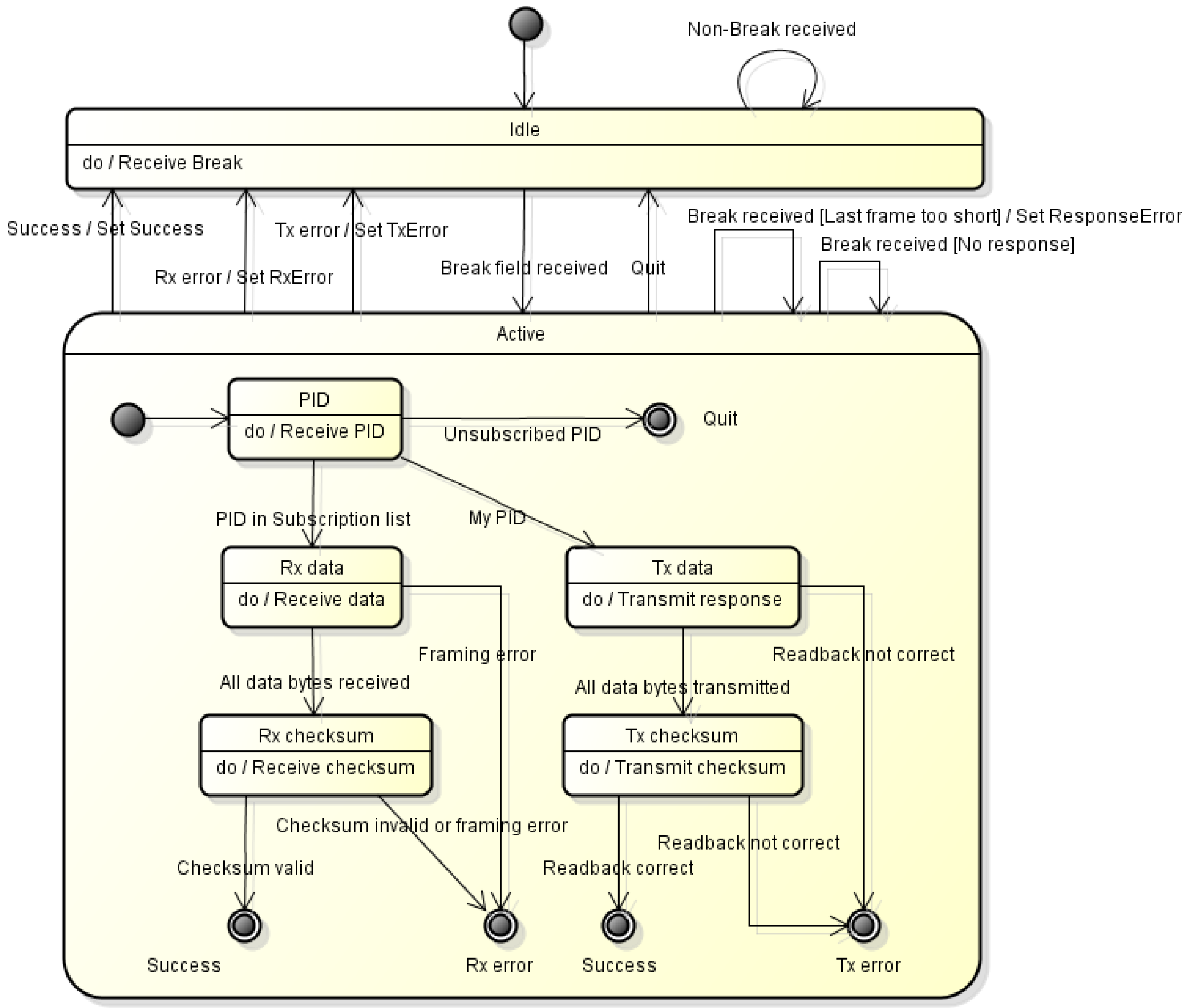


Figure 6-25 Slave task state machine

6.6.2 Physical Layer

The physical layer is implemented as a broadcast bus. Standard RS-485 line interface is used, with differential signals, thus giving a good S/N ratio. Compared to LIN, which will run on a single signal wire, RS-485 requires a twisted pair, which makes it possible to increase the transmission rate substantially. Usually UARTs in small Microcontrollers can achieve Baud rates in the order of hundreds of kbps which is high compared to LIN's 20 kbps.

Given the relative short distances, the physical bus is relatively robust. With ordinary EMI (Electro Magnetic Interference) shielding of the hoist system, the error rate is assumed to be very low, and total packet loss is expected to happen rarely, unless a node or the bus malfunctions.

Other systems, though, may work under different conditions, so some error detection is done. Single bit errors can be detected directly by the UART but the decision on what to do about it involves the MCU. Therefore error handling must be done on the Link layer.

6.6.3 Link Layer

6.6.3.1 Addressing strategy

Different addressing strategies have different challenges. E.g. with unicast addressing; if a given object is consumed by more than one node there could be a rather large delay from when the first node receives it until the last node receives it, depending on their relative position in the transmission schedule. The delays then depends on how many nodes are producers and how many are consumers. If anything is supposed to happen simultaneously on two different nodes the control becomes complex as the delays will have to be taken into account.

A way to avoid this delay and give all consumers the data simultaneously, is to broadcast the data frames.

With broadcast, at least parts of the frame must be read by all hosts to identify the source and/or object. Since the nodes do not have any dedicated communication controller, the CPU must be interrupted for each byte received. Therefor the frames must be kept short and match the most common object size to avoid too much overhead. Rare large objects could be segmented, and sent one after another in the node's allocated time slots.

6.6.3.2 Frame structure

A frame consists of a header and a response. The header is supplied by the master according to the schedule, and the response is supplied by the slave responsible for the scheduled object.

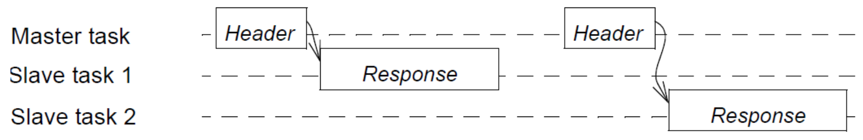


Figure 6-26 Frames

The header transmitted by the Master node has only two fields; *Break* field and *Protected identifier* field *(PID)*. In contrast to the LIN specification [16], *PID* here is a node ID rather than a frame ID. We wish to decouple the communication service from the object IDs, because another network technology might be used in another system.

The *Break* field identifies the start of a header. In the given system, byte value 0x00. This causes a minor problem, since data in the response is in binary format and the value 0x00 could appear in the data. To make the protocol transparent and still have a fixed frame size, the bytes in the response are base64 encoded. The penalty for this is that the response increases 33% in size.

Another way to deal with the illegal value problem, is to send all values as ASCII strings. An example of this is the NMEA-0183 **Invalid source specified.** standard used for e.g. communication with a GPS device.

This method is immediately rejected because it poses two problems; values have to be converted to strings, and string length depends on value.

First problem increases execution time. Per byte it will be less than for Base64 encoding but the amount of bytes will typically increase. A one byte value could translate into three characters.

Second problem could be solved by prefixing with zeros but then the frame increases even more, which increases the transmission delay.

An alternative way, used in e.g. Modbus RTU [17], is to define gaps of a certain length of silence between each frame. In this way the receiving host knows when a new frame starts by time instead of by value. The monitoring of gaps would also take up resources, e.g. an extra Timer. Which method is the best may depend on the available hardware resources.

The *PID* addresses the node allowed to send the response, and is read by all nodes on the bus. They use it to see if the following response will be carrying any Real Time Objects of interest for them.

The response consists of 3 fields; *Response Type*, *Data* and a *checksum*. There are three types of response; *Service*, *Push* and *None*. *Service* frames are used to establish a channel between two event services, and must be received and processed by all nodes. *Push* frames transfer Real Time Objects and are only processed by nodes that supply or consume the transferred data object. The *None* frame is an acknowledgement from the node that it is alive even though it has nothing to send.

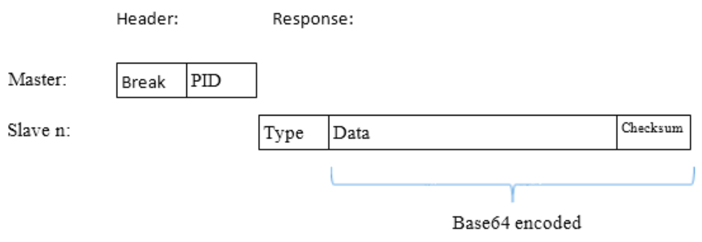


Figure 6-27 Frame structure of an Update or Setup frame

When a node wants to subscribe as *Consumer* on an object from a remote node, it transmits a *Service frame*. All other nodes listen to this frame type and deliver the response to the Event Channel Gateway. The Event Channel on the node that supplies the object prepares a reply to be sent in a *Service frame*. The Event Channel Gateway in each node will maintain a Subscription list with the supplier of each consumed RTO. This list will later be consulted by the Communication Service when updates of RTOs are received.

Figure 6-28 shows an example where Slave 2 wishes to subscribe to RTO1 supplied by Slave 1. The Master addresses Slave 2 according to schedule. The request is also received by Slave 1. Slave 2 responds with a *Service frame* containing a request for the desired object.

The *Service frame* is received both by the Master and Slave 1. Since it is a *Service frame*, they will both have to receive the message and deliver it to their Event Channel Gateway, but the Master does not supply this object, and ignores it. Slave 1 supplies the object, so it prepares a Supply message and puts it in the transmission queue.

Later Slave 1 is addressed by the Master. It transmits the Supply message from the queue in a *Service frame*. Both Master and Slave 2 receive this and deliver the message to their Event Channel Gateway. Both are now aware of that Slave 1 can supply RTO1.

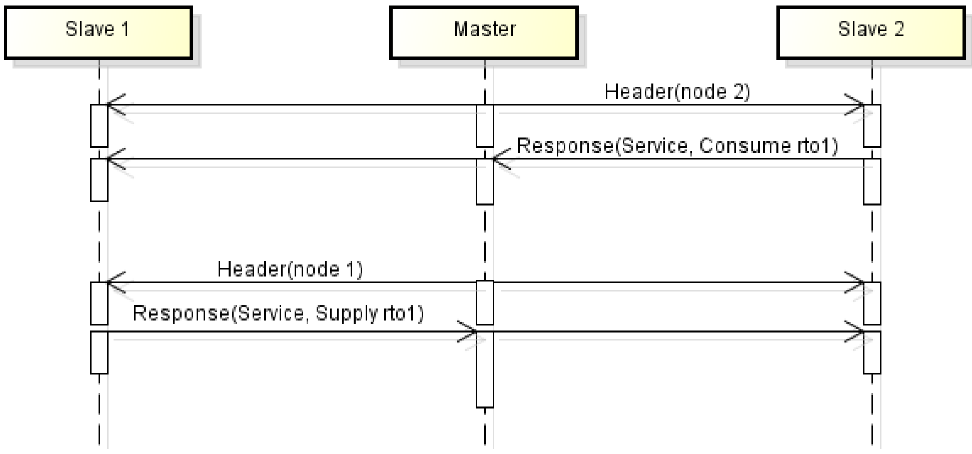

Slave 1
Master
Slave 2
Header(node 2)
Response(Service, Consume rto1)
Header(node 1)
Response(Service, Supply rto1)

Figure 6-28 Establishment of connection between Supplier and Consumer

Slave 2 can now start consuming RTO1 every time Slave 1 pushes updates.

Figure 6-29 shows an example of how a Real Time Object is transferred from slave 1 to slave n after the Consume and Supply messages have been exchanged.

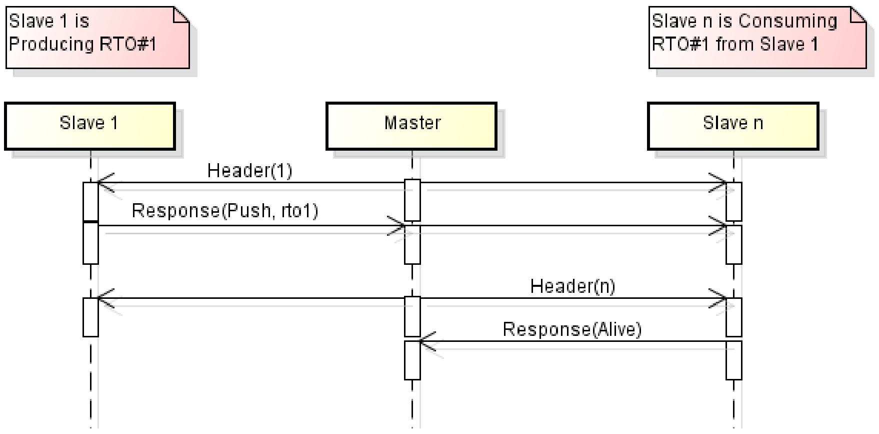


Figure 6-29 Exchange of Real Time Objects

The Master, who controls the communication schedule, sends the header with a PID identifying Slave 1. Slave 1 puts a *push* response on the bus with a Real Time object; RTO1. Since the header is broadcast on the bus Slave n also receives it, and can, by looking into the received Response Type field see that this is an update on objects from Slave 1. By a look-up in its subscription list, it sees that it subscribes to objects from Slave 1 so it reads the entire response and delivers the Real Time Object to the Event Service. The Master node is not subscribing any objects from Slave 1, and since it was a push response, it ignores the message.

6.6.4 Error handling in link layer

Given the robust physical layer and short distances, error control in the Link layer can be relatively simple.

With RS-485, it is possible to listen to the bus while transmitting. In this way the sending host will receive its own transmission. If the received frame is different from the transmitted a communication error has occurred, e.g. EMI[5] could have garbled the transmission, or a Slave started to transmit outside its allowed time frame.

The Master will do this read-back to make sure that the header was transmitted correct on the bus. If an error is detected it will abort the transmission of the current header and wait for the next scheduled one. The idea behind this simple approach is that a short current spike from e.g. relays and motors, might corrupt data on the bus. An immediate retransmission might therefor also get corrupted.

Even though the node address in the *PID* is protected by two parity bits an error could still change it to another valid value. If an unscheduled node then starts to transmit, it can't be stopped as the nodes themselves don't know the schedule. In this case, a slave got an extra chance to transmit.

The slaves will also listen to their own transmission and abort if an error is detected. The frame will then be too short. This will be detected when the next header is transmitted. A node that receives the Break character, must discard an unfinished frame and start processing the new one. The node that aborted transmission must then hold back the frame until next time it is addressed.

A transmitting slave node must abort if it receives the Break character. This indicates that it took too long time to transmit the frame.

The nodes will detect a garbled response frame by calculating and comparing the checksum. The checksum calculation is done as what in the LIN standard is called an extended checksum. An example of calculating this is given in 4.3.2. The checksum covers the *PID* in the header and all bytes in the response frame. In this way the checksum only matches if it was sent as response to the corresponding header. A delayed response frame from another slave would fail to produce the correct checksum.

In case of checksum error the frame is discarded by the receiving nodes. An immediate retransmission of a failed frame would mean a temporary change in the communication schedule. This would create jitter on the communication, which could be problematic since it involves the whole cluster.

The Master therefore sticks to the schedule, so only the nodes consuming the fail-transmitted RTO are affected.

If the fault causing a checksum error is on the communication bus, it will also be detected by the transmitting node (as it is listening to its own transmission). The failed frame will then be retransmitted in the next round.

In case a slave is not responding with a frame, when addressed, the Master continues with the schedule. After 4 failed attempts the slave is reported "disconnected" to the Event Channel Gateway. The Master will continue to address the slave so it can be detected if it comes online again.

Other Slaves that consume objects from a malfunctioning node, will also detect the missing response frame, and in the same way as the Master reports the node offline to their local Event Service.

[5] Electromagnetic Interference

Table 3 Error messages reported to the Event Service

Status and error message	Meaning
DISCONNECTED, n	Node n is not responding on requests (or checksum error on frames from that node)
CONNECTED, n	Node n is responding on requests, after it was offline.

6.6.4.1 Checksum calculation [18]

The purpose of the checksum is to detect if errors have altered bits in the frame. There are many ways to calculate this. Often Cyclic Redundancy Check (CRC) is used to produce a Frame Check Sequence that is appended to the frame. They are very effective in detecting both random bit and burst errors. CRCs are based on polynomial divisions, and are therefore computational costly. Simpler checksums are often used in small embedded systems, e.g. XOR, 2-complement sum or 1-complement sum.

A comparison between different checksum types is beyond the scope of this thesis, but have been done in [19]. This study concludes that a 1's complement addition is the best trade-off between error detection and computation cost. It has more or less the same computational cost as both XOR and 2-complement sum, but is detecting random bit errors more effectively.

1-complement addition is the method used in LIN, and is also adopted by this system.

6.6.5 Frame timing

Latency in the communication service depends on the schedule, which again depends on number of nodes and the designer's choice of priority for a node to access the bus. E.g. one slave which produces objects with a high rate could be addressed, by the schedule, more often than one that produces objects with a lower rate.

A schedule could also address the slaves in a simple round robin. This would give all slaves the same priority. Figure 6-30 shows an example of this.

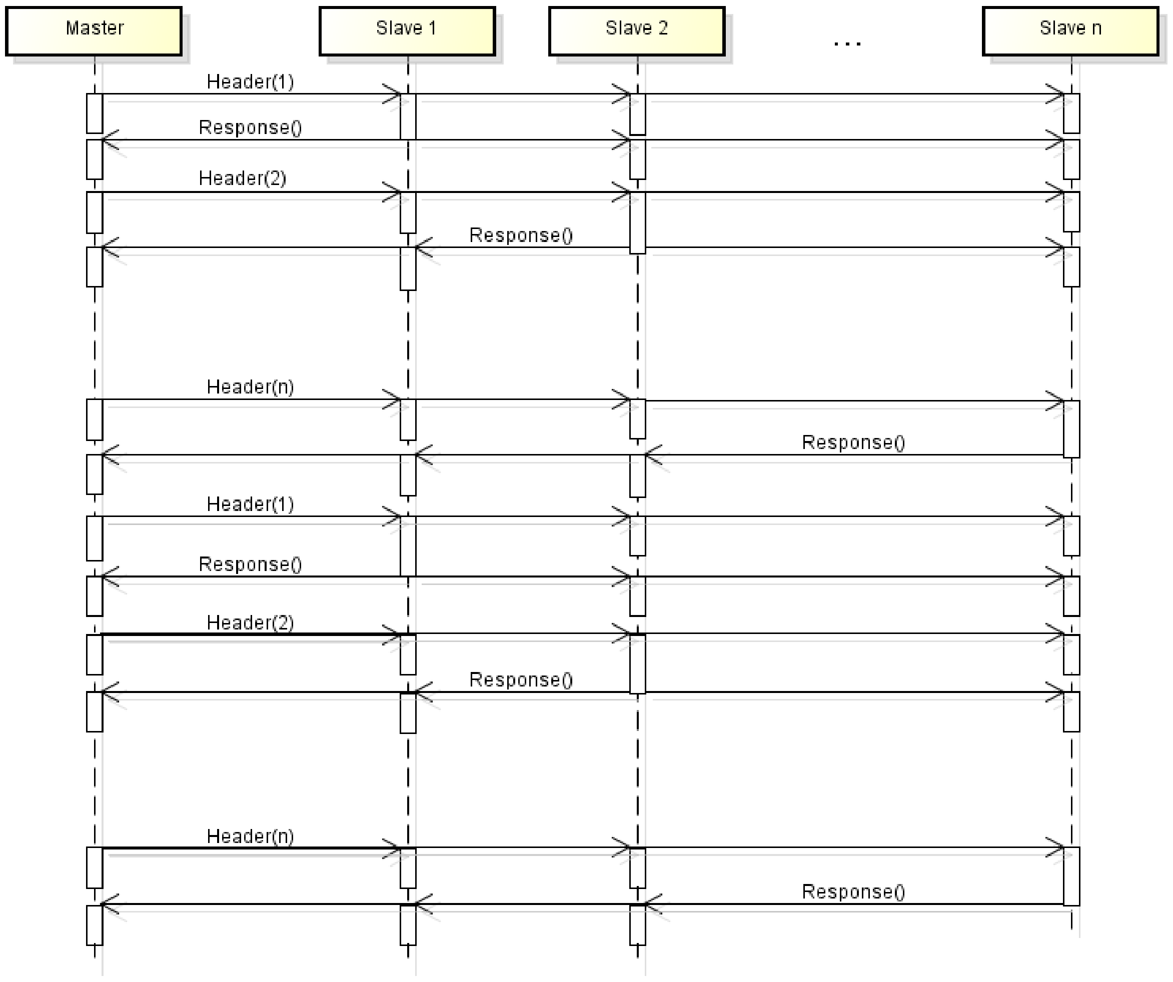


Figure 6-30 Simple round robin schedule

A Header is two bytes; *Break* and *PID*.

The length of the response depends of the configurable data field length but for the calculation examples here it is 10 bytes in total, one for *Response Type*, 8 for *data* and one for *checksum*. The *data* and *checksum* fields are encoded as Base64 before transmission. This increases the response to 13 bytes.

Each byte is transmitted by the UART as 10 bit (1 start, 8 data, 1 stop), so the nominal transmission times then becomes

$$T_{Header\_Nominal} = \frac{10 * (1_{Break} + 1_{PID})}{Baud\ rate}$$

$$T_{Respone\_Nominal} = \frac{10 * 13_{Base64}}{Baud\ rate}$$

$$T_{Frame\_Nominal} = T_{Header\_Nominal} + T_{Response\_Nominal}$$

In compliance with the LIN specification, an additional 40% transmission time allows for some space between the bytes and gaps between header and response, so

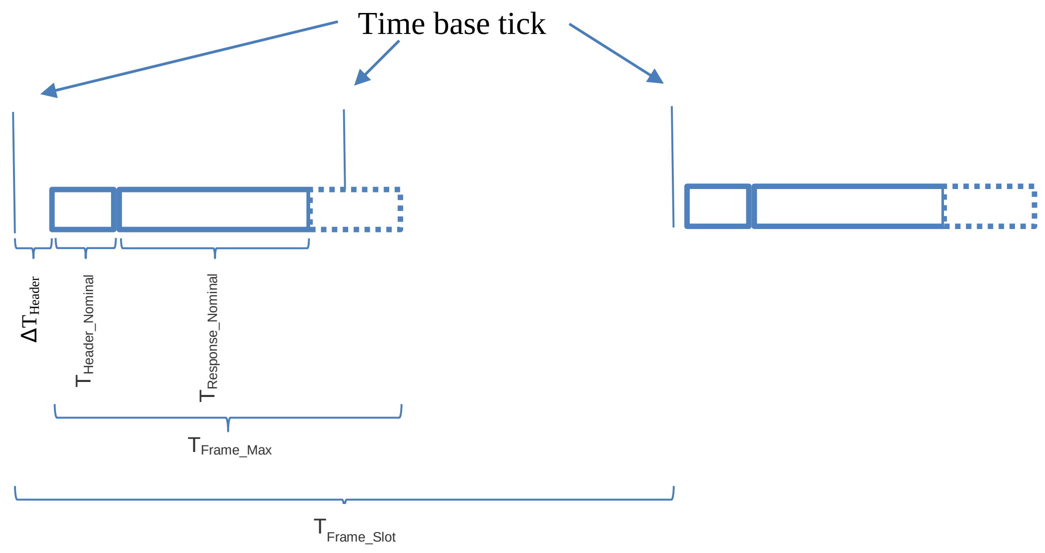
$$T_{Frame\_Max} = 1.4 * T_{Frame\_Nominal}$$

The Master node defines a time base, T_{Base}, used to control the timing of the schedule. The start of the time base is defined as a time base tick. A frame slot, T_{Frame_Slot}, is an integer multiple of the time base. A frame slot always starts at a time base tick. The variation of time from a time base tick and the start of header transmission is jitter, ΔT_{Header}.

Jitter could occur if some higher priority interrupts, e.g. sampling of signals with constant rate is allowed to block transmission.

The frame slot must be larger than the maximum transmission time of a frame plus jitter.

$$T_{Frame\_Slot} > \Delta T_{Header} + T_{Frame\_Max}$$



For the receiving nodes, jitter is the sum of ΔT_{Header} and variation in the transmission of the frame, ΔT_{Frame}, which is 40% of the nominal transmission time.

The latest time an application can update an RTO is as follows:

- Master node – before the frame transmission is initiated.
- Slave node – before *PID* is received.

With n nodes, one round in the schedule takes $n*T_{Frame\_Slot}$.

6.6.6 Sending a frame

A frame can be transmitted when a node receives a header with its own *PID*. It polls the next message to be sent from the Event Channel Gateway. The checksum is calculated and placed in the frame before it is encoded into Base64 format. The encoded frame is then sent to the UART (Universal Asynchronous Receiver Transmitter) driver's transmit buffer. Once the first byte is written transmission on the bus begins. When the byte has been transferred an interrupt triggers loading the next byte into the UART transmit register. This continues until the transmit buffer is empty, which is when the whole frame has been sent.

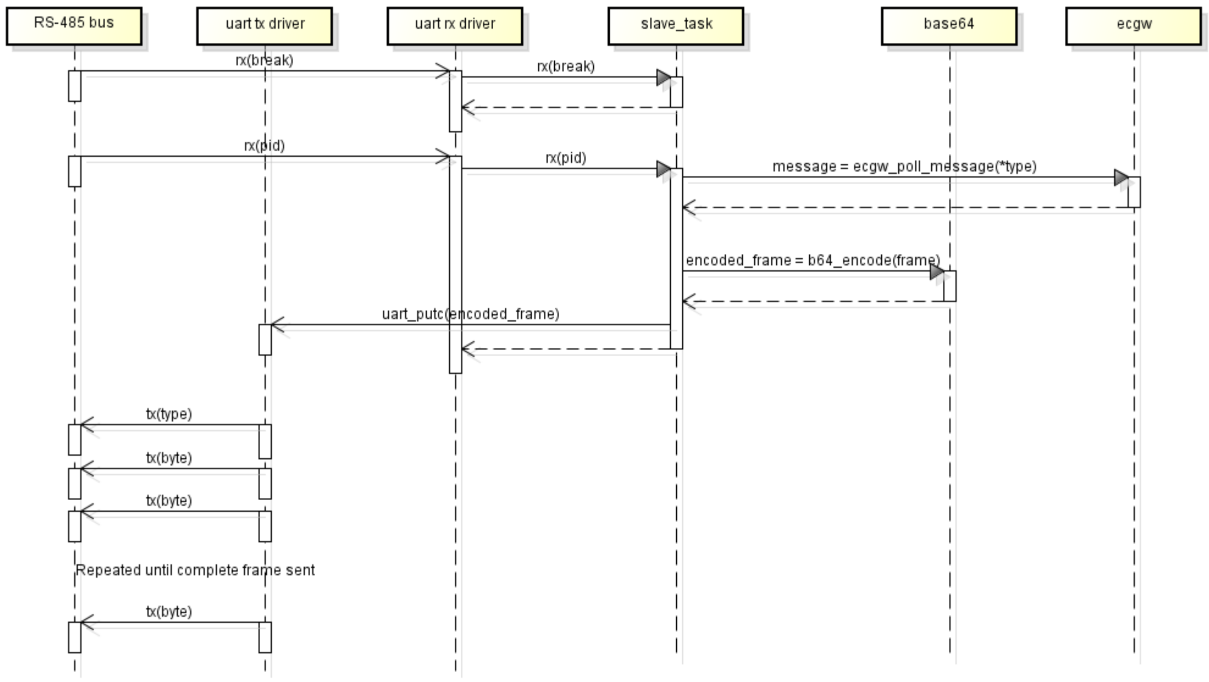


Figure 6-31 Receiving own PID triggers transmission of a message

6.6.7 Receiving a frame

The slave task receives frames and delivers them to the Event Channel Gateway. The interface function *ecgw_interested_in_node()* is used to filter away messages of no relevance for the receiving node. If the Event Channel Gateway is not interested in the frame, execution time in the *slave_task* is reduced considerably as it returns to IDLE state (see Figure 6-25) and ignores everything until the next *Break*.

A sequence diagram showing an example of a node that subscribes on objects from another node, is given in Figure 6-32. The receiving node makes use of the Event Channel Gateway function; *ecgw_interested_in_node().*

The slave task is triggered by interrupt from the UART. Each byte received on the bus, interrupts the CPU, and the Interrupt Service Routine moves the byte from the UART's receive register to the slave task. The slave task is part of the interrupt routine, so it is important that it is effective in order not to block other interrupts.

The function *slave_task* returns; Save or DROP to the UART driver. If the received byte is a part of the response frame, it should be saved in the ring-buffer implemented in the driver. Otherwise it will be dropped. When the complete frame has been received, *slave_task* can fetch it from the ring-buffer.

The frame is encoded in Base64 format, and must be decoded before the checksum can be verified. If the checksum is verified the message is delivered to the Event Channel Gateway by calling the *ecgw_push_message(...).*

In case the checksum verification fails, the frame is dropped. It will be retransmitted by the sending node automatically, if the cause was detectable on the bus.

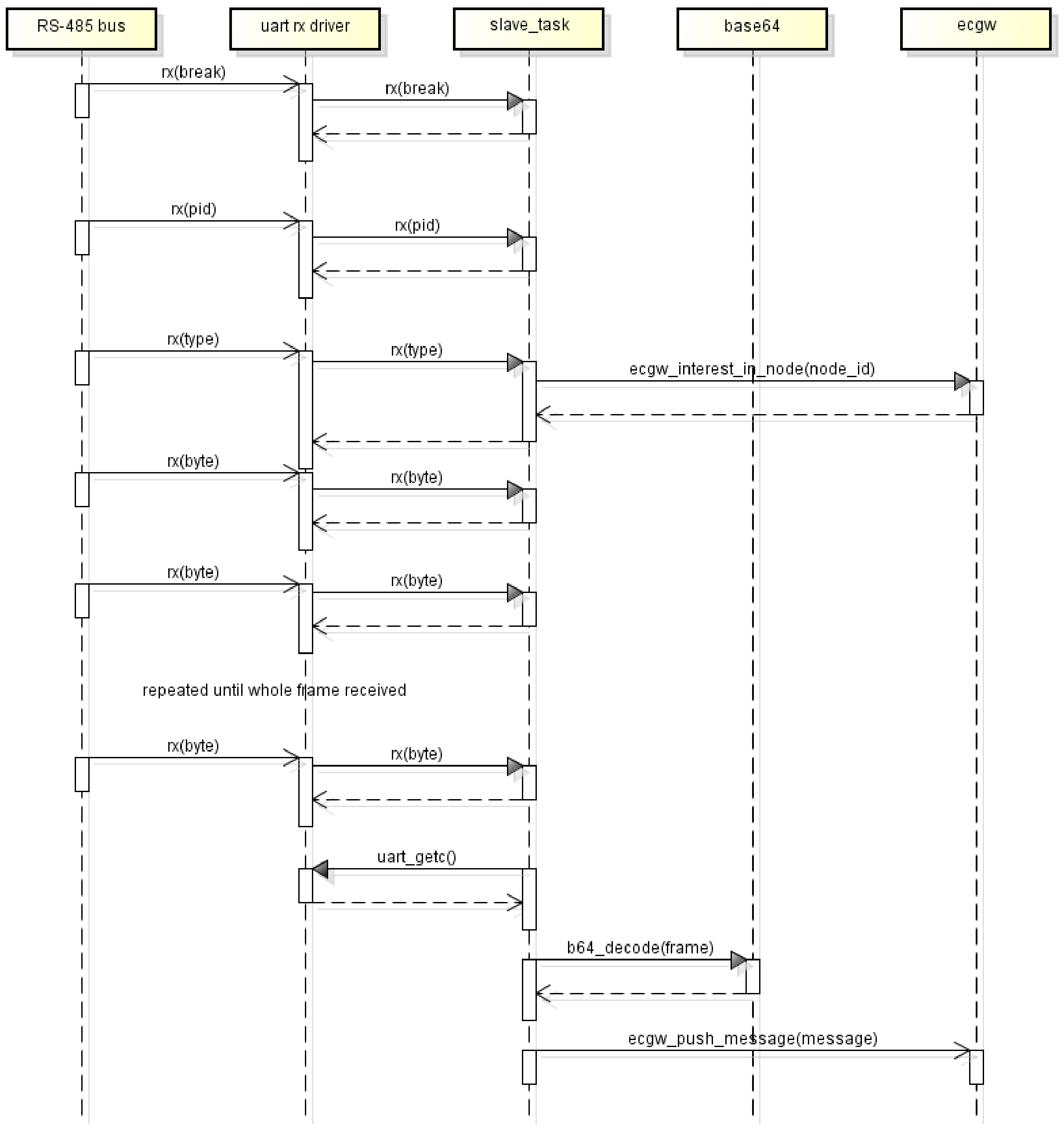


Figure 6-32 Subscriber receiving a frame from a publisher

An activity diagram for the full slave task can be found in Appendix C.

6.7 Design Prototype

To verify the overall design a prototype has been implemented in Java. The experience achieved from this prototype, has been incorporated into the design.

The main benefits we extracted from this prototype was the knowledge about which data structures would be needed, and of course it was used to convince our self that the overall design was feasible.

Due to the fact that is was implemented in Java which is an object oriented programing language, we could not use any of the implementation directly in C implementation.

6.8 Real-time Aspects

If the middleware should be used in a real-time system, then it will be necessary to be able to calculate different timing aspects of the middleware. In this section we have tried to set up a few formulas. These formulas is inspired of the formulas found in *Chapter 11 Scheduling Real-Time Systems* in [20].

For a single node system setup the following formula can be used for calculating the response time from a supplier pushes an RTO/Proxy to a given consumer has finished execution.

$$R_i = CP + CC + C_i + \sum_{j \in cra(i)} \left(C_j + CS + CC + \sum_{l \in pu(j)} R_l \right) + \sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH$$

where

R_i is the response time from an RTO is pushed and until a consumer (i) has finished its execution of the push function

CP is the cost (execution time) of the push call made by the supplier.

CC is the cost of the push call made by the Event Service towards the consumer.

C_i is the cost of the consumer (i) itself.

cra(i) is the consumers registered on the same RTO after the consumer (i) is registered.

C_j is the cost for the *j* consumer.

CS is the Event Service cost of switching to the next consumer.

pu(j) is the set of push calls the *j* consumer performs.

R_l is the response time/cost for the *l* push call the *j* consumer executes.

Γ_s is the set of sporatic interrupt service routines that can occur in the application.

T_k is the minimum time between *k* interrupts.

IH is the cost of an interrupt service routine.

In the above formula it is assumed that all interrupt service routines has the same cost. If this is not the case, the last part of the formula must be changed to have a specific *IH* for each interrupt service routine.

Note: The above formula is recursive if a consumer makes push calls. This can in worst-case lead to infinite response times. It will be necessary to set some restriction, or at least document, how consumers can/must push proxies.

In a distributed system setup the timing aspects of the Event Service Gateway and the COM-Service must be included in the timing aspects. In a given system the Event Service Gateway will be the same independent of the used COM-Service. For the Event Service Gateway we have tried to set up the following formula.

$$R_{ECGW_i} = CFI + CFO + CPM + CSE_i + CCOM + CUSE_i + \sum_{k \in mhp(i)} (CFO + CPM + CSE_k + CCOM + CPUM + CUSE_k)$$

where

R_{ECGW_i} is the response time/cost from an RTO is pushed to the local Event Channel Gateway and until it is pushed into the remote Event Service.

CFI is the cost for a FIFO in operation.

CFO is the cost for a FIFO out operation.

CPM is the cost for a *poll_message* operation on the Event Service Gateway.

CSE_i is the cost for serialisation the (i) message for the COM-Service.

CCOM is the cost the COM-Service to send the message from one node to the other nodes.

CPUM is the cost for a *external_push* operation on the Event Service Gateway.

$CUSE_i$ is the cost for un-serialisation of the (i) message received from the COM-Service.

mhp(i) is the set of higher priority messages in one of the three FIFOs. These messages are messages that are pushed to the Event Channel Gateway's internal consumer and still are in one of the FIFOs.

CSE_k is the cost for serialisation the (k) message for the COM-Service.

$CUSE_k$ is the cost for un-serialisation of the (k)message received from the COM-Service.

We have not tried to setup formulas for the COM-Service cost (*CCOM*), because the formula for the COM-Service will be very dependent of the specific implementation of the COM-Service, where the formulas for the Event Service and the Event Service Gateway are independent of the COM-Service.

The resulting response time from an RTO is pushed on one node, until its consumer is executed on another node will be:

$$R_{D_i} = 2R_i + R_{ECGW_i}$$

where

R_{D_i} is the total response time from a RTO, in a distributed system setup, is pushed on one node to a connected consumer has finished execution on another node.

R_i is the response time from an RTO is pushed and until a consumer (i) has finished its execution of the push function

R_{ECGW_i} is the response time/cost from an RTO is pushed to the local Event Channel Gateway and until it is pushed into the remote Event Service.

7 Implementation

All code written to the Event Service, Event Service Gateway and the main parts of the COM-Service is written in C according to C99 standard without use of any platform specific libraries or functions. This makes it easy to port the middleware to other platforms.

On resource constraint platforms and in safety critical software it is normally not recommended to use dynamic memory allocation (using the heap) [21]. Due to time limitations in this project it is decided to use dynamic memory allocation, but only during initialisation of the Event Service and the Event Channel Gateway. The dynamic memory used are never released, this eliminates the main problems using dynamic memory – defragmentation of the dynamic memory area and misuse of dangling pointers (pointing to memory that has been freed).

7.1 Data structures

7.1.1 FIFO

Send queues (FIFO) in the Event Channel Gateway are needed; thus FIFO data structure has been implemented. In addition to a normal FIFO implementation it has been necessary to implement a few extra functions like the FIFO must be able to tell if a given element is already in the FIFO, and to tell if a given element with a given content is already in the FIFO.

The FIFO is implemented as an Abstract Data Type (ADT)[6].

7.1.2 Linked List

Lists are used in Event Service and Event Channel Gateway; thus a single linked list data structure has been implemented. An iterator for the linked list has also been implemented as part of the linked list data structure.

The linked list is implemented as an Abstract Data Type (ADT)[6].

7.2 Implementation test

A part of the implemented code has been unit tested using CppUTest [22]. We first moved to MinGW and complied to be used on windows.

The FIFO and linked list data structures are tested with 100% code coverage with the result: `OK (12 tests, 12 ran, 102 checks, 0 ignored, 0 filtered out, 0 ms)`. Parts of the COM-Service has also been tested.

Much more unit testing must be performed before the middleware can be released.

The unit test code can be found in the electronic media see Appendix E.

To test the Event Service and the Event Channel Gateway, a COM-Service simulator has been implemented. This simulator is using the Event Channel Gateway interface (IECGW) for communication. The results from running the COM-Service simulator is simple console outputs, which has been manually controlled. Example of this output can be seen in Appendix B. In future these functions test should be integrated in the unit testing suite.

7.3 Code documentation

The source code is documented using Doxygen [23]. The documentation is divided in different sections: Usage, initialization, port to new platform, usage demonstration application etc.

The generated documentation can be found on the electronic media see Appendix E.

[6] By implementing data structures as ADT all details about how the internal data structures in the implementation is completely hided and not accessible from outside the implementation.

7.4 Portability

The middleware can easily be ported to new target platforms as long as they can be programmed in standard C (C99). This section describes the platform dependent implementations that is necessary to do on a platform to be able to use the middleware on the platform.

7.4.1 Periodic Timer

Both the Event Service and the COM-Service need a periodic time signal. A periodic timer functionality must be implemented on the target platform. It must use the following interfaces

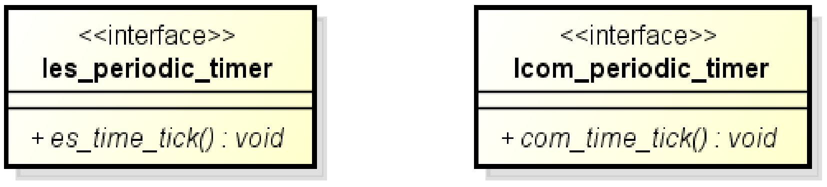


Figure 7-1 Periodic timer interface implemented by Event Service and COM-Service

that are implemented by the Event-Service and the COM-Service, and call the two *time_tick()* functions. How often these functions must be called depends on the application and the COM-Service implementation.

The interfaces are defined in the header file *es_timer_interface.h*, and *com_timer_interface.h* where more detailed documentation can be found.

7.4.2 Communication Service

The middleware needs a communication service to establish communication between the different nodes in the system. This communication service must typically be implemented to match the target platform. In 6.6 we have described an example of a communication service using a modified LIN protocol to communicate between nodes.

A communication service must use the following interface that the Event Channel Gateway implements.

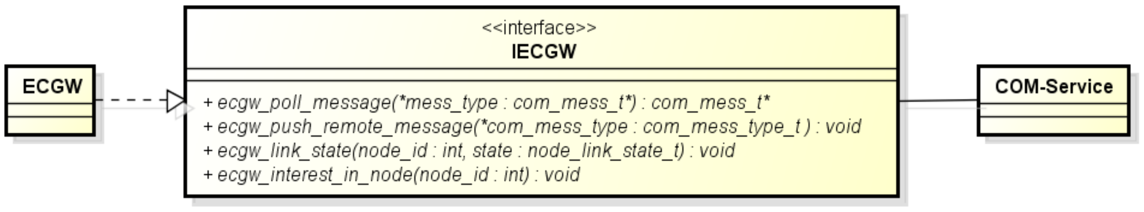

ECGW
<<interface>>
IECGW
+ ecgw_poll_message(*mess_type : com_mess_t*) : com_mess_t*
+ ecgw_push_remote_message(*com_mess_type : com_mess_type_t) : void
+ ecgw_link_state(node_id : int, state : node_link_state_t) : void
+ ecgw_interest_in_node(node_id : int) : void
COM-Service

Figure 7-2 Event Channel Gateway Interface the COM-Service must use

This interface is defined in the header file *ecgw_interface.h*, where more detailed documentation can be found.

7.4.2.1 ecgw_poll_message(…)

The COM-Service must call this function each time it will check if there is messages there has to be sent out on the network that connects the nodes in the system.

The message type tells the kind of message to be sent. In our implementation of the LIN based COM-Service we have found that it can be useful for optimisation purposes in the COM-Service to know the type of messages. The message type can be *none*, *push* and *service*. *None* means that no messages are ready to be send, *push* means that the message contains and event to a RTO, *service* means that it is a message used to establish, destroy or maintain an Event Channel.

Note: All messages must be broadcasted to all nodes that has the middleware installed.

7.4.2.2 ecgw_push_message(…)

The COM-Service must call this function when it has received a message from another node in the system.

For optimisation purpose the COM-Service can use the *ecgw_interested_in_node(…)* function (see 7.4.2.4) to ask if the Event Channel Gateway is interested in push messages from a given node. If not it can save the call to *ecgw_push_message(…)*.

Note: Messages of type service must always be pushed to the Event Channel Gateway.

7.4.2.3 ecgw_link_state(…)

When the COM-Service observes changes in the connection/link state to a node, it must notify the Event Channel Gateway of the change by calling this function.

The states a link can be in are *connected* or *disconnected*.

7.4.2.4 ecgw_interest_in_node(…)

For optimisation purpose the COM-Service can use the *ecgw_interested_in_node(…)* function (see 7.4.2.4) to ask if the Event Channel Gateway is interested in push messages from a given node. If not it can save the call to *ecgw_push_message(…)*.

Note: Messages of type service must always be pushed to the Event Channel Gateway.

7.4.3 Protection of Critical Section

Some parts of the Event Service and the Event Channel Gateway implementation need to be executed atomically. It will be different from platform to platform how this protection must be done.

Two macros must be defined for the target platform:

ES_INIT_CRITICAL_SECTION, ES_ENTER_CRITICAL_SECTION and

ES_LEAVE_CRITICAL_SECTION

Our prototype platform has an ATMEGA64 MCU and on that CPU, we need to disable interrupts to enter a critical section, and enable it again when leaving a critical section. In this case the macros are defined like this:

```
#define ES_INIT_CRITICAL_SECTION        \
   uint8_t _sreg;

#define ES_ENTER_CRITICAL_SECTION       \
   _sreg = SREG;                        \
   cli();

#define ES_LEAVE_CRITICAL_SECTION       \
   SREG = _sreg;
```

7.5 Event Service API

The API for the Event Service has turned out to be very simple to use. There is only a short list of functions the application programmer needs to know, and only a few things that needs to be configured to be able to use the Event Service.

Detailed documentation can be found as Doxygen documentation.

7.5.1 Configuration

To build an application using the API, two configuration files need to be customized.

- *com_config.h* (for the Communication Service) - only needed in a distributed setup

- *event_service_config.h* (for the Event Channel)

7.5.1.1 com_config.h

Both Master and Slave task needs configuration. A macro definition (*MASTER*) in the file decides if the node will be compiled as a Master or a Slave.

Configurations only needed for the Master are (names in parentheses refer to definitions in *com_config.h*):

- Time base (*T_BASE*), to define the time_tick that triggers header transmission
- Timeout (*CS_NODE_TIMEOUT*). If a Slave node does not respond, when addressed, the next Slave is addressed after the timeout
- Number of nodes in the cluster (*NUMBER_OF_NODES*).
- Schedule (*cs_schedule[]*); the order of polling the Slave nodes

The Slave node needs to know Maximum Segment Size (*DATA_FIELD_SIZE*) to define the payload size in a frame.

Common for Master and Slave are the Baud rate configuration (*BAUDRATE*).

7.5.1.2 event_service_config.h

The Event Service itself does not have a master or slave, but is the same on both slaves and masters and therefor needs 100% the same configuration.

(names in parentheses refer to definitions in *com_config.h*):

System setup (*ES_DESTRIBUTED*) must be defined if the system is distributed. In a standalone system the Event Channel Gateway code will not be included in the middleware.

A *typedef* of a *struct* (*rto_id_t*) with all the RTO ids in the system. The last element in this *struct* **must** always be (*NO_OF_RTOS*). The *rto_id_t* is system global.

A *typedef* of a *struct* (*supplier_id_t*) with all supplier ids in the system. Two of the elements in this *struct* **must** always be (*ECGW_SUPPLIER* and *PROXY_NO_SUPPLIER*). The *supplier_id_t* is system global.

If the system setup is distributed (*ES_DESTRIBUTED* is defined) then a *typedef* of *struct* (*node_id_t*) with a node IDs for each node in the system.

7.5.2 Initialisation

The Event Service is initialised by calling a single init function:

```
void es_init_event_service();
```

This will initialise Event Service, Event Channel Gateway and Communication Service.

Note: Before a call to this initialisation function, a system global configuration header file (*event_service_config.h*) must be filled out first. See 0.

7.5.3 Work with Proxies

7.5.3.1 Obtain a Proxy for a RTO

To obtain a proxy from the Event Service the following function must be called:

```
proxy_p es_obtain_proxy(rto_id_t rto_id);
```

It will return a pointer to the proxy for the RTO with the id given as parameter.

7.5.3.2 Connect a Supplier to a Proxy

To connect a supplier to a proxy the following function must be called:

```
proxy_return_code_t proxy_connect_supplier(proxy_p proxy, supplier_id_t s_id);
```

The first parameter specifies the proxy that the supplier should be assigned to, the second parameter specifies the id of the supplier.

Note: In this implementation of the Event Service only one supplier is allowed for a proxy.

7.5.3.3 Disconnect a Supplier from a Proxy

To disconnect a supplier to a proxy the following function must be called:

```
proxy_return_code_t proxy_disconnect_supplier(proxy_p proxy, supplier_id_t s_id);
```

The first parameter specifies the proxy that the supplier should be disconnected from, the second parameter specifies the id of the supplier.

7.5.3.4 Connect a Consumer to a Proxy

To connect a consumer to a proxy the following function must be called:

```
proxy_return_code_t proxy_connect_consumer(proxy_p proxy, consumer_push c_push);
```

The first parameter specifies the proxy that the consumer should be assigned to, the second parameter is a function pointer to the consumer's push function.

The signature for a consumer's push method must be:

```
void name_of_push_function(rto_id_t rto_id, rto_event_t event)
```

The consumer's push functions will automatically be called when a new event is pushed to the Proxy, or the status of the Proxy/RTO change.

The *rto_event_t* that an event can have is defined like this:

```
typedef enum {
   RTO_EVENT_UPDATED, /**< Value has been updated */
   RTO_EVENT_STATUS_CHANGED, /**< Status has been updated */
   RTO_EVENT_META_CHANGED, /**< Meta data has been updated */
   RTO_EVENT_NONE /**< Nothing has happened yet  */
} rto_event_t;
```

Note: In this implementation of the Event Service multiple consumers is allowed for a proxy.

7.5.3.5 Disconnect a Consumer from a Proxy

To disconnect a consumer from a proxy the following function must be called:

```
proxy_return_code_t proxy_disconnect_consumer(proxy_p proxy, consumer_push c_push);
```

The first parameter specifies the proxy that the consumer should be disconnected from, the second parameter is a function pointer to the consumer's push function.

7.5.3.6 Push a value to a Proxy

To get a pointer to the value of a Proxy the application must first get access to it by calling this function:

```
void *proxy_get_value(proxy_p proxy);
```

The length of the value is available in the rto_def[] array see 7.5.3.9.

When an RTOs value has been changed, the application must tell the Event Service that it is ready to be pushed by calling this function:

```
proxy_return_code_t proxy_push(proxy_p proxy);
```

7.5.3.7 Get the RTO ID from a Proxy

The ID of an RTO can be obtained by calling this function:

```
rto_id_t proxy_get_rto_id(proxy_p proxy);
```

The parameter specifies for which proxy the RTO id is wanted.

7.5.3.8 SetGet Current Status of RTO from a Proxy

To get the status of an RTO the following function must be called:

```
status_type_t proxy_get_status(proxy_p proxy);
```

The parameter specifies for which proxy the RTO status is wanted.

The status returned has the type *status_type_t.*

```
typedef enum {
   PROXY_STATUS_OK, /**< Everything is OK. value is valid */
   PROXY_STATUS_NOT_UPDATED, /**< Has not been updated, and the value is NOT valid */
   PROXY_STATUS_TIMED_OUT, /**< Has not been updated in time, and the value is NOT valid */
   PROXY_STATUS_NO_SUPPLIER, /**< Has no supplier, and the value is NOT valid */
   PROXY_STATUS_INVALID /**< The value is NOT valid */
} status_type_t;
```

7.5.3.9 Get Meta data of RTO from Proxy

To get the meta data of an RTO the following function must be called:

```
void *proxy_get_meta_data(proxy_p proxy);
```

The parameter specifies for which proxy the RTO status is wanted.

In this version of the implementation the meta data is a *uint16_t* containing the update period in ticks.

7.5.3.10 Set Update Period for an RTO

To set the update period for an RTO the following function must be called:

```
void proxy_set_update_periode(proxy_p proxy, uint16_t timer_ticks);
```

The parameter specifies for which proxy the RTO update period should be set.

The second parameter specifies the maximum allowed period between the RTO must be pushed.

Note: If the update period is set to zero, then the time out function is disabled – this is the default value.

For more documentation of the API see the Doxygen documentation on the electronic media (see Appendix E).

7.5.4 Example of usage

A simple example of an *event_service_config.h* configuration file for a system with two RTOs (*MOTOR_1_RTO*, *RTO2*) one supplier (*SUPPLIER1_ID*) and two nodes (*NODE1*, *NODE2*) can be seen here:

```
#ifndef EVENT_SERVICE_CONFIG_H_
#define EVENT_SERVICE_CONFIG_H_

#include <stdint.h>

/**
 @ingroup event_service_config
 @brief Are the system distributed?.

 Must be defined in a distributed system.
 */
#define ES_DESTRIBUTED

/**
 @ingroup event_service_config
 @brief System global definition of RTO id's.

 @note the sequence of the RTO id's in this list must be exactly the same as the sequence
of definitions in the rto_def[] array!!
 */
typedef enum {
   MOTOR_1_RTO, /**< RTO ID for the motor data in the system.*/
   RTO2, /**< ID for the second RTO in the system.-replace with your own name and comments*/
   NO_OF_RTOS
} rto_id_t;

/**
 @ingroup event_service_config
 @brief Possible priorities of a RTO.

 Must be used to setup the rto_def array.

 @see rto_def[].
 */
typedef enum {
   RTO_PRIORITY_HIGH,
   RTO_PRIORITY_LOW,
} rto_priority_t;

/**
 @ingroup event_service_config
 @brief Data type to use to define rto_def array.

 Example of a rto_def array:
 @code
 rto_def_t rto_def[] = { { sizeof(rto1_t), RTO_PRIORITY_LOW }, { sizeof(rto2_t), RTO_PRIOR-
ITY_HIGH } };
 @endcode
 Where rto1_t is a typedef of the structure holding the value for the RTO with the ID de-
clared as the first in the rto_id_t etc..

 @note The rto_def array will typically be defined where all the RTO values data types are
defined.
 @see rto_id_t.
 @see rto_def[].
 */
typedef struct {
   uint8_t size_of_value; /**< sizeof the RTO's value - the value is typically a struct */
   rto_priority_t priority; /**< priority of the RTO */
} rto_def_t;
```

```
/**
 @ingroup event_service_config
 @brief System global definition of the supplier id's in the system.

 @note If the Event Service Gateway is used, then ECGW_SUPPLIER must be defined as one of
the suppliers!!
 */
typedef enum {
   SUPPLIER1_ID, /**< ID for the a push supplier in the system. - replace with your own com-
ments. */
   ECGW_SUPPLIER /**< If the Event Service Gateway is used, then this must be defined as one
of the suppliers. */
}supplier_id_t;

/**
 @ingroup event_service_config
 @brief System global definition node id's in the system.
 */
typedef enum {
   NODE1, /**< ID first node in the system - replace with your own comments. */
   NODE2 /**< ID for second node in the system - replace with your own comments. */
}node_id_t;

/**
 @ingroup event_service_config
 @brief System global definition of the priority of the RTO's and the size of the RTO's
value struct's.

 Example of a rto_def array:
 @code
 rto_def_t rto_def[] = { { sizeof(rto1_t), RTO_PRIORITY_LOW }, { sizeof(rto2_t), RTO_PRIOR-
ITY_HIGH } };
 @endcode
 Where rto1_t is a typedef of the structure holding the value for the RTO with the ID de-
clared as the first in the rto_id_t etc..

 @note The rto_def array will typically be defined where all the RTO values data types are
defined.
 @see rto_id_t.
 */
extern rto_def_t rto_def[];

#endif /* EVENT_SERVICE_CONFIG_H_ */
```

Besides the configuration in *event_service_config.h* a few extra things needs to be setup in the application code.

In this example *motor_t* and *rto2_t* are the types of the first and the second RTOs value fields.

E.g. the value the proxy for MOTOR_1_RTO should hold could be defined like this:

```
typedef struct {
   uint8_t state;
   int16_t motor_speed;
   uint8_t motor_direction;
   int16_t motor_temperature;
} motor_t;
```

The corresponding declaration of the *rto_def[]* array could then look like this:

```
rto_def_t rto_def[] = {
   { sizeof(motor_t), RTO_PRIORITY_HIGH },
   { sizeof(rto2_t), RTO_PRIORITY_LOW }
};
```

In this example *motor_t* and *rto2_t* are the types of the first and the second RTOs value fields. The first RTO (*MOTOR_1_RTO*) has in this example been given a high priority (*RTO_PRIORITY_HIGH*). The priority field tells the Event Service which RTOs that must be communicated first to the COM-Service in a distributed system.

With these configurations the Event Service is ready to be started. The commented application program to the above configuration could look like this:

```
/*! @file main_ex.c
 @brief Simple example application.

 @author IHA

 @defgroup usage_ex Simple Event Service Usage Example
 @{
 @brief Demonstrations of the different calls to the Event Service.

 This demonstration application shows how the Event Service can be used.
 There is no different in the usage for standalone- and distributed system setups.
 @}

  @defgroup usage_ex_configuration Simple Event Service Usage Configuration Example
  @{
     @brief Configuration for the Simple Event Service Usage Example.
  }
 */

#include "../Event_service_middleware/include/event_service.h"
#include "../COM-Service/slave/com_service.h"
#include "platform/timer.h"

// The definition of the value for the MOTOR_1_RTO real-time object - this is of course up
to the application how this is defined, it can also just be a simple variable.
// NOTE: pointers is not allowed!!
typedef struct {
   uint8_t state;
   int16_t motor_speed;
   uint8_t motor_direction;
   int16_t motor_temperature;
} motor_t;

// The definition of the value for the RTO1 real-time object - this is of course up to the
application how this is defined, it can also just be a simple variable.
// NOTE: pointers is not allowed!!
typedef struct {
   uint8_t just_a_value;
} rto1_t;

proxy_p proxy_motor_rto; // A variable that can hold the proxy reference to MO-TOR_1_RTOs
proxy – here defined global, but it a reel application it will typical be declared locally
where it should be used
proxy_p proxy_rto1; // A variable that can hold the proxy reference to RTO1s proxy

// Needed definition - the sequence must be the same as in the definition of rto_id_t in
event_service_config.h
rto_def_t rto_def[] = { { sizeof(motor_t), RTO_PRIORITY_HIGH }, { sizeof(rto1_t), RTO_PRI-
ORITY_LOW }};

/********************************************//**
@ingroup usage_ex
@brief Example of consumer.

The same consumer function can be used/registered for multiple consumers.

@param[in] rto_id of the of the Proxy the event is coming from.
```

```
@param[in] rto_event the event that has been pushed to the Proxy/RTO.
***********************************************/
void motor_1_consumer(rto_id_t rto_id, rto_event_t rto_event) {
   switch (rto_event) {
   case RTO_EVENT_UPDATED:
     {
      // Use the value to something useful
      // The new value can be accessed like this
      motor_t *_motor = (motor_t *)proxy_get_value(proxy_motor_rto);

      uint8_t _new_direction = _motor->motor_direction;
      uint16_t _new_speed = _motor->motor_speed;
      uint16_t _new_temperature = _motor->motor_temperature;
      uint8_t _new_state = _motor->state;
      break;
     }

   case RTO_EVENT_STATUS_CHANGED:
     {
      // Ask the proxy about the new status
      status_type_t _new_status = proxy_get_status(proxy_motor_rto);
      switch(_new_status) {
         case PROXY_STATUS_TIMED_OUT: // The RTO is not updated in due time
           {
            // Do what is needed
            break;
           }

         case PROXY_STATUS_NO_SUPPLIER: // We have lost the supplier
           {
            // Do what is needed
            break;
           }

         case PROXY_STATUS_INVALID: // A supplier has invalidated the RTO
           {
            // Do what is needed
            break;
           }

         default:
           {
            break;
           }
        }
      break;
     }

   case RTO_EVENT_META_CHANGED: // This is the status when the meta data is changed - in the
current version it is the update_periode that has been changed by the supplier
     {
      // This is how to get the new update period if the app should be interested
      uint16_t the_new_update_period = proxy_get_update_period(proxy_motor_rto);
      break;
     }

   default:
     {
      break;
     }
   }
}

int main(void)
{
   es_init_event_service(); // Initialize the Event Service
   sei(); // Enable global interrupt

   // Example: How to a connect a consumer to the MOTOR_1_RTO real-time object
   proxy_motor_rto = es_obtain_proxy(MOTOR_1_RTO); // First obtain the proxy from the event
service
   proxy_connect_consumer(proxy_motor_rto, &motor_1_consumer); // Connect the local consumer
to the proxy

   //Example: How to connect a supplier to the MOTOR_1_RTO real-time object
   proxy_motor_rto = es_obtain_proxy(MOTOR_1_RTO); // First obtain the proxy from the event
service - actually not needed here, we have it already from above
   proxy_connect_supplier(proxy_motor_rto, SUPPLIER1_ID); // Connect the local supplier to
the proxy

   //Example: How to set the update period for the MOTOR_1_RTO real-time object
   proxy_motor_rto = es_obtain_proxy(MOTOR_1_RTO); // First obtain the proxy from the event
service - actually not needed here, we have it already from above
   proxy_set_update_periode(MOTOR_1_RTO, 120); // Here it is set to 120 Event Service timer
ticks

   //Example: How to set the push a new value the MOTOR_1_RTO real-time object
   proxy_motor_rto = es_obtain_proxy(MOTOR_1_RTO); // First obtain the proxy from the event
service - actually not needed here, we have it already from above
   // Use the value to something useful
   // The value can be accessed like this
   motor_t *_motor = (motor_t *)proxy_get_value(proxy_motor_rto);
   _motor->motor_speed = 1035; // Just set the new speed
   proxy_push(proxy_motor_rto); // Push the event when ready

   //Example: How to invalidate the MOTOR_1_RTOs real-time object
   proxy_motor_rto = es_obtain_proxy(MOTOR_1_RTO); // First obtain the proxy from the event
service - actually not needed here, we have it already from above
   proxy_set_status(proxy_motor_rto,PROXY_STATUS_INVALID);

   //Example: How to disconnect a consumer from the MOTOR_1_RTO real-time object
   proxy_motor_rto = es_obtain_proxy(MOTOR_1_RTO); // First obtain the proxy from the event
service - actually not needed here, we have it already from above
   proxy_disconnect_consumer(proxy_motor_rto, &motor_1_consumer);

   //Example: How to disconnect a supplier from the MOTOR_1_RTO real-time object
   proxy_motor_rto = es_obtain_proxy(MOTOR_1_RTO); // First obtain the proxy from the event
service - actually not needed here, we have it already from above
   proxy_disconnect_supplier(proxy_motor_rto, SUPPLIER1_ID);

   for (;;)
   {

   }
}
```

In the above example app it can be seen how the different function calls to Event Service are done. The example application will **not** work at all, and is only used as an illustration of the usage of the different Event Service functions.

Note: The application code will be exactly the same in a distributed setup and in a standalone system setup, only the configuration will differ.

8 Results – Evaluation

Evaluation of the middleware and COM-Service is done by a range of experiments, as illustrated in Figure 8-1. Common for all experiments are a set of evaluation points characterized as Quantitative and Qualitative evaluation, see 8.1 and 8.2. The experiments are designed to give some answers on the amount of resources needed, response time, latency, how well the system scales and how easy it is to make changes to the application. Unfortunately, we do not have access to the original source code for the hoist system, so a comparison is not directly possible.

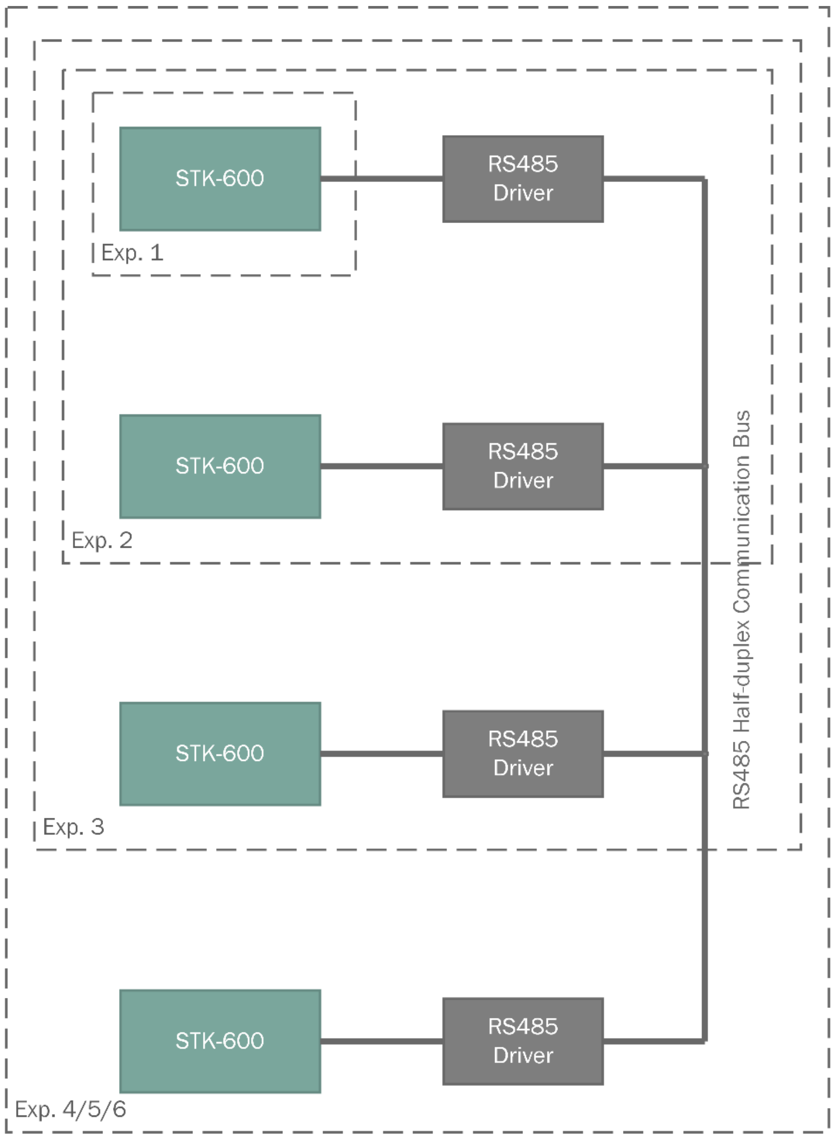


Figure 8-1 Test setup used for quantitative evaluation

We configure the test application to supply RTOs at a regular interval to update the status of LEDs on the evaluation boards. We measure the timing using a digital oscilloscope oscilloscope (PicoScope 3204A[7]).

As hardware platform, we connect four ATMEL STK600 Evaluation Boards[8] to the communication bus via RS-485 line drivers. The Evaluation Boards have a standard UART for communication and 8 LEDs to display status of RTOs.

[7] https://www.picotech.com/
[8] http://www.atmel.com/tools/stk600.aspx

The RS-485 line drivers has been designed and build as part of this project. The schematics for the line drivers can be found in Appendix D.

8.1 Quantitative Evaluation

Measurements on the test application configured to four nodes each with four RTOs has shown the following results:

- The Memory footprint of the standalone Event Service middleware:
  - Program memory: 2058 bytes
  - Static RAM usage of Event Service is 130 bytes
  - RAM usage for each RTO is 60 bytes plus the value size of the RTO plus 6 bytes extra for each possible consumer.

- Memory footprint of a distributed Event Service, incl. Event Channel Gateway and communication service:
  - Program memory: 10.184byte
  - Static RAM usage: 585 bytes
  - RAM usage for each RTO is 60 bytes plus the value size of the RTO plus 6 bytes extra for each possible consumer.

8.1.1 Source Code

The number of effective source code and comments lines is counted with the CLOC (Count Lines Of Code) tool [24].

The results are:

Module	Effective Code Lines	Comment Lines	Header File Lines
Event Service	484	536	129
Event Channel Gateway	463	243	83
COM-Service	933	595	120
Total	1880	967	332

The number of code lines are is in a range where we expects it to be maintainable.

8.1.2 Experimental Tests

We have conducted a number of experiments on the four different test setups shown in Figure 8-1. Table 4 shows the minimum and the maximum delay from a supplier is updating an RTO until the different consumers receives the update/event. The minimum delay is similar to the latency in the middleware, and the difference between the minimum and the maximum delay is similar to the jitter on the latency. Each of the experiments is done with a single RTO (RTO1), two RTO's (RTO1, RTO2) and with four RTO's (RTO1, RTO2, RTO3, RTO4). The test application's supplier will update the RTO's simultaneously and at the same time it sends a pulse out on an external port pin. When the consumers receives the update/event the consumer will send a pulse out on an external port pin, this gives the possibility to measure the delay times between supplier events and consumer events.

Experiment 1 uses a single node where Event Service Gateway and COM-Service are not used.

Table 4 Response Times for Event Service on a single node. All values in µs

	Master	Supplier		Consumer 1		Consumer 2		Consumer 3		Consumer 4	
				Node 1		Node 1		Node 1		Node 1	
	Node	Node	RTO	min	max	min	max	min	max	min	max
Exp. 1a	Node 1	Node 1	RTO1	8,8	8,8						
	Node 1	Node 1	RTO1	13	13	8,8	8,8				
	Node 1	Node 1	RTO1	17,2	17,2	13	13	8,8	8,8		
	Node 1	Node 1	RTO1	21,4	21,4	17,2	17,2	13	13	8,8	8,8
Exp. 1b	Node 1	Node 1	RTO1	8,8	8,8						
	Node 1	Node 1	RTO2	8,8	8,8						
	Node 1	Node 1	RTO1	13	13	8,8	8,8				
	Node 1	Node 1	RTO2	13	13	8,8	8,8				
	Node 1	Node 1	RTO1	17,2	17,2	13	13	8,8	8,8		
	Node 1	Node 1	RTO2	17,2	17,2	13,3	13	8,8	8,8		
	Node 1	Node 1	RTO1	21,4	21,4	17,2	17,2	13	13	8,8	8,8
	Node 1	Node 1	RTO2	21,4	21,4	17,2	17,2	13	13	8,8	8,8
Exp. 1c	Node 1	Node 1	RTO1	8,8	8,8						
	Node 1	Node 1	RTO2	8,8	8,8						
	Node 1	Node 1	RTO3	8,8	8,8						
	Node 1	Node 1	RTO1	13	13	8,8	8,8				
	Node 1	Node 1	RTO2	13	13	8,8	8,8				
	Node 1	Node 1	RTO3	13	13	8,8	8,8				
	Node 1	Node 1	RTO1	17,2	17,2	13	13	8,8	8,8		
	Node 1	Node 1	RTO2	17,2	17,2	13	13	8,8	8,8		
	Node 1	Node 1	RTO3	17,2	17,2	13	13	8,8	8,8		
	Node 1	Node 1	RTO1	21,4	21,4	17,2	17,2	13	13	8,8	8,8
	Node 1	Node 1	RTO2	21,4	21,4	17,2	17,2	13	13	8,8	8,8
	Node 1	Node 1	RTO3	21,4	21,4	17,2	17,2	13	13	8,8	8,8
Exp. 1d	Node 1	Node 1	RTO1	8,8	8,8						
	Node 1	Node 1	RTO2	8,8	8,8						
	Node 1	Node 1	RTO3	8,8	8,8						
	Node 1	Node 1	RTO4	8,8	8,8						
	Node 1	Node 1	RTO1	13	13	8,8	8,8				
	Node 1	Node 1	RTO2	13	13	8,8	8,8				
	Node 1	Node 1	RTO3	13	13	8,8	8,8				
	Node 1	Node 1	RTO4	13	13	8,8	8,8				
	Node 1	Node 1	RTO1	17,2	17,2	13	13	8,8	8,8		
	Node 1	Node 1	RTO2	17,2	17,2	13	13	8,8	8,8		
	Node 1	Node 1	RTO3	17,2	17,2	13	13	8,8	8,8		
	Node 1	Node 1	RTO4	17,2	17,2	13	13	8,8	8,8		
	Node 1	Node 1	RTO1	21,4	21,4	17,2	17,2	13	13	8,8	8,8
	Node 1	Node 1	RTO2	21,4	21,4	17,2	17,2	13	13	8,8	8,8
	Node 1	Node 1	RTO3	21,4	21,4	17,2	17,2	13	13	8,8	8,8
	Node 1	Node 1	RTO4	21,4	21,4	17,2	17,2	13	13	8,8	8,8

Table 4 shows the measured response times for Experiment 1.

Exp. 1a shows how the Event Service scales when a single RTO (RTO1/Proxy1) have one to four consumers see also Figure 8-2 and Figure 8-3. The result shows that it scales linear – this is not surprisingly thus the consumers are stored internally in a linked list, and is called in turn.

Exp. 1b to 1d shows how it scales when the number of RTOs are increased from one to four see also Figure 8-4 and Figure 8-5. The result shows that all RTO’s has the same response time, measured from a push to the consumer is executed, independent of the number of RTOs. Again no surprise because without an operating system or interrupt service routines running only one RTO can be pushed at the same time.

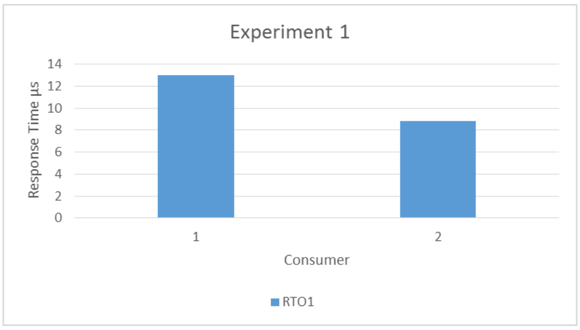


Figure 8-2 One RTO with two consumers

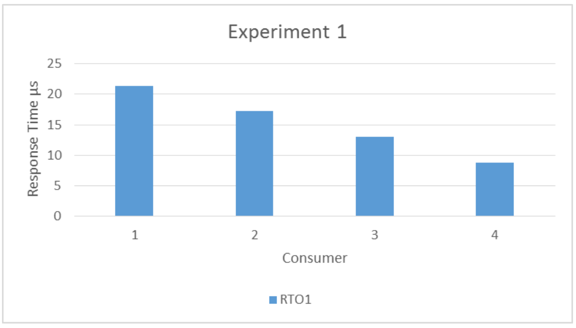


Figure 8-3 One RTO with four consumers

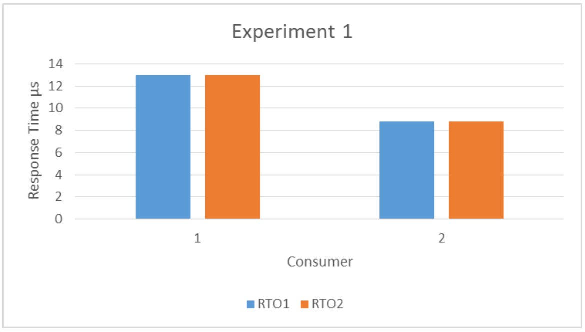


Figure 8-4 Two RTOs with two consumers each

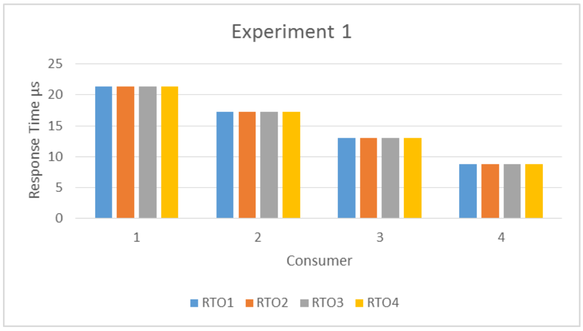


Figure 8-5 Four RTOs with four consumers each

In Experiment 2 the setup is four nodes where Node1 is both Master in COM-Service and supplier of four RTOs. All four nodes consumes these RTOs.

Table 5 Response Times for a distributed system with four node. All values in µs

	Master	Supplier		Node 1		Node 2		Node 3		Node 4	
	Node	Node	RTO	min	max	min	max	min	max	min	max
Exp. 2	Node 1	Node 1	RTO1/Consumer 1	92,9	104,5	1550	1920	1550	1920	1550	1920
	Node 1	Node 1	RTO2/Consumer 2	105,8	106	4470	4844	4470	4844	4470	4844
	Node 1	Node 1	RTO3/Consumer 3	118,6	119,2	7373	7742	7373	7742	7373	7742
	Node 1	Node 1	RTO4/Consumer 4	252,4	274,5	10430	10770	10430	10770	10430	10770

The results shows how Event Channel gateway and COM-Service introduce delays and jitter to response times. It can be seen that the consumers on Node 1 (Master and Supplier) have less response times than the consumers on the other nodes.

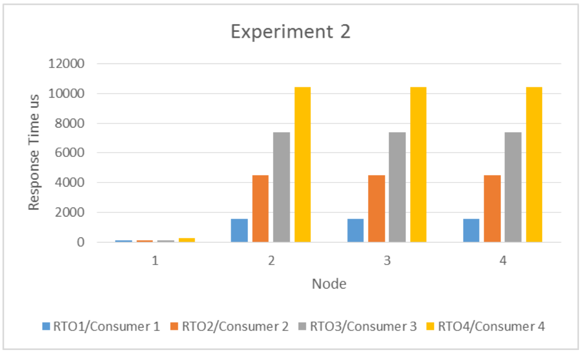


Figure 8-6 Two RTOs with two consumers each

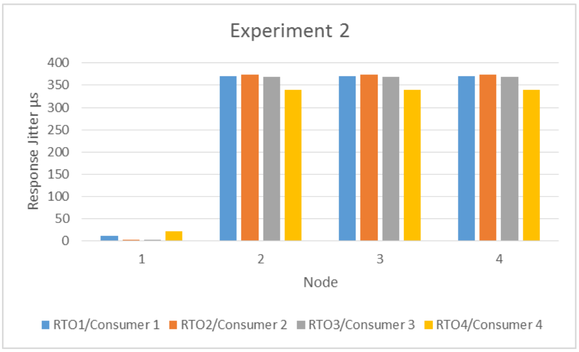


Figure 8-7 Four RTOs with four consumers each

In Figure 8-6 the COM-Service's round robin polling of the nodes can be seen. The master node (Node 1) will send one RTO push message each time it is polled. The response time for consumers of specific RTO is the same independent of which node the consumer resides. It will be like this because the COM-Service broadcast the push messages and all nodes will receive it at the same time. The middleware scales linear with number of nodes. The jitter in Figure 8-7 are higher on the “remote” nodes. This is due to the fact that the polling sequence of nodes is not synchrony to the push messages done by the suppliers in Node 1.

In Experiment 3 the setup is four nodes where Node1 is both Master in COM-Service and supplier of four RTOs. All four nodes consumes these RTOs. RTO4 is configured to have high priority.

Table 6 Response Times for a distributed system with four node, where RTO4 has high priority. All values in µs

	Master Node	Supplier Node	RTO	Node 1		Node 2		Node 3		Node 4	
				min	max	min	max	min	max	min	max
Exp. 3	Node 1	Node 1	RTO1/Consumer 1	92,6	93,4	3267	4502	3267	4502	3267	4502
	Node 1	Node 1	RTO2/Consumer 2	88,4	105,9	4807	6416	4807	6416	4807	6416
	Node 1	Node 1	RTO3/Consumer 3	102,4	118,4	7770	8293	7770	8293	7770	8293
	Node 1	Node 1	RTO4 (H)/Consumer 4	81,3	81,7	952	2120	952	2120	952	2120

The results shows how RTO4 is distributed first on the communication bus Figure 8-8. It is not clear why the jitter on the response times are lower for RTO3 compared with the jitter that was observed in Experiment 2.

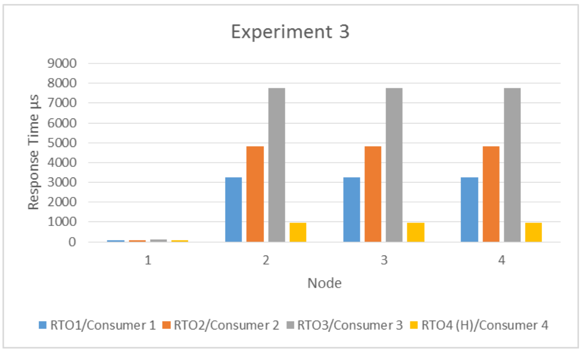


Figure 8-8 Two RTOs with two consumers each

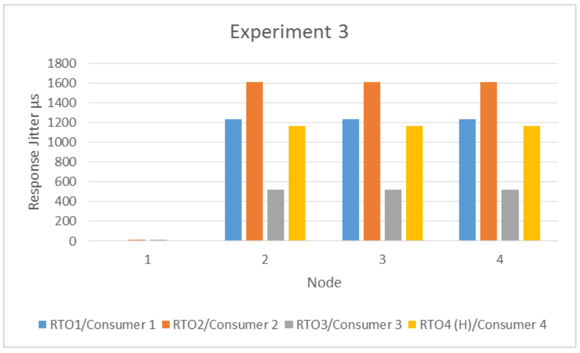


Figure 8-9 Four RTOs with four consumers each

In Experiment 4 the setup is four nodes where Node1 is Master in COM-Service and Node 2 is supplier of the four RTOs. All four nodes consumes these RTOs. RTO4 is configured to have high priority.

Table 7 Response Times for a distributed system with four node, where Node 2 have all the suppliers. All values in µs

	Master Node	Supplier Node	RTO	Node 1		Node 2		Node 3		Node 4	
				min	max	min	max	min	max	min	max
Exp. 4	Node 1	Node 2	RTO1/Consumer 1	1104	2095	92,1	93,1	1104	2095	1104	2095
	Node 1	Node 2	RTO2/Consumer 2	3320	4340	105,4	105,9	3320	4340	3320	4340
	Node 1	Node 2	RTO3/Consumer 3	5135	6378	110,9	118,1	5135	6378	5135	6378
	Node 1	Node 2	RTO4/Consumer 4	7236	8611	107,6	131	7236	8611	7236	8611

The results is comparable with the results from Experiment 2 except that it is now Node 2, the node with the suppliers, which has the lowest response times. It must be like this because the COM-Service is not involved in pushing consumers on the node that has the suppliers.

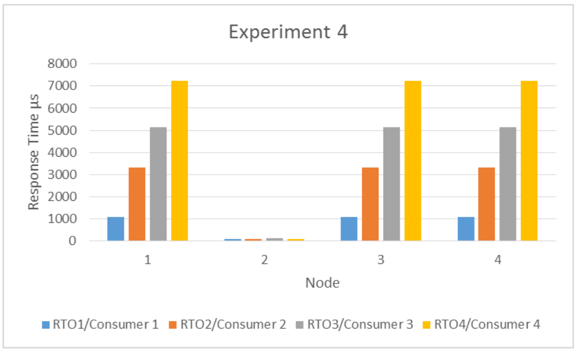


Figure 8-10 Two RTOs with two consumers each

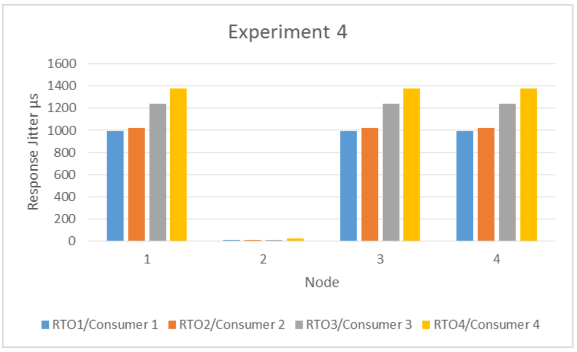


Figure 8-11 Four RTOs with four consumers each

In all experiments Node 1 will be master in the bus-communication the COM-Service performs. The COM-Service has a baud rate of 230.4 kb/s, and the MCUs are running at 16 MHz clock frequency.

8.2 Qualitative Evaluation

We evaluate the quality of the middleware by discussing ease-of-use, portability and generality.

8.2.1 Ease-of-use

Section 7.5.4 gives an example on how to use the API. This example illustrates the low coupling between the application and API which makes it useful for a programmer with only little experience in implementation of distributed systems. This contributes to a shorter development time for a distributed embedded system.

Many, special new, programmers find it difficult to obtain low coupling in an application. The subscriber/consumer design used in the Event Service will automatically help programmers to get a lower coupling between individual application parts.

As a QoS feature, a link state information is monitored for all nodes. This is used by the Event Channel Gateway to clean up subscribers and consumers connected to the defective node (see 6.4, optimization step 5).

If the node comes up again, the consumers and subscribers are reestablished automatically. This contributes to, that the system goes into normal operation again after a temporary error, e.g. a watchdog reset on one of the nodes.

The communication protocol is very robust, in the way that slave tasks will always get back on track if a temporary event corrupts one or more bytes during transmission. The message might get lost but the slave task will get synchronized to the frame start by the next *Break* character, and the next frame will be transferred as expected. At this point of time no automatic retransmissions are sent, so there is no guarantee for delivery. This problem could cause a missed sample and lead to the event: PROXY_STATUS_TIMED_OUT (see the example application).

8.2.2 Portability

The low coupling between the Event Service and Communication Service, makes it possible to port the middleware to another hardware platform and/or communication technology by changing very little in the middleware.

- New macros to disable and enable interrupts, in *app/platform/platform_port.h*, are needed for the target, as they are used by the Event Service and Event Channel Gateway.

- The timer used by the Communication Service and Event Service needs to be implemented in the new platform. An example is found in *app/platform/timer.c*
- The UART driver must be adapted if another MCU type is used. All hardware dependent communication functions are found in *COM-Service/uart/uart.c*
- The four interface functions needs to be supported by the communication service, if a different is implemented. It must include the *Event_service_middleware/interfaces/ecgw_interface.h* to be able to access the interface functions.

8.2.3 Generality

In the Introduction we stated that a resource constrained ECU has less than 128 kByte Program memory and 16 kByte RAM, and that we aimed for systems with less than 64 kByte Program memory and 4 kByte RAM. This goal has been achieved with the present middleware, as this takes about 10 kByte of Program memory. This leaves plenty of space for more advanced applications.

With this small footprint, it should be possible use it in an implementation of the hoist system described in section 2.1.

The measurements shows that a simple system can be implemented without a scheduling system. The cost is that interrupts, in some circumstances, are blocked for a relative long period. This is unfortunate, e.g. if the application has to sample a sensor at a constant rate.

The reason for the long periods is that the slave task is called from UART receive interrupt only. This means that a node will have to query the Event Channel Gateway, calculate checksum and base64-encode the response frame within the interrupt routine.

Measurements on the execution time of the slave task shows that it takes about 150 µs at 16 MHz CPU clock frequency. This creates a corresponding jitter on other interrupts, as they can be pending for this time.

8.3 Reflections

8.3.1 The Work Done

The long receiver interrupt routine, in the communication service, might seem like a bad design choice. It was, though, a well thought decision. We wanted to try this design as it would make it possible to create a slave node completely without use of timers, but still be quick to respond to the reception of headers on the bus. This would be desirable because it would lower hardware requirements on the system and make the application development simple. The only hardware requirement would then be one UART, an amount of Program memory and Data memory. The rest of the CPU features would be available for the application.

This approach would work fine for some applications that are not time critical, e.g. a weather station that polls some slow changing parameters.

A more general communication service should not interfere so much with other interrupts, so we think that the best solution is to use a timer in the slave nodes also, and make a Cyclic Executive scheduling application. It would not make a big change to the slave task, but it must be split into two. One part that runs periodically to poll for messages from the Event Channel Gateway and create the frame, and a simpler Receiver Interrupt routine that could just pick a ready-packed frame when it was time to deliver the response. This would prevent Interrupts from being blocked for long time.

8.3.2 The Process

From the start of this project we were pointed in the right direction by our supervisor, who gave us some relevant articles to inspire us. This gave us the first ideas of how we could design the middleware and focus our search in the direction of the CORBA Event Service architecture.

The work was later divided into two focus areas; Event Service and Communication Service.

We did not create a time schedule for the project because it was completely unpredictable how much time we would be able to put into it in any given week. The project was done mostly in the spare time alongside our fulltime jobs, which at times demanded a lot of overtime.

To make sure that the project would progress, we scheduled as many workshop days as possible, to meet and work together. This helped to inspire and keep up the motivation through the long project period.

Ib focused on the Event Service and Event Channel Gateway, while Erland focused on the Communication Service.

The interface was decided so software could be developed and tested independently.

It was a great time saver that we were able to compile and run the unit test of most parts of the software on PCs

8.3.2.1 The communication Service by Erland

The physical layer of the communication bus was given in advance as we had set ourselves the goal to make a system that could run on the hardware from the described case. In the search for a suitable link layer protocol I found inspiration in the simple LIN protocol. This seemed to fit well with some of our initial ideas.

Early in the process an activity diagram was created of the slave task. The time spend on this was an extremely good investment as this has been the roadmap throughout implementation, unit test development and execution, and debugging. It turned out, that it contained exactly the right level of details. During implementation it was easy to keep track of how far the implementation had come, even when other work duties required my attention for long periods.

During the development of unit tests, it was a road map to ensure that all corners of the design were tested. When the test was executed – and failed – it was clear why it failed by following the flow on the diagram. Some bugs literally took minutes to fix with that insight. Some of them lead to changes in the design and the diagram had to be updated.

Unfortunately unit tests could not be executed on the hardware dependent UART driver, so there were other bugs that took many hours to find and fix, but it has been a very interesting project and so far I'm pleased with the results.

8.3.2.2 The Event Service and Event Channel Gateway by Ib

The research in related work was very useful as inspiration, when first the CORBA Event Service and the TAO Real-time CORBA implementation was known we were ready to start the design work.

It has been very interesting to first make a basic design of a middleware in which an important criterion has been that it must be able to be implemented and executed on a hardware platform with very limited resources and still be effective. It turned out that parts of the relatively simple interface used in CORBA Lightweight Event Service could be used advantageously in our design. The distributed event service part is mainly inspired by the TAO project. When the basic design was in place, the focus has been on how it could be optimized. Another exciting and educational aspect has been to implement an Event Service and an Event Channel Gateway independent of the platform it should be executed on. At the same time it has been designed and implemented to have a very low coupling to the underlying communication layer (the COM-Service).

The Event Service and Event Channel Gateway is designed to 100% independent on a master node, and all nodes in a distributed system will have exactly the same implementation of these services.

8.3.3 Outcome

8.3.3.1 Personal outcome, by Erland

It has been a while since last time I was involved in development of programs larger than a few hundred lines of code. Resent years the only programs I have written were small exercises for the students I teach, and I'm not teaching programming classes - so I was a little rusty at first.

It has been a great opportunity for me to try out some of the many tools that is now available. For me it was a new experience to use Git for versioning control, but it proved its worth as we were two programmers constantly making changes to the code.

The framework for Unit Testing C-programs was also a first time for me. It saved a lot of time debugging the code and refining the design. I had to replace the hardware dependent UART routines with a stub to run the tests, so there were a part of the code that could not be Unit Tested.

The clear interface between Communication Service and Event Channel Gateway also made it easy to test each part separately.

The big challenge was to debug the hardware dependent UART driver. We had debuggers available, but when it comes to something so dependent on interrupts it becomes tricky. I learned, though, a few tricks about conditional breakpoints that could be of some use. Also our Oscilloscope and Logic Analyzer were heavily used to convince myself (and others) that the communication on the bus works as designed to do.

I had never worked with LIN before but I felt early in the process that this could be a good way to go, since it had very little overhead. We couldn't use it in the standard version, since the transmission rate was much too low and we had another physical layer, but with the changes we made, I feel confident that it was a good decision.

I have also gained a lot of new insight in distributed systems and it has been interesting and inspiring to learn more details about event service systems.

8.3.3.2 Personal outcome, by Ib

A large part of the time we have spent on the project has been used to research what others have been doing in our field, unfortunately it turned out that we not able to find much that targeted resource constrained hardware platforms, but plenty much targeting powerful platforms.

Another large part of time has been spent on making a solid design, which we carefully worked through before we started the implementation. And once again, it has proven to be a great advantage for the entire project implementation.

The scientific approach to a Master project has been a major, but instructive, challenge which we unfortunately have not got sufficient knowledge about during our Master study.

8.4 Future Improvements

There are a couple of improvements that could be implemented in the future. Some are necessary to make our work run reliably in a real application, and some are improvements that would make a better system.

If we had more time, the code could be optimized further to bring down the code memory and RAM usage. The Event Service and the Event Channel Gateway allocates a fixed amount of memory independent of the number consumers that are actually connected to a supplier. This could be improved so memory isn't allocated until is to be used, and in the lowest possible amount.

The error handling in the Communication Service needs to be improved. Retransmissions of failed frames has not yet been implemented. Currently, if the checksum verifications fails, the frame is dropped. This is a problem when RTOs are pushed to the consumers as they cannot request a retransmission. The communication services need to be reliable because the application has no way to verify if a RTO has been delivered as expected.

The current version does not check the data echoed back from the bus when transmitting. When that part is finished, it would make the sending node able to detect bit errors on the bus. This will be an improvement, because the checksum algorithm cannot detect all errors. A two-bit error could slip undetected through the check, if the bits cancel out each other in the summation.

Messages that are longer than one frame are not allowed. A future improvement to the Communication Service could segment long messages and send them in multiple frames.

As discussed under Reflections, a Cyclic Executive approach to the design of the application and middleware/communication services could bring down the long Interrupt times. This would make a more stable design, where the application could use interrupts more freely.

A version that runs under a Real Time Operating System, e.g. FreeRTOS could also be an improvement. This could make the design more elegant when restructured into tasks, and the build-in communication features, like queues could be used to pass messages instead of the many local buffers.

Since the use of a RTOS increases the complexity for the application programmer, it might not always be desirable. If another hardware platform is to be used, the RTOS also needs to be ported, if this hasn't been done already.

A better example of usage could be made and more performance measurements to understand how it would work in a real example. A simplified version of the hoist system would complete the project.

9 Related Work

In this section some related work will be briefly described.

9.1 Robot Operating System (ROS)

ROS implements a Publisher/Subscriber model for distributing messages between nodes. The application subscribes on topics, and does not need to know who publishes it. There can be multiple publishers and multiple subscribers. For the publishers and subscribers to find each other a Master is used.

A node can notify the Master that it wants to advertise messages to a certain topic. Other nodes can then query the Master for subscription to this topic. Once a match is found between a publisher and a subscriber to a topic they can communicate peer-to-peer.

For communication service, ROS uses TCP/IP.

9.2 CORBA

Common Object Request Broker Architecture (CORBA) is a standard by the Object Management Group (OMG) founded in 1989 [25]. The first CORBA standard (1.0) was released in 1991. The latest CORBA standard (3.3) was released in 2012 [26]. The main purpose of CORBA was to merge Remote Procedure Call (RPC) with Object Orientation (OO). The vision was to set a standard for a complete infrastructure for distributed computing in its widest extent – independent of platforms and programming languages.

CORBA is described more thoroughly in section 4.1.

9.3 TAO

"TAO is a freely-available, open-source, standards-compliant, real-time CORBA ORB that provides end-to-end quality of service guarantees to applications by vertically (i.e. network interface ↔ application layer) and horizontally (i.e. end-to-end) integrating CORBA middleware with OS I/O subsystems, communication protocols, and network interfaces. TAO is implemented using the ACE framework, which contains a rich set of high-performance and real-time reusable software components. These components automate common communication tasks such as connection establishment, event de-multiplexing and event handler dispatching, message routing, dynamic configuration of services, and flexible concurrency control for network services." [7]

TAO is described more thoroughly in section 4.2.

9.4 Local Interconnect Network

Local Interconnect Network is a standard by the LIN Consortium. Currently revision 2.2 is recommended for new designs. The first revision, 1.0 was published in 1999 with the purpose to create a common standard for a hierarchical vehicle network. The aim was to reduce cost and enhance quality compared to the many proprietary solutions in the automotive industry, where the higher bandwidth of CAN is not required.

LIN is described more thoroughly in section 4.3.

10 Conclusion

The main purpose of this project is to make it easier for programmers to develop distributed embedded systems that are more maintainable.

Our thesis for the project was that it would be possible to implement high-level object sharing concepts and middleware technologies in resource constrained distributed embedded systems.

We sat the goal to implement an improved version of the hoist system, using the middleware technology that we designed for resource constrained systems.

We didn't quit achieve that goal since time ran out. Instead we implemented a simple test program to demonstrate how RTOs are distributed among the connected nodes.

We feel, though, that the most important part of the goal has been achieved, and that we can answer our thesis with a "Yes, it can be done!", and the proof is the design and implementation of the Event Channel, Event Channel Gateway and Communication Service.

We have shown a design that takes only the most necessary elements from Real Time CORBA, and optimised it to be more efficient when used in resource constrained systems with low communication bandwidth.

The design is lean. The implementation comprises of only 1880 code lines and fits into an ATMEGA64 microcontroller. On each node less than 10 kByte of program memory is used by the middleware. This is about 15% of this MCU's program memory. The RAM usage depends mostly of the number of RTOs and consumers in the system as memory is allocated for queues. The test system use 585 bytes. This is about 14% of the on-board RAM in the ATMEGA64.

The small codebase also makes it easy to maintain, and we believe that the documentation is sufficient for others to understand the design.

We have designed a middleware that has a well-defined API for applications, and shown, with an example, how to use it.

Network communication is always a bottleneck in resource constrained systems, but we believe that the implemented communication services works as efficiently as possible without a dedicated communication MCU. The control of the communication is taken out of the hands of the application programmer and into the control of a master node. This makes the timing more predictable as the messages are send according to a schedule instead of sporadic. We have shown that with a Baud rate of 230.400 bps the network latency is in the order of a few milliseconds. The broadcast approach ensures that all remote consumers of a RTO receive it simultaneously.

The test system does not use a real-time operating system. This is not needed, but it could improve the design. The advantage of a real-time operating system is that it would be possible to calculate on deadlines and ensure that none are missed. The code also gets more maintainable as each task becomes simpler.

Measurements has shown that interrupts are blocked for a long time, and that this could cause some problems for real-time applications. A change of the middleware to use RTOS would eliminate this problem if the *slave_task* of the Communication Service was an OS task, and the receiver interrupt service routine did not have to do checksum calculations and framing work.

Time did not allow us to model check the design in UUPALL. We gave that a low priority because it was not our goal to make a bullet proof design. Just a proof of concept.

With another perspective on our middleware it can be seen as a simple event based distributed operating system.

10.1 Contribution

This project has been followed with great interest from some of our colleges.

A research project at VIA University College, Department of ICT Engineering, with the title: "A real-time Java tool chain for resource constrained platforms" [27], will implement a real industrial case in Java using the HVM[9] and our middleware. This is going to be a proof-of-concept to show the community that Java is ready to be used for embedded systems.

[9] A Java development and execution environment for resource constrained embedded systems.

11 References

[1] Alberto Sangiovanni-Vincentelli, "Integrated Electronics in the Car and the Design Chain Evolution or revolution?," in *Design, Automation and Test in Europe*, Munich, Germany, 2005.

[2] Atmel Corporation. Automotive Selector - Automotive Microcontrollers. [Online]. http://www.atmel.com/products/automotive/automotive_microcontrollers/avr-based_microcontroller.aspx [Accessed] 03-11-2014.

[3] Johann Stelzer. (2005, February) LIN Bus--An Emerging Standard for Body Control Applications. [Online]. http://electronicdesign.com/automotive/lin-bus-emerging-standard-body-control-applications [Accessed] 15-09-2014.

[4] CENELEC, "Medical device software - Software life-cycle processes (DS/EN 62304:2006)," 2006.

[5] Rainer Finocchiaro, "Design and Implementation of a Realtime CORBA Event Service with Support for a Realtime Network," Aachen, Diploma Engineering Thesis 2002.

[6] Object Management Group (OMG). (2004) Object Management Group (OMG) - Event Service Specification - version 1.2. [Online]. http://www.omg.org/spec/EVNT/1.2/PDF [Accessed] 7-11-2014.

[7] Carlos O'Ryan, David L. Levine, Douglas C. Schmidt, and J. Russel Noseworthy, "Applying a scalable CORBA event service to large-scale distributed interactive simulations," in *Object-Oriented Real-Time Dependable Systems, 1999. WORDS 1999 Fall. Proceedings*, Monterey , California, 1999.

[8] Object Computing Inc. (2013) Home. [Online]. http://www.theaceorb.com/ [Accessed] 2-11-2014.

[9] Allen Newell and Herbert A. Simon, "Computer Science as Empirical Inquiry: Symbols and Search," *Commun. ACM*, vol. 19, no. 3, pp. 113-126, 1976.

[10] Mikael Berndtsson, Jörgen Hansson, Björn Olsson, and Björn Lundell, *Thesis Projects A Guide for Students in Computer Science*, 2nd ed. London: Springer-Verlag London Limited, 2008, ISBN 13: 978-1-84800-009-4.

[11] ISO/IEC, "Software engineering - Product quality - Part 1: Quality model (ISO/IEC 9126-1)," Geneva, 2001.

[12] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, Anne-Marie Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114-131, 2003.

[13] Hermann Kopetz, *Real-Time Systems, Design Principles for Distributed Embedded Applications*.: Springer, 2011, p. Chapter 5 Temporal Relations, ISBN 978-1-4419-8236-0.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, 1st ed.: Addison-Wesley, 1995, ISBN-13: 078-5342633610.

[15] Carlos O'Ryan, Douglas C. Schmidt, and J. Russel Noseworthy. (2001) Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations. [Online]. http://www.cs.wustl.edu/~schmidt/PDF/CSSE.pdf [Accessed] 17-11-2014.

[16] LIN Steering Group, "LIN Specification Package Rev. 2.2A," LIN Consortium, Standard 2010.

[17] Modbus.org. [Online]. http://modbus.org/ [Accessed] 15-10-2014.

[18] Andrew S. Tanenbaum, *Computer Networks, Third Edition*, 3rd ed.: Prentice Hall International Editions, 1996, ISBN 0-13-394248-1.

[19] T.C. Maxino and P.J Koopman, "The Effectiveness of Checksums for Embedded Control Networks," *Dependable and Secure Computing, IEEE Transactions on*, vol. 6, no. 1, pp. 59-72, 2009.

[20] Alan Burns and Andy Wellings, *Real-Time Systems and Programming Languages, ADA, Real-Time Java and C/Real-Time POSIX*, 4th ed. Essex, England: Pearson Education Limited, 2009, ISBN: 978-0-321-41745-9.

[21] MIRA Limited, *MISRA C:2012 Guidelines for the use of the Clanguage in critical systems*, 2012th ed. Warwickshire: MIRA Limited, 2013, ISBN 978-1-906400-11-8.

[22] Michael Feathers et al. Cpputest. [Online]. https://cpputest.github.io/ [Accessed] 14-08-2014.

[23] Doxygen. [Online]. http://www.doxygen.org/ [Accessed] 12-03-2014.

[24] (2015) CLOC Count Lines of Code. [Online]. http://cloc.sourceforge.net/ [Accessed] 17-02-2015.

[25] Object Management Group (OMG). (2014) Object Management Group (OMG). [Online]. http://omg.org/ [Accessed] 16-12-2014.

[26] Object Management Group (OMG). (2014) Common Object Request Broker Architecture (CORBA) Specification, Version 3.3 - Part 1: CORBA Interfaces. [Online]. http://www.omg.org/spec/CORBA/3.3/Interfaces/PDF [Accessed] 15-12-2014.

[27] Stephan K. Korsholm, Hans Søndergaard, and Anders Peter Ravn, "A real-time Java tool chain for resource constrained platforms," *Concurrency and Computation: Practice & Experience*, vol. 2013, no. 1532-0626, pp. 1-25, September 2013.

[28] Stephen Cass, Nick Diakopoulos, and Joshua Romero. (2014, July) Interactive: The Top Programming Languages. [Online]. http://spectrum.ieee.org/static/interactive-the-top-programming-languages [Accessed] 21-9-2014.

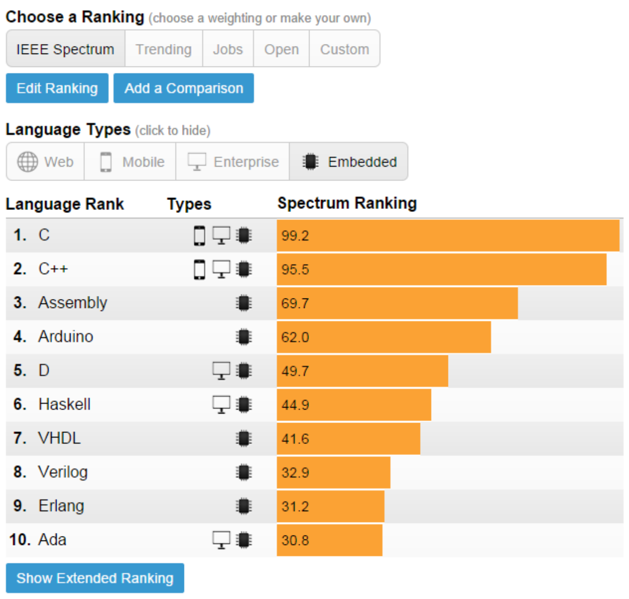
12 Glossary

ACE	Adaptive Communication Environment is an open-source framework used for network programming
ADT	Abstract Data Type
C99	A standard for the C programming language
CAN	Controller Area Network
COM-Service	Our communication service
Consumer	Part of an application that uses data that have been supplied by other parts (suppliers)
CORBA	Common Object Request Broker Architecture is a standard maintained by OMG
CORBA Event Service	A service that allows clients and servers to communicate asynchronously
CRC	Cyclic Redundancy Check – a method to add check sequence to a data stream
Doxygen	Source code documentation tools
DRE	Distributed Real-time Embedded System
ECGW	Our Event Channel Gateway
ECU	Electronics Control Unit
EMI	Electro Magnetic Interference
Event Channel Gateway	Our implementation of a mediator for communication between distributed Event Services
FIFO	First In First Out data structure
IDL	CORBA’s Interface Description Language
IECGW	Interface definition of an Event Channel Gateway
IIOP	Internet Inter-ORB Protocol – is a standard for mediation of CORBA invocations over the TCP/IP protocol
LIN	Local Interconnect Network is a standard by the LIN Consortium, for creating hierarchical vehicle networks
MCU	Micro Controller Unit
MinGW	A contraction of "Minimalist GNU for Windows", is a minimalist development environment for native Microsoft Windows applications.
MMU	Memory Management Unit
Modbus	A serial communications protocol originally published by Modicon (now Schneider Electric) in 1979 for use with its programmable logic controllers (PLCs)
MW	Middleware
OMG	Object Management Group

OO	Object Orientation
PWM	Pulse Width Modulation, typically used for controlling the speed of motors etc.
QoS	Quality of Service
Real time CORBA	Real-time support development of real-time systems
RMI	Remote Method Invocation
ROS	Robot Operating System
RPC	Remote Procedure Call
RS232	The Electronic Industries Association (EIA) standard RS-232-C for full duplex serial point to point communication
RS485	Electrical characteristics of the generator and the receiver for half duplex serial bus communication with up to 32 nodes
RTO	Real Time Object
Supplier	Part of an application that can supply data to be used of other parts (consumers)
TAO	The ACE ORB – a real-time CORBA ORB
UART	Universal Asynchronously Receiver Transmitter
UI	User Interface
UML	Unified Modelling Language – a standard for documentation of software designs etc.

Appendix A Embedded Programming Languages

Ranking of programming languages used for embedded development July 3, 2014. [28].



Language Rank	Types	Spectrum Ranking
1. C	Mobile, Enterprise, Embedded	99.2
2. C++	Mobile, Enterprise, Embedded	95.5
3. Assembly	Embedded	69.7
4. Arduino	Embedded	62.0
5. D	Enterprise, Embedded	49.7
6. Haskell	Enterprise, Embedded	44.9
7. VHDL	Embedded	41.6
8. Verilog	Embedded	32.9
9. Erlang	Embedded	31.2
10. Ada	Enterprise, Embedded	30.8

Appendix B Example of output from COM-Simulator

```
Program started!
Obtain Proxy RTO1
Obtain Proxy RTO2
Connect supplier: RTO1, SUPPLIER1_ID

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_SERVICE
              Len:          12
              Message-Type: MESS_TYPE_SERVICE
              service_type: SERVICE_TYPE_RTO_SUPPLIED
              RTO_id:       RTO1
COM-Service Polling ended

COM-Service pushed COM_MESS_TYPE_SERVICE: from_node_id: 1, service_type: SER-
VICE_TYPE_RTO_CONSUMED, RTO_id: RTO1

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_SERVICE
              Len:          12
              Message-Type: MESS_TYPE_SERVICE
              service_type: SERVICE_TYPE_RTO_SUPPLIED
              RTO_id:       RTO1
COM-Service Polling ended

COM-Service pushed COM_MESS_TYPE_SERVICE: from_node_id: 1, service_type: SER-
VICE_TYPE_RTO_SUPPLIED, RTO_id: RTO2

COM-Service Polling started
COM-Service Polling ended
Connect consumer: RTO2, con_push1

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_SERVICE
              Len:          12
              Message-Type: MESS_TYPE_SERVICE
              service_type: SERVICE_TYPE_RTO_CONSUMED
              RTO_id:       RTO2
COM-Service Polling ended

COM-Service pushed COM_MESS_TYPE_SERVICE: from_node_id: 1, service_type: SER-
VICE_TYPE_RTO_SUPPLIED, RTO_id: RTO2

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_SERVICE
              Len:          12
              Message-Type: MESS_TYPE_SERVICE
              service_type: SERVICE_TYPE_RTO_CONSUMED
              RTO_id:       RTO2
COM-Service Polling ended
Connect consumer: RTO1, con_push

COM-Service Polling started
COM-Service Polling ended

COM-Service pushed COM_MESS_TYPE_PUSH: from_node_id: 1, push_type: PUSH_TYPE_UPDATED,
RTO_id: RTO2
   value: [00] [10] [0F] [00]

con_push1 called :rto_id: RTO2 event: RTO_EVENT_UPDATED
   value len: 4, value: [00] [10] [0F] [00]
```

```
COM-Service Polling started
COM-Service Polling ended
Push 0: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
              Len:          13
              Message-Type: MESS_TYPE_PUSH
              push_type:    PUSH_TYPE_UPDATED
              RTO_id:       RTO1
              value:        [01]
COM-Service Polling ended
Push 1: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
              Len:          13
              Message-Type: MESS_TYPE_PUSH
              push_type:    PUSH_TYPE_UPDATED
              RTO_id:       RTO1
              value:        [01]
COM-Service Polling ended
Push 2: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
              Len:          13
              Message-Type: MESS_TYPE_PUSH
              push_type:    PUSH_TYPE_UPDATED
              RTO_id:       RTO1
              value:        [01]
COM-Service Polling ended
Push 3: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
              Len:          13
              Message-Type: MESS_TYPE_PUSH
              push_type:    PUSH_TYPE_UPDATED
              RTO_id:       RTO1
              value:        [01]
COM-Service Polling ended
Push 4: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
              Len:          13
```

```
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 5: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 6: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 7: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 8: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 9: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started
```

```
   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 10: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 11: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 12: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 13: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 14: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
```

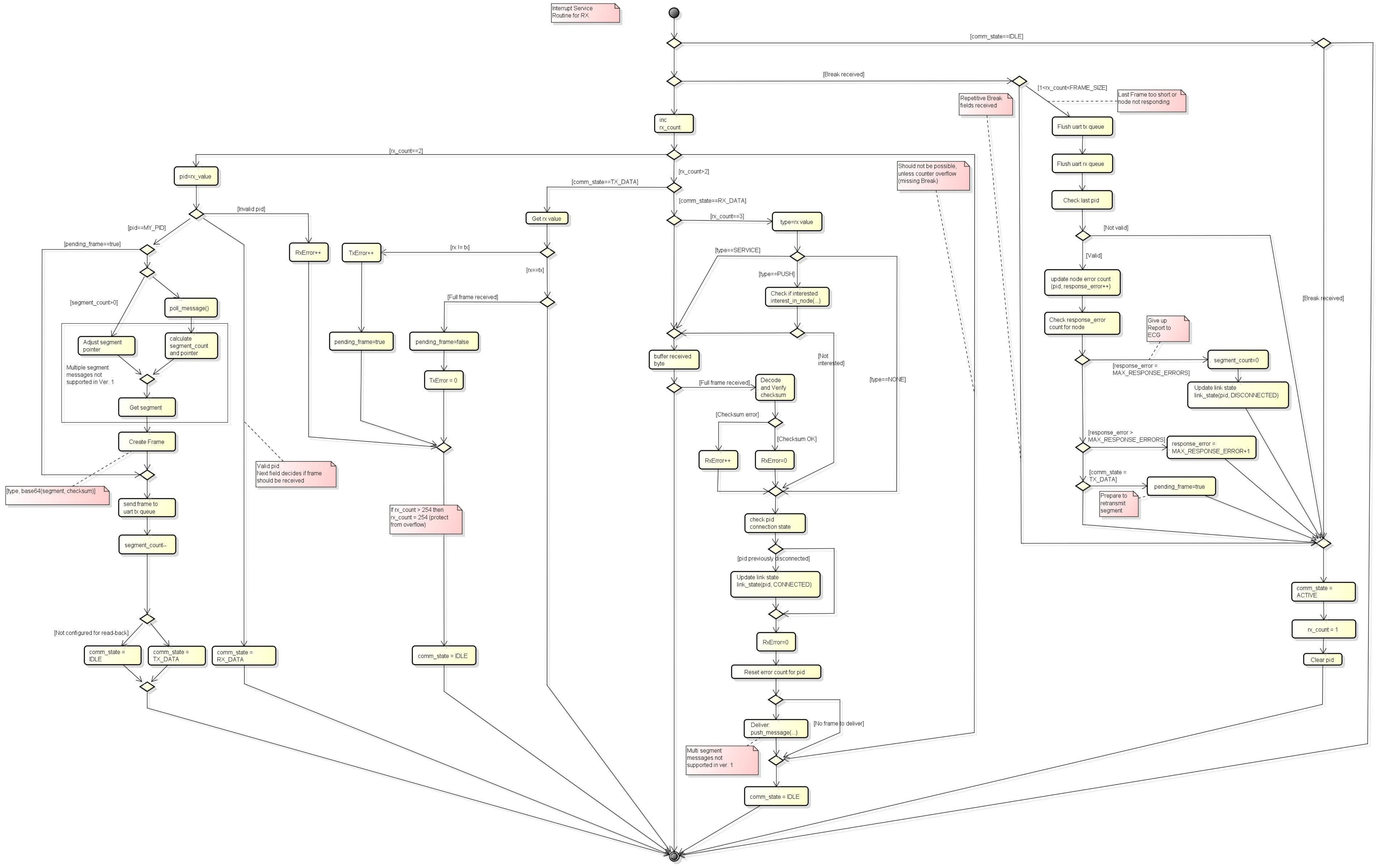
```
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 15: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 16: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 17: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
Push 18: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
             Len:          13
             Message-Type: MESS_TYPE_PUSH
             push_type:    PUSH_TYPE_UPDATED
             RTO_id:       RTO1
             value:        [01]
COM-Service Polling ended
```

```
Push 19: RTO1

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
              Len:          13
              Message-Type: MESS_TYPE_PUSH
              push_type:    PUSH_TYPE_UPDATED
              RTO_id:       RTO1
              value:        [01]
COM-Service Polling ended

COM-Service Polling started
COM-Service Polling ended

COM-Service Polling started
COM-Service Polling ended

COM-Service Polling started
COM-Service Polling ended

COM-Service Polling started
COM-Service Polling ended

COM-Service Polling started
COM-Service Polling ended

set_update_perode(RTO1, 5) called

es_timer_tick() called

es_timer_tick() called

es_timer_tick() called

es_timer_tick() called

es_timer_tick() called

con_push called :rto_id: RTO1 event: RTO_EVENT_STATUS_CHANGED
   Status: PROXY_STATUS_TIMED_OUT

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
              Len:          14
              Message-Type: MESS_TYPE_META
              meta_type:    META_TYPE_UPDATE_PERIODE
              RTO_id:       RTO1
              value:        [05] [00]

   COM-Message-Type: COM_MESS_TYPE_PUSH
              Len:          12
              Message-Type: MESS_TYPE_STATUS
              status_type:  PROXY_STATUS_TIMED_OUT
              RTO_id:       RTO1
COM-Service Polling ended

es_timer_tick() called

COM-Service Polling started
COM-Service Polling ended

set_update_perode(RTO1, 5) called
```

```
push(RTO1) called

con_push1 called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [01]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
              Len:          13
              Message-Type: MESS_TYPE_PUSH
              push_type:    PUSH_TYPE_UPDATED
              RTO_id:       RTO1
              value:        [01]
COM-Service Polling ended

Value of RTO1 changed to 10

push(RTO1) called

con_push1 called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [0A]

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [0A]

COM-Service pushed COM_MESS_TYPE_SERVICE: from_node_id: 1, service_type: SER-
VICE_TYPE_RTO_CONSUMED, RTO_id: RTO1

COM-Service pushed COM_MESS_TYPE_SERVICE: from_node_id: 1, service_type: SER-
VICE_TYPE_RTO_SUPPLIED, RTO_id: RTO2

Value of RTO1 changed to 11

push(RTO1) called

con_push1 called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [0B]

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [0B]

Disconnect supplier from RTO1

con_push1 called :rto_id: RTO1 event: RTO_EVENT_STATUS_CHANGED
   Status: PROXY_STATUS_NO_SUPPLIER

con_push called :rto_id: RTO1 event: RTO_EVENT_STATUS_CHANGED
   Status: PROXY_STATUS_NO_SUPPLIER

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
              Len:          13
              Message-Type: MESS_TYPE_PUSH
              push_type:    PUSH_TYPE_UPDATED
              RTO_id:       RTO1
              value:        [0B]

   COM-Message-Type: COM_MESS_TYPE_PUSH
              Len:          12
              Message-Type: MESS_TYPE_STATUS
              status_type:  PROXY_STATUS_NO_SUPPLIER
              RTO_id:       RTO1
   COM-Message-Type: COM_MESS_TYPE_SERVICE
              Len:           12
              Message-Type:  MESS_TYPE_SERVICE
              service_type:  SERVICE_TYPE_RTO_NOT_SUPPLIED
              RTO_id:        RTO1
COM-Service Polling ended

Value of RTO2 changed to 15, 12

remote push(RTO2) called

COM-Service pushed COM_MESS_TYPE_PUSH: from_node_id: 1, push_type: PUSH_TYPE_UPDATED,
RTO_id: RTO2
   value: [0C] [00] [0F] [00]

con_push1 called :rto_id: RTO2 event: RTO_EVENT_UPDATED
   value len: 4, value: [0C] [00] [0F] [00]

Value of RTO1 changed to 12

push(RTO1) called

con_push1 called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [0C]

con_push called :rto_id: RTO1 event: RTO_EVENT_UPDATED
   value len: 1, value: [0C]

COM-Service Polling started

   COM-Message-Type: COM_MESS_TYPE_PUSH
              Len:           13
              Message-Type:  MESS_TYPE_PUSH
              push_type:     PUSH_TYPE_UPDATED
              RTO_id:        RTO1
              value:         [0C]
COM-Service Polling ended

Program Ended!
```

.

Appendix C Activity diagram for Slave task


Interrupt Service Routine for RX
[comm_state==IDLE]
[Break received]
inc rx_count
[rx_count==2]
pid=rx_value
[Invalid pid]
[pid==MY_PID]
[pending_frame==true]
[segment_count>0]
poll_message()
Adjust segment pointer
calculate segment_count and pointer
Multiple segment messages not supported in Ver. 1
Get segment
Create Frame
[type, base64(segment, checksum)]
send frame to uart tx queue
segment_count--
[Not configured for read-back]
comm_state = IDLE
comm_state = TX_DATA
comm_state = RX_DATA
Valid pid Next field decides if frame should be received
RxError++
TxError++
[rx != tx]
[rx==tx]
[Full frame received]
pending_frame=true
pending_frame=false
TxError = 0
If rx_count > 254 then rx_count = 254 (protect from overflow)
comm_state = IDLE
[rx_count>2]
[comm_state==TX_DATA]
Get rx value
[comm_state==RX_DATA]
[rx_count==3]
type=rx value
[type==SERVICE]
[type==PUSH]
Check if interested interest_in_node(...)
[Not interested]
[type==NONE]
buffer received byte
[Full frame received]
Decode and Verify checksum
[Checksum error]
[Checksum OK]
RxError++
RxError=0
check pid connection state
[pid previously disconnected]
Update link state link_state(pid, CONNECTED)
RxError=0
Reset error count for pid
Deliver: push_message(...)
[No frame to deliver]
Multi segment messages not supported in ver. 1
comm_state = IDLE
Should not be possible, unless counter overflow (missing Break)
Repetitive Break fields received
[1<rx_count<FRAME_SIZE]
Last Frame too short or node not responding
Flush uart tx queue
Flush uart rx queue
Check last pid
[Not valid]
[Valid]
update node error count (pid, response_error++)
Check response_error count for node
Give up Report to ECG
[response_error = MAX_RESPONSE_ERRORS]
segment_count=0
Update link state link_state(pid, DISCONNECTED)
[response_error > MAX_RESPONSE_ERRORS]
response_error = MAX_RESPONSE_ERROR+1
[comm_state = TX_DATA]
pending_frame=true
Prepare to retransmit segment
[Break received]
comm_state = ACTIVE
rx_count = 1
Clear pid

Appendix D RS-485 Driver

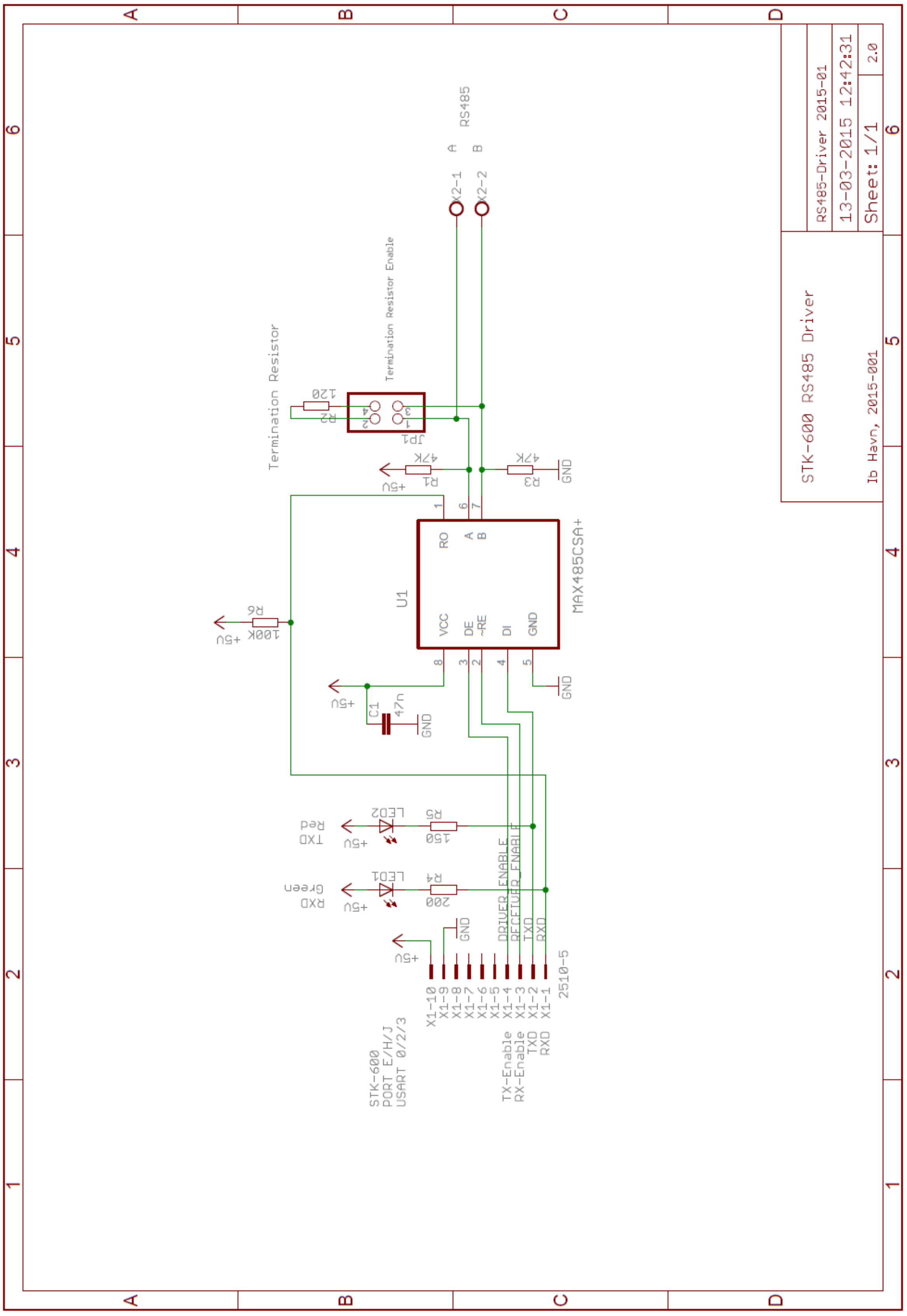


Figure 12-1 RS-485 Driver Schematics

Appendix E Electronic Media Contents

The CDROM has an *index.html* file that shows what the CDROM contains and has links to the content:

1. The Thesis in PDF-format
2. Source code for Event-Service including Event Channel Gateway
3. Doxygen documentation for Event-Service and Event Channel Gateway
4. Source code for COM-Service
5. Doxygen documentation for COM-Service
6. Source code for demo-application
7. Unit test source code for Linked List and FIFO data structures
8. Unit test source code for COM-Service