Architecture of distributed real-time embedded system

CHEUK WING LEUNG

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:155

KTH Information and
Communication Technology

Architecture of distributed real-time embedded system

Master of Science Thesis

CHEUK WING LEUNG

Examiner:
Mats Brorsson, KTH

Supervisor:
Detlef Scholle, XDIN AB
Barbro Claesson, XDIN AB

TRITA-ICT-EX-2013:155

Acknowledgements

My deepest gratitude goes to my supervisors Detlef Scholle and Barbro Claesson for giving me this opportunity to work in the project. They have inspired me and given me valuable advices throughout the project.

I would like to thank the examiner Mats Brorsson for the support and guidance to my thesis project.

I thank the employees in XDIN AB for good company and creating an good environment during the project. Especially Sebastian Ullström and David Skoglund for the help in testing of the system and developing the demonstration setup. I want to thank Joanna Larsson, Amir Kouhestani, José Pérez, Robin Hultman and Tobias Lindblad, the students who working together in the same project for moral support and inspiring discussions on details of the project.

Finally, I would like to thank my parents and my friends, especially Markus Näslund for all of the support.

Abstract

CRAFTERS (Constraint and Application Driven Framework for Tailoring Embedded Real-time System) project aims to address the problem of uncertainty and heterogeneity in a distributed system by providing seamless, portable connectivity and middleware.

This thesis contributes to the project by investigating the techniques that can be used in a distributed real-time embedded system. The conclusion is that, there is a list of specifications to be meet in order to provide a transparent and real-time distributed system.

This thesis has implemented a basic system that provides support of scalability, accessibility, fault tolerant and consistency. The system is tested in different areas and it shows its potentials to be a well transparent real-time system. This built the basis for further development.

Contents

List of Figures

List of Tables

List of Abbreviations

ABS Anti-lock Braking System

ACID Atomicity, Consistency, Isolation and Durability

API Application Programming Interface

ARS Address Resolution Sevice

ASIL Automotive Safety Integrity Level

AUTOSAR Automotive Open System Architecture

BCCT Best Case Communication Time

BCET Best Case Execution Time

CRAFTERS Constraint and Application Driven Framework for Tailoring Embedded Real-time System

DII Dynamic Invocation Interface

DRE Distributed Real-time Embedded

DSI Dynamic Skeleton Interface

DVFS Dynamic Voltage and Frequency Scaling

ECSI Electronic Chips & Systems Design Initiative

ECU Engine Control Unit

EDF Earliest Deadline First

FT Fault Tolerant

I/O Input and Output

IDL Interface Definition Language

LIST OF ABBREVIATIONS

ITEA2 Information Technology for European Advancement

MANY Many-core Programming and Resource Management for High-performance Embedded System

MBAT Combined Model-based Analysis and Testing of Embedded Systems

MLF Minimum Laxity First

MUF Maximum Urgency First

ORB Object Request Broker

OSA+ Open System Architecture Platform for Universal Services

QoS Quality of Service

RMS Rate Monotonic Scheduling

RT CORBA Real-time Common Object Request Architecture

RTE Run-time Environment

TAO The Adaptive Communication Environment ORB

VFB Virtual Functional Bus

WCCT Worst Case Communication Time

WCET Worst Case Execution Time

Chapter 1

Introduction

1.1 Background

There is an increasing demand of higher performance in embedded system. The system development trend is therefore shifting from centralized embedded systems to distributed embedded systems, motivated, in addition, by the reason that distributed systems generally have higher scalabilities, reliabilities and flexibilities than centralized systems. Embedded systems in, for example, brake-by-wire system and aircraft system require both high-processing performance as well as to meet the time constraint and it leads to a rising of importance of Distributed Real-time Embedded (DRE) system.

A distributed real-time embedded system works similar to other real-time embedded systems, but with a high degree of heterogeneity. The heterogeneity allows the system to perform parallel tasks that have different requirements on the hardware, which in terms benefit the functionality of the system. A DRE system is heterogeneous in many aspects, from hardware structures and software components to the scheduling policies, communication protocols and memory managements. The processing performance and flexibility of the system can benefit from the complexity of the heterogeneity. Such complexity, however, introduces new challenges in maintaining and coordinating the system. It also increases the difficulties for developers to program parallel applications that can fully utilize the processing potential of the system.

The communication across nodes and the scheduling and mapping of tasks in a DRE system can be complicated. A middleware is often used in such system to maintain the coordination and to allow the different independent hardware components of the network work as a whole system. Apart from the communication and dispatching of tasks, the system also needs to meet the real-time constraint as well as other constraints for an embedded system. This raises the challenge for designing a middleware for a DRE system.

1.2 Problem statement

There are a lot of aspects to be considered when constructing a middleware and the focus of this thesis is to serve as a study of essential components needed in a distributed real-time embedded system. It is going to exploit how developer can handle the complexity by adopting protocols and techniques that are suitable for such a system in the middleware. The thesis is going to answer the following questions:

- *What structure and components are required in a distributed real-time embedded system?*
- *What different protocols and techniques can be used to develop a DRE system?*
- *How does the middleware handle the coordination of the system?*
- *How should the middleware be constructed in order to maintain high scalability and transparency?*

1.3 Team goal

This work contributes to a project that builds up a distributed real-time embedded system, which is used as a brake-by-wire system in automobile. The contribution of this work provides a middleware and design of the architecture of a distributed embedded system. The goal of the project is to achieve real-time performance with low power consumption and maintain the transparency of a distributed system. It also aims to maximize the performance on real-time and safety-critical aspects.

1.4 Method

1.4.1 Theoretical study

In the theoretical study, an in-depth study of distributed real-time embedded system should be done. Different architectures, protocols and topologies of DRE system will be studied and investigated.

1.4.2 Implementation

In the implementation phase, the control logic of a middleware is defined and the middleware is implemented to manage a distributed real-time embedded system that demonstrates the coordination within the network.

1.5 Delimitations

The support of real-time performance requires OS and hardware support. In this project, deterministic communication platform and real-time constraint are ad-

dressed in best effort but is not solved in this project as providing the solution for real-time support from OS and hardware is not in the scope of this work. The distributed real-time embedded system, therefore, does not suppose to be completed in this project, but the core functionalities of the system should be constructed through this project.

Chapter 2

Requirements of the system

2.1 Overview of the requirements

To help defining a clearer goal for the work, the system designed in this work should be able to fulfill the folloing requirements:

SW_REQ_1 For hard deadline tasks, the system must always meet the deadlines.

SW_REQ_2 For soft deadline and firm deadline tasks, the deadlines must be met in majority of the time.

SW_REQ_3 The system must have small memory footprint and can be mapped to small devices.

SW_REQ_4 Scaling up of the system must not result in severe deterioration of performance

SW_REQ_5 The middleware shall be able to control the system regardless the diversity of the hardware and the operating system.

SW_REQ_6 Tasks in the system shall be able to handle hardware failure without interference from the user.

SW_REQ_7 The DRE system shall guarantee an end-to-end quality of service.

SW_REQ_8 Task in the system shall be able to interact with other tasks in same manner regardless where and how the target task is.

SW_REQ_9 The communication and resource accessing in the system must be reliable and data consistency must be preserved throughout the system.

In this chapter, the explanations of these requirements are discussed.

2.2 Explanations of the requirements

SW_REQ_1 For hard deadline tasks, the system must always meet the deadlines.

SW_REQ_2 For soft deadline and firm deadline tasks, the deadlines must be met in majority of the time.

The system is designed for distributed real-time embedded system, therefore, the real-time constraints have to be fulfilled. THe system should therefore provide the support for handling real-time tasks. Consider this system would be tested with a brake-by-wire system, which is a hard real-time system, the system should provide real-time support for handling hard deadline tasks, soft deadline tasks and firm deadline tasks. The distinctions between these three types of real-time requirements are further explained in section 3.2.1.

SW_REQ_3 The system must have small memory footprint and can be mapped to small devices.

As an embedded system, the memory usage of the system should not be high and should be kept as minimal, due to the fact that an embedded system usually provides limited resources. The use of resources for communication should be designed to be efficient and the system should be able to run on embedded system smoothly.

SW_REQ_4 Scaling up of the system must not result in severe deterioration of performance.

The system should be designed to handle different scale of hardware. The system should be performing as deterministic as possible regardless the scale of the hardware.

SW_REQ_5 The middleware shall be able to control the system regardless the diversity of the hardware and the operating system.

As a distributed system, the hardware structures between each node can be different. The middleware should be handling the system in a general method instead of giving specific solution to just a certain type of hardware or operating system.

SW_REQ_6 Tasks in the system shall be able to handle hardware failure without interference from the user.

The distributed property of the system means that the system contains many different parts of hardware. This nature can increase the risk of hardware. In order not to corrupt the whole system because of any non-crucial failure in hardware, the system should be decided to handle these problems.

SW_REQ_7 The DRE system shall guarantee an end-to-end quality of service.

This system is targeting real-time tasks and it involves a lot of communications. The quality of service, especially, the communication latency, should be guaranteed to provide basic support of solving the real-time problem.

SW_REQ_8 Task in the system shall be able to interact with other tasks in same manner regardless where and how the target task is.

Since the system is running on different targets, a common interface should be developed such that the development of tasks can be done in a same style regardless how the system structure is and what different hardware it is using. The tasks should be able to assign to run on different node without changing of interfaces.

SW_REQ_9 The communication and resource accessing in the system must be reliable and data consistency must be preserved throughout the system.

A common resource is accessible throughout the system, policies need to set up to guarantee there is no conflicts in accessing resources, such that the system give a more deterministic behavior.

Chapter 3

Distributed Real-time Embedded System

3.1 Motivation on distributed system

The traditional systems are constructed in the way of centralized system. Besides this traditional approach, some other approaches like decentralized system and distributed system are also often used. The trend of constructing a system is shifting from centralized system to distributed system. In the following, these three approaches are introduced and compared.

3.1.1 Centralized system

Embedded systems are traditionally centralized systems. The structure of a centralized system can be conceived as in Figure 3.1. Such system has one controller that coordinates different component units in a system. The component units, including sensors and actuators, are often located closely and linked directly to the controller.

In system that is small-scaled, this kind of systems can easily be implemented. Without a hierarchical controlling structure, the controlling of sensors and actuators is direct and fast, and therefore, it is very suitable for small system that requires high performance. As the communication cost between sensors and controller is low, such system is also a good candidate of implementing real-time system.

Centralized system does not scale well though. An increase of component units not only would result in an increasing number of connections, but also would increase the complexity of handling the control of the units. The communication demand would require the controller to have a high bandwidth and a high performance to process all the operations. On top of that, central control unit would confine the physical routing and layout of the sensors and actuators.

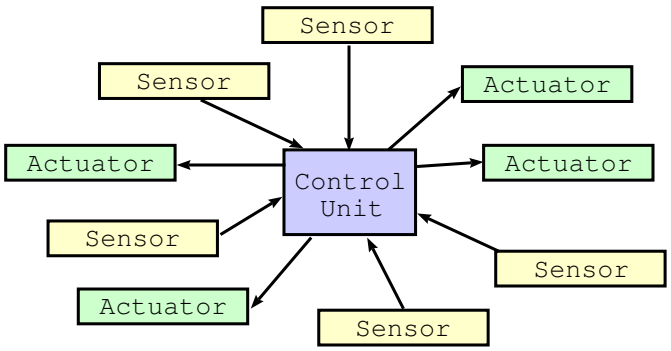


Figure 3.1. Centralized System

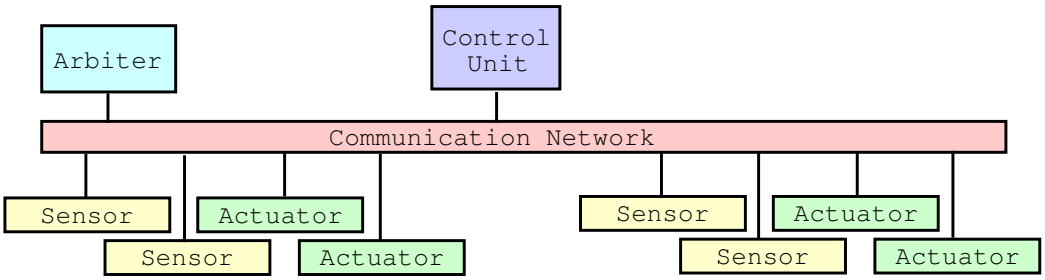


Figure 3.2. Decentralized System

3.1.2 Decentralized system

Decentralized system suggests the moving of sensors and actuators away from the controlling unit. Instead of directly connecting to the control unit, the component units connect to the control unit through a connection network or a communication medium[1]. Figure 3.2 shows the general structure of a decentralized system.

Because of the support of communication network, the sensors and actuators can be located further away from the controlling unit. However, as the component units are sharing the same medium, additional hardware logic, like an arbiter, has to be implemented to take care of the communication between component units and the control unit. Policies and protocols are also needed to determine which component is allowed to use the common network, or else, conflicts may occur in the network and it may result in loss of data or faulty packet.

This kind of systems allows easier routing and to layout the system. With the communicating medium, it scales better than the traditional centralized system. However, like the centralized system, there exists only a single controller that handles the operations. The performance requirement is therefore still very high and the control logic would become very complex when the system scales.

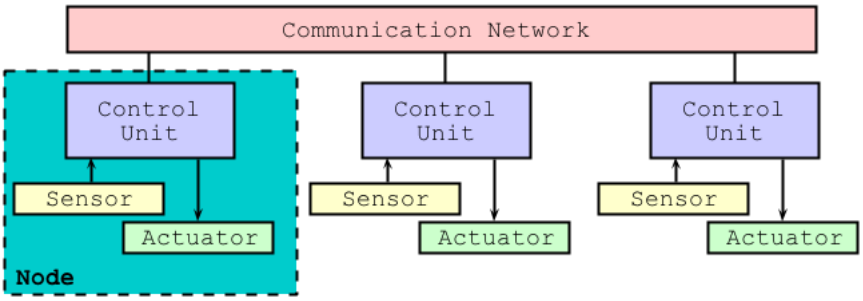


Figure 3.3. Distributed System

3.1.3 Distributed system

To put the decentralization further, distributed system introduces more control units into the system. Each of the control unit is directly connected to the sensors and the actuators that are local to the control unit. This kind of sub-network is referred to a "node" in a distributed system. Each of the control unit handles the local sensors and the local actuators. The whole system is constructed by connecting these nodes together. The nodes cooperate with each other and each node provides services for other nodes to use the local sensors and local actuators as well as other resources.

The structure a distributed system is shown in Figure 3.3. The hierarchical architecture of the system allows the system to have a better scalability. Since each of the control unit is connected directly to the local component units, there is no need for a complicated routing in the system view and the physical constraints is no longer a problem. With the direct connection to sensors and actuators, the operation can be done with higher performance and security.

When the system scales, additional control units can be added to handle the additional component units. This property also allow a more flexible system as the reconfiguration of the system is done by simple addition of hardware units. However, sophisticated software, known as middleware, has to be developed to handle the interaction between each node[2]. Comparing to centralized system and decentralized system, distributed system also provides a much higher degree of transparency. The transparency of the system is discussed in section 3.3.4.

3.2 Background of Distributed Real-time Embedded System

A distributed real-time embedded system falls into real-time system domain, embedded system domain and distributed system domain. The distributed system is introduced in the previous section. In this section, the other two domains are introduced and the challenges in all of the three domains are discussed.

3.2.1 Real-time system

A real-time system does not only require the operations are rightly performed. It also requires the results to be delivered within a physical time[3]. The system has to be able to handle the tasks that are running in the system with concern of timing. The period of tasks, execution time of the tasks and the deadline of the tasks need to be considered when the system schedules the execution sequence of the tasks. Real-time systems often involve interaction with the surrounding environment and give responses to the environment according to the status of the surrounding. Real-time system can be categorized into three groups regarding the timing constraints, which are soft real-time, hard real-time and firm real-time.

Soft real-time

Soft real-time system is a system that the operations are required to be executed within time requirements. However, if the operations are not completed within the time limit, the system does not create harmful consequence. Therefore infrequent missing of deadlines is tolerated.

Hard real-time

Hard real-time system meaning that missing of deadlines of the operations of the systems would result in a catastrophic consequence in terms of life and death. A hard real-time system should be safety-critical and the system should fulfill all the timing requirements.

Firm real-time

A firm real-time operation requires the execution of it to be completed before the deadline. If the deadline is missed, the results of the operation are no longer useful to the system, and the operation should be aborted. Missing deadlines in such system would affect the Quality of Service (QoS) of the system.

3.2.2 Embedded system

Distributed real-time embedded system is a distributed approach to implement a real-time system with embedded constraints. An embedded system is a system that is usually designed for a particular task. The functions of an embedded system should not change during its lifetime. It should be a low-cost and power-effective system with a low memory requirement[4].

Some common uses of DRE system are the supporting system of aircraft configuration management, brake-by-wire automobile system and autonomous system used in modern building. Different DRE systems, depending on the usage, would have different real-time constraints.

3.3 Challenges in distributed real-time embedded system

A distributed real-time embedded system has to fulfill the requirements for embedded system, distributed system and real-time system. In this section, the concerns on each of the domains are discussed.

3.3.1 Embedded concerns

The shifting from centralized system to a distributed system requires extra hardware in supporting the system, and the large scale of system may result in a need of large middleware to take care of the system. This may result in a middleware that is large in size or consume too much power for the use of an embedded system[5]. The complexity may also lead to a risen demand of higher performance processing units, which, for an embedded system, is not preferable. Building a DRE can be time-consuming as the distribution of components result in a complicated controlling and communicating logic.

To address these concerns, the middleware needs to be designed with only the crucial functions to minimize the memory footprint. A well-structured system should be designed to minimize the complexity within the network but with the possibility of scaling up. Also, in order to shorten the time-to-market, the system should be designed such that it can easily be reused.

3.3.2 Distributed concerns

Due to the hierarchical approach, the communication and the connection between nodes can be complicated[5]. Since there is no shared memory pool between them, often the communication is done by message passing and the choice of tools that can be used would be limited by this restriction. Each of the subsystems may have a different protocol supported for communicating, and in addition, as each of the nodes may be running in different operating system, the coordination done by middleware can become complicated.

To tackle these problems, choosing of communication tool should be made carefully for the targeted platforms. To solve the heterogeneity, APIs of the middleware may need to be implemented to achieve the communication and to monitor and control each of the nodes.

3.3.3 Real-time concerns

The maintaining of predictability for the whole system is far more complicated than that for a multi-core system. It involves in the real-time capability of the underneath system, and to maintain the real-time property, it may induce a large traffic in the network. If the system is a hard real-time system, meaning that the missing of task deadline would result in a catastrophic consequence, the communication protocol and scheduling of tasks should be made fault-tolerant.

The subsystems must therefore be real-time systems and provide enough timing information such as execution time and communication time needed, so the middleware can schedule tasks accordingly. The services within the system should also provide and guarantee the QoS. In addition, the system should also be tested for the security and fault-tolerant to ensure it can handle a hard real-time application.

3.3.4 Dimensions of middleware

As discussed in section 3.1.3, a middleware is often used to maintain the coordination between each node in a distributed system. Middleware is software that enables communications and interactions between executing applications, hardware component, resources and operating system[6]. It should provide basic functional units that assist the tasks to perform successfully in the system and at the same time hides the heterogeneity of the implementations and the system structure from the applications[7].

To have a sustainable middleware, it should be designed to be transparent. The users of the middleware should not need to consider the hardware structure of the system and still be able to program it and the system should work as it is. By transparent, it means that the implementation of interactive mechanism should be invisible to and not concern about the software. The quality of a middleware is determined by its transparency and level of abstraction. In the followings, the different dimensions of transparency are discussed. The dimensions on transparency are based on [8].

Accessibility

Transparent in accessing means that an application should have the same accessing method to a target component regardless of the physical location of that target component[8]. The application, with the help of the middleware, should be able to address the remote component without knowing the physical address. In addition, all open resources within the system network should be accessible by any applications running in the system no matter it is located in the node locally to the application or in a remote node. This type of transparency hides the locality of the targeting component and thus the programmer of the application does not need to take the structure of the system into concerns during coding. The whole system can be treated as a pool of resources and they can be used to perform tasks interactively.

Migratability

If a component can be migrated and relocated from one node to another and the application that requires the component does not need to know the transition nor locating the new address of the component, the system can be said to be transparent in migration. In order to achieve this transparency, the original component and the newly migrated component should be synchronized such that all the status of the

original component is reflected by the migrated component. With this transparency, the system can has a higher flexibility in specifying where the tasks should be executed without disturbing the applications. Once the connection between the applications and the target components is established, the migration of the target components should not affect the linkage. This allows the system to lower the power consumption by having a high degree of freedom in scheduling of tasks. It also provides a safety measures such that when a node fails, the component can migrate to another node and continuous working instead of shutting down the whole system.

Concurrency

The system is concurrency transparent to the users if the mechanism of concurrency handling is hiding from the users. This implies that a user does not need to take care of the concurrency conflicts if there is more than one application accessing the same shared resources in the system. The middleware should resolve such conflicts. If a system is concurrency transparent, the application does not need to know how many applications are sharing the same resources. By this transparency, the system also protects itself from illegal access of resources made by application, as the application should have no control on direct accessing and locking of the resources. The concurrency conflicts should be handled by the middleware but not the application.

Scalability

A good middleware should have be scalability transparent to users. It means that the scaling of the system should not be reflected to the users and the mechanism of the scaling should not be a concern to the user and the application. When the system scales up, the performance should not be deteriorated and communication overhead should not be increased significantly. In order to achieve this type of transparency, the middleware should be able to allocate new resources to handle the demand of newly added units to the system as well as the original units. How the components are mapped should be hidden from the users.

Failure

Failure transparency is to hide the recovery from failure of the system from users. The system should be able to maintain the operation even after a node failed. With the migratability and concurrency of the system, the middleware should be able to direct resources from other part of the system to continue with the job of the failed node. The users should not need to take care of how the system handles the failure and the applications should run as it is without getting affected by the failed node. The measures in identifying and handling the failures should be done by the middleware.

Configurability

Configurability refers to the possibility of the system to be re-designed without large portion of changes.

3.4 Summary of the distributed real-time embedded system

This project is going to focus on building a system that is using the distributed approach and the system should also be a real-time embedded system. The system, therefore, should be able to address the problems that is commonly faced in real-time system, embedded system and distributed system, while the middleware of the DRE system is to be designed to demonstrate the ability in providing different aspects of transparency.

Chapter 4

Middleware Architecture

4.1 Structure of a middleware

The middleware is the backbone of a distributed system. A middleware handles the coordination of the system resources, communications among different components, scheduling and guarantee the quality of service. There are various different approaches to construct a middleware, the functionalities, however, can be summarized into several categories. In the following, the basic functionalities of middleware and their motivations are presented. These functionalities are based on [8] and [3]. Then some implementations of the functionalities that are used in related middleware are studied. This section answers the first question stated in chapter 1:

What structure and components are required in a distributed real-time embedded system?

4.1.1 Communication within the system

Communication unit is one of the most important parts of the middleware. It should be able to handle different communication protocols across the nodes within the network of the system. The unit that handles the communication may or may not include the data transition, but it should provide mechanism in creating and connecting sockets and to allocate buffers. The communication unit usually works together with the unit that helps locating the target object and the controlling units that guard the components. Communications in a network are done commonly in two ways, through shared memory or by message passing.

Shared memory

Communication in shared memory is storing and loading of a memory section that is commonly accessible in the network. The communication needs to be monitored carefully for possible concurrent accessing to the shared memory[9]. This communication scheme is generally faster then the message passing and requires only one single copy of the data, and therefore has a lower memory footprint.

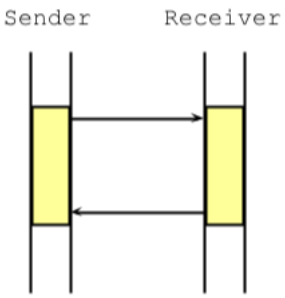


Figure 4.1. Synchronous Communication

Message passing

Sending and receiving data across the communicating parties do communication through message passing. This often requires the communication to have some protocols supported and also requires some common tools that can be used by both ends of the communication[9]. This approach is in general not as fast as using shared memory but it allows the message passing across the boundary of operating system. Due to the differences of the subsystem structure of each of the node, communicating through shared memory is normally not possible in a DRE system and therefore message passing is used.

4.1.2 Synchronous protocol within the system

How the communication is done is governed by the synchronous protocol. A middleware usually defines and handles the protocol that is used within the system such that the application developer needs not to take care of the synchronization between the sender and the receiver in a communication.

In general, there are two basic types of synchronous protocols, synchronous communication and asynchronous communication[8].

Synchronous communication

Synchronous communication ensures the receiver get the message. The idea of this principle is to stall the sender of the message and wait until the receiver receives the message, and sends a reply after the message is processed. Since the sender is on hold by the transporting and processing of the message, it cannot be used to continue with the rest of the tasks or even cannot perform other tasks. Figure 4.1 shows that the sender is always occupied or blocked during the communication. In this figures and the other figures in this document that show the communication policies, the colored areas represent the time that the units are performing communication, during these time slots, the units are not able to perform other tasks.

In synchronous communication, the message guarantees to be received by the receiver. The blocking time for the sender depends on the processing time of the

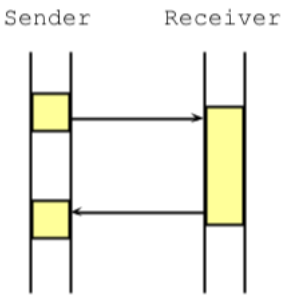


Figure 4.2. Asynchronous Communication

message and also how many other applications are accessing or waiting for accessing the receiver. Therefore, such principle is not recommended for applications that require high level of safety or have a hard or firm deadline. This principle is used in calling of short function and when the rest of the operation requires the results from the receiver.

Asynchronous communication

Asynchronous communication leaves the synchronization to be taken care of by the receiving end of the message. The message from the sender is sent to the receiver. After the receiver has process the message, the results is sent back from the receiver by using the callback function of the sender. Figure 4.2 illustrates such communication.

It can be seen that the communication does not occupy the sender. The synchronization relies on the receiver instead of the sender and the sender is allowed to perform other operations during the receiver is processing the message. Asynchronous gives a shortest response time for sending the results back to the sender, and at the same time, does not requires the sender to be blocked from operating. Therefore, this principle is suitable for safety-critical applications. However, the sender has to be able to provide a callback function for the receiver to return the results.

4.1.3 QoS support

Guaranteeing the Quality of Service (QoS) is very important for a real-time system especially for hard real-time system. A real-time middleware should provide mechanisms in providing timing-related information for the developer and should be able to maintain the QoS throughout the operation of the system[3]. QoS support acts as the contract of the system to guarantee the timing performance to the application, and in general, the QoS contract contains information on:

- Best Case Execution Time (BCET)

- Worst Case Execution Time (WCET)
- Best Case Communication Time (BCCT)
- Worst Case Communication Time (WCCT)

These bits of information are needed for the system to guarantee an end-to-end latency. They act as the constraints to the developer and the basis of the system specification.

4.1.4 Transparent to applications and system

To increase the transparency of the middleware, interfaces are used in constructing the distributed real-time embedded system. In this section, the two types of interface are introduced and how they can make the system transparent is discussed.

Interfacing between middleware and nodes

One of the functionalities of the middleware is to hide the heterogeneity of the distributed system. In a distributed system, each of the nodes is running their own version of subsystem and operating system. The coordination among them can be complicated. The middleware would become complex if it handles the operations for each of the individual operation system and allow them to cooperate with each other directly. Therefore, interfaces need to be implemented to allow different operating system have the same mechanism in using the middleware and hide the complexity of the system[10].

In addition, the interface increases the usability of the middleware. The middleware software can be reused in different systems by implementing an interface to the hardware of the new systems. This eliminates the problem of reconstructing of middleware and reduces the time-to-market of the system. Having an interface also allow a higher flexibility and scalability of the middleware.

This type of interface allows a same piece of middleware to take control of the different systems that run on different nodes. The interface acts as the adapter of the nodes, which means that with this layer of the interface, the method that the middleware uses to handle each of the nodes would be the same. The interface act as a translating tool and provide access for the middleware to control the resources throughout the network.

Application programming interface

Application programming interface (API) is the interface between the middleware and the application, it simplifies the complexity of the system for the user applications. API provides services on the system to the applications and the applications do not need to care about the implementation behind each operation[10]. For example, if an application needs to find the target in communication, what it needs to

do is to use a command that is specified in the API. How the API triggers the operation in middleware and how the middleware locates the target does not concern the application.

API usually provides a set of instructions for the application to access the resources in the network. Through the API, the application can register the services that it provides to the network and use the services that are provided by other components and applications. User applications need not to be programmed according to the complexity of the middleware. Instead, they use the API to trigger the operation of the middleware. This implies that the user applications can be easily ported to other system without large portion of change of code.

4.1.5 Fault-tolerant

Middleware should be able to provide mechanism that allows the system to run and hide the failure in case of some of the components are failing. For a distributed system, the numbers of components and nodes are generally large. A component failure can easily occur in the system. The system should not sacrifice all the operations just because of a single operation done by the failing component. A successful middleware should cope with the system and avoid single point of failure in the system. General approach in tackling the fault-tolerant is to use replications of the component.

4.1.6 Consistency control

A distributed system consists of multiple nodes concurrently running in the system, the resources are shared and can be accessed by more than one component. The concurrency access of the resources can be problematic if it is not controlled. In results, lost update and inconsistent analysis can occur[8]. Lost update means that the update done with an operation is overwritten by another and results in logical error. Inconsistent analysis means that the value of a component read by another component may not be most updated when there is another component at the same time writing to the component. Therefore, there is a need of adopting protocol to deal with the concurrency problem and thus increases the reliability of the system.

4.1.7 Other coordinating units

Apart from the functionalities that are listed above, there are other functionalities that are needed within the middleware to coordinate the interaction of the systems, like locating the receiving component, clock synchronization among the network, tracing and recording the status of the system, etc.

Locating the receiving component

In order to provide user applications the access of the remote component within the system while maintaining the transparency of accessibility, mechanisms in finding

and addressing the target component should be provided. Usually a naming unit or a trading unit is used. Naming is to search the target component by name or ID and trading is finding of targeting component by specifying the functionalities and the QoS of the component[11].

Clock synchronization

Clock synchronization is needed to support the timing performance of a real-time system. Since the distributed system consists of individual nodes, each having the own clock, the "time" of the system is not commonly understood by all the components without the help of clock synchronization. Therefore there is a need of clock synchronization.

Tracing the status of the system

Tracing of a system is usually not for functional consideration. The tracing of the system is usually used for debugging or checking the behavior of the system during designing. However, with the increasing demand on QoS in a real-time system, tracing is also used for the specification of the timing performance of the system. Tracing can be done by internal or by extending the middleware with tracing tools.

4.2 Related work: TAO

The Adaptive Communication Environment ORB (TAO)[12] is an open source implementation of Real-time Common Object Request Architecture (RT CORBA)[13]. It is aiming at providing a middleware for a distributed real-time system. As in RT CORBA, it provides some degree in controlling the resources to provide a better QoS and, therefore, a better real-time performance. It uses an object-oriented approach meaning that the functionalities are provided by objects and the components inside the system can be able to use the functionalities.

4.2.1 Object-oriented middleware

Object-oriented approach allows the functionalities of an object to be seen by the rest of the network. The object registers its functionalities and variables that are allowed to be call and read[14]. If a client wants to use the functionalities, it has to create an interface object that has the same type of the interface provided by the object, and then bind it to locate the server that contains the object. The functions and parameters can then be used and read through the bounded reference.

This approach allows the client to use the object like a local object within the client. And the programmer can write the code in a usual way without need to know where to find the object as the binding is done by the middleware. It also provide the system a high portability and adding new component to the system can be done easily.

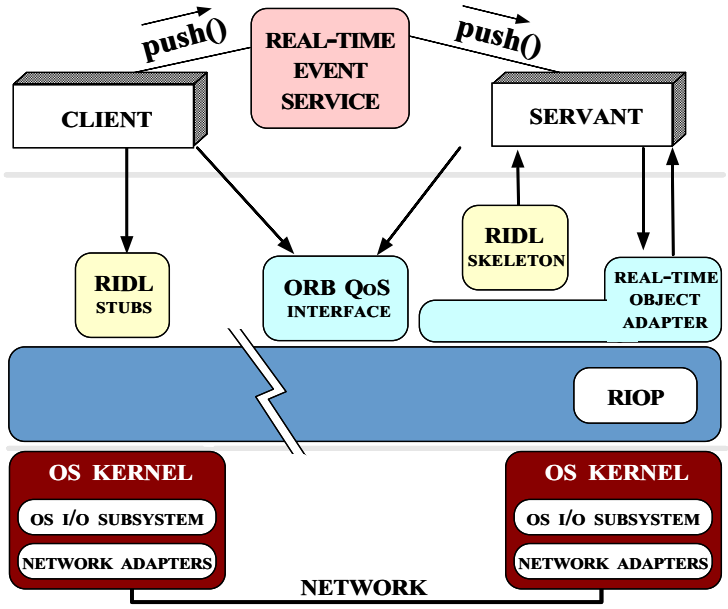


Figure 4.3. TAO Structure[15]

4.2.2 TAO Structure

The structure of the TAO CORBA system is shown in figure 4.3. As shown in this figure, the ORB core of each nodes work together to form a platform. The client core can then see the object in the server core. An object needs to provide the interface for client to use it. In CORBA, the interface provided is called "skeleton", which is like a header such that the client can follow the format and use the object by filling in the content "Stubs". TAO does not give a uniform system. The operating system is not wrapped and acts like any other service.

The object adapter shown in the figure is used to receive the requests sent by clients. It provides the mechanism for the communication between the client and object. The details are discussed in the section 4.2.3. The ORB cores are running on top of the operating system.

4.2.3 Communication in TAO

The communication between client and object is done in the ORB core. In order to send a message, the client needs to know where the object is. The ORB core uses name, functionality or specific QoS to find an object for the transaction. The ORB core then connects to the server, the object provider, using a socket-based connection and allocates memory for transaction. After the connection is done, it would send the parameter from the client to the server[16]. Note that since the data representation in the client end and the server end may be different, the ORB core

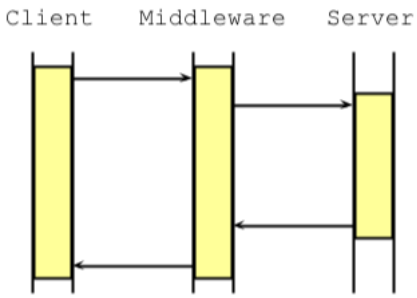


Figure 4.4. Middle-tier Server Blocking Protocol

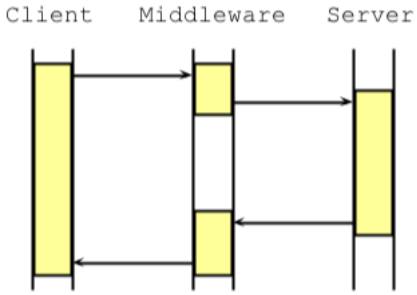


Figure 4.5. Asynchronous Method Handling Synchronization Protocol

would handle the conflict and translate the data to the format used by the targeting object.

At the server end, the object adapter allocates memory and receives the message and it would locate the object that handles the functionality. It passes the message to the object and locates the client that sends the request for the reply message. In addition, it maintains the concurrency of the server core. After the transaction is done, the memory is then de-allocated by the object adapter.

In TAO, the broadcasting and group request are supported.

4.2.4 Synchronization Protocol in TAO

In CORBA implementation, the synchronization protocol is done with a middle-tier server blocking protocol as shown in Figure 4.4[17]. The step of such synchronization protocol is that the client application sends a request to the middleware. The middleware then re-direct the request to the server that handles the operation of the request. When the operation is done, the server returns the data to the middleware and then to the client application. This method is said to be a synchronous handling method and each of the server thread can handle one communication in this protocol.

M. Desgoande et al introduced asynchronous method handling synchronization protocol in TAO[18]. The motivation of this approach is to allow more concurrent tasks that can be handled by the same thread instead of handling only one like

the conventional CORBA implementation does. The approach of this protocol is illustrated in Figure 4.5. The step in this protocol is that the client application sends a request to the middleware. The middleware then stores the information of the request in a handler and re-directs the operation to the server and at the same time takes back the control. The server then processes the operation and returns the results to the middleware. The middleware uses the information stored in the handler and returns the results to the client application.

From the simulation of [18], it shows that using the asynchronous method handling protocol, the middleware can maintain a higher throughput than a normal CORBA implementation in heavy load and when the blocking time is increased, asynchronous method handling gives a higher throughput then conventional implementation.

4.2.5 QoS support in TAO

Static scheduling suffers from disadvantage in handling aperiodic task and limited guaranteed utilization of the system, therefore, C. Gill, D. Levine and D. Schmidt[19] implement a dynamic scheduling service in TAO to resolve the problem. The scheduling service supports Rate Monotonic Scheduling (RMS), Earliest Deadline First (EDF), Minimum Laxity First (MLF) and Maximum Urgency First (MUF) scheduling scheme.

The service is divided into static part and dynamic part. In the static part, the scheduling service first tries to construct a feasible schedule by analyzing the QoS information that the application specified and then assigns the priorities to the tasks. These configurations are then supplied to the middleware core for the dispatching in runtime. In the dynamic part, the core configures the dynamic queue based on the queue configuration set by static part in runtime.

The simulation results from [19] shows that such service allows a more dynamic scheduling and with an acceptable overhead. However, the configuration under EDF and MUL is efficient when the load is moderate but not in heavy load.

4.2.6 Interface in TAO

The application interface in TAO follows the Interface Definition Language (IDL) in CORBA specification[20]. This approach aims to allow the objects and applications to have a high flexibility in the language they can be written and allow a unified standard so that the middleware core can understand the objects and the applications.

IDL is a strongly typed language that creates the interface to encapsulate the implementation and the data structure of an object in CORBA architecture. The type of the interface is defined by the IDL and this interface provides information on the operations that the object can handle and the variable within the object that can be read publicly. This interface type is independent to the language that the object is written.

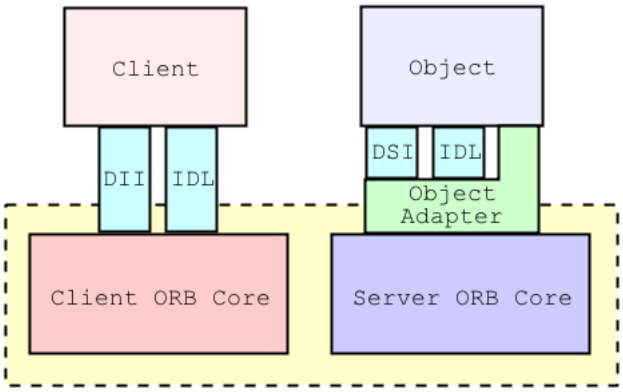


Figure 4.6. Structure of the CORBA Interface

The object in CORBA architecture is defined with a type that is the same as the interface type. The object provides the implementation of the operations listed in the interface. A more detailed structure of the CORBA interface is shown in Figure 4.6. When the client want to trigger an operation of the object, it would use the Dynamic Invocation Interface (DII) to invoke an object if the type of interface of object is unknown prior runtime, or use the IDL stub that is corresponding to the interface type that the object is defined. The object receives the request through the interface skeleton or the Dynamic Skeleton Interface (DSI) and provides the implementation with the support of object adapter.

The object adapter is the interface layer between the object and the core of the CORBA. It is a set of standard that guides how the object should look like and, therefore, can attach to the CORBA. IDL has been mapped to commonly-used programming language including C, C++, Java, ADA, Python, etc. This allows the object to be written in different languages and provide a higher heterogeneity to the system[21]. In addition, A. Gokhale and D. Schmidt suggests using active demultiplexing in locating the operation instead of traditional way of layered demultiplexing[22] to lower the overhead of going through stub interface and skeleton interface creates a large overhead, especially when going through dynamic skeleton interface and dynamic invocation interface, due to the data copying[23].

4.2.7 Fault-tolerant in TAO

TAO adopt the Fault Tolerant (FT) CORBA service[24]. This service provides measures in supporting fault-tolerant, which includes replication management and fault management. The objective of this service is to provide replica to server objects so that a failure in a core can be recovered by using the replica of the objects of that core to continue with the operation.

The concept of this service is that by using replication management, the replicas of an objects can maintain consistency and by using fault management, the system

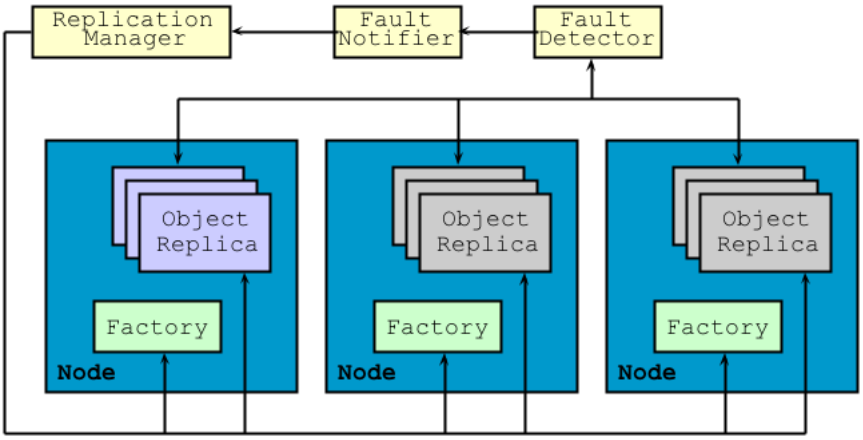


Figure 4.7. Infrastructure of the Fault-Tolerant Service

can keep track of whether of not there is fault occurred in the system[25]. The infrastructure of the fault-tolerant service is shown in Figure 4.7.

A request to the replication manager on replicating an object can be made by an application. The replication manager reads the request and the fault tolerance properties of the object and invokes factories of the nodes to create the object replicas. The fault tolerant properties include replication style, consistency style, factories, etc. FT CORBA maintains a strong replica consistency[26], which means that the states of the group of replicas should have be the same at the end of a method or a state transfer, depending on its replication style.

The fault detection is done by fault detectors. A fault detector acts like a watchdog and monitors a replica of an object by periodically sending request to the replica. If the replica fails to respond to the fault detector, the fault detector would then report to the fault notifier and the resource manager can add a new replica or invoke a replica to take over the job.

The specification[25] states the limitation of the FT CORBA that the replication of an object can only be done within a common infrastructure and FT CORBA requires the objects to have a deterministic behavior to achieve fault-tolerant.

4.2.8 Consistency control in TAO

The consistency in TAO follows the concurrency control service[27] from CORBA. The concurrency control is a hierarchical lock based protocol that guard the resources. Hierarchical locking is a protocol that aiming to solve the problem that cannot be solved by the two-phase locking protocol[28]. The idea of hierarchical locking is to lock all the resources that are within a composite resource at a single time using a single lock. Since some of the resources within the composite resource can be locked individually, this locking protocol introduces three other locking operation, which are intention read, intention write and upgrade.

Table 4.1. Resolution Table for Hierarchical Locking in TAO

Request	Intention Read	Read	Upgrade	Intention Write	Write
Intention Read	✔	✔	✔	✔	✖
Read	✔	✔	✔	✖	✖
Upgrade	✔	✔	✖	✖	✖
Intention Write	✔	✖	✖	✔	✖
Write	✖	✖	✖	✖	✖

Intention read and intention write means that there is reading and writing operation *about* to take action to the resources within the composite resource, and upgrade are used if a component need to do read operation to the resource and then wants to upgrade from reading to writing later. Table 4.1 is the resolution table of hierarchical locking protocol. In this table, each row represents the current state of the lock and the column represents the operation required. Intension write and intension read are compatible with each other since they are not the actual read and write operation, they are just trying to acquire the lock. Same idea applies to intention write and intention write. Two upgrades are not compatible, because this would lead to a deadlock, and it is the same reason of why intention write and write are not compatible with upgrade.

In [29], N. Sharifimehr and S. Sadaoui point out that the concurrency control service of CORBA are not fault-tolerant in acquiring the lock and the service cannot deal with deadlock situation.

4.2.9 Additional feature in TAO

The concurrency model used in TAO is to use a leader/followers thread pool server concurrency model[30] compared to a server queue-based worker thread pool concurrency model that is used in CORBA standard. The idea of the model that TAO used is illustrated in Figure 4.9. This model is motivated by the reasons that the model used by the CORBA standard increases locking overhead due to context switch and synchronization and can result in unbounded blocking due to priority inversion[31].

Instead of using a I/O thread that selects and reads the requests and passes the requests to a queue and which is handled by some worker threads which is used in server queue-based worker thread pool as shown in Figure 4.8, in this model, every threads can be a leader thread or a followers thread. The leader thread selects and reads a request from the thread queues and then gives up the identity of leader and

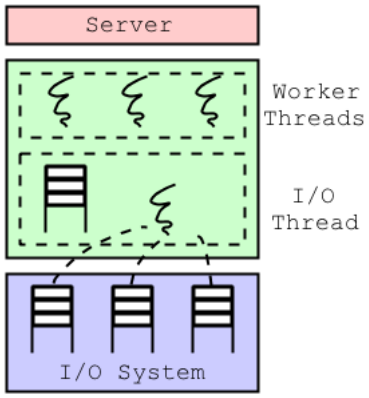


Figure 4.8. Server Queue-Based Worker Thread Pool Server

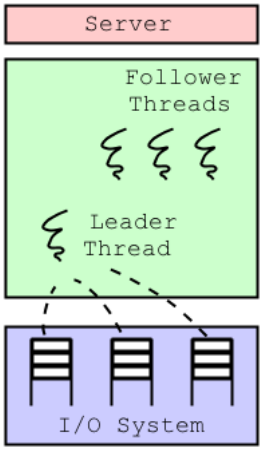


Figure 4.9. Leader/Followers Thread Pool Server

handles the communication with the server of the request. A follower thread can then becomes a leader and read from the thread queues.

This model is implemented and simulated in [30]. The results shows that the performance of this model in handling application workload is around 8% compared with the concurrency model that is used by original CORBA standard.

4.3 Related work: AUTOSAR

Automotive Open System Architecture (AUTOSAR)[32] is an architecture standard used in automobile. In this section, the Run-time Environment (RTE), which is the middleware of AUTOSAR, is discussed.

AUTOSAR is an architecture that is commonly used in the automobile industry. It tackles the embedded constraints, distributed constraints and real-time con-

straints. Instead of building a universal RTE to be used for any system, AUTOSAR provide a RTE generator that generate a middleware for the system by specifying what runnable entities there is in the system and the hardware connection of each of the Engine Control Unit (ECU).

4.3.1 Component-based middleware

AUTOSAR is a component-based middleware. A component-based middleware groups the functionalities and the data into components such that within a component, the data and the functionalities are highly interdependent. The components have low dependencies with each other. Components are usually reusable in systems as they are usually encapsulated in specific styles[33].

Accessing and communicating to a component is done through an interface. The implementation of the functionalities of a component is transparent to the middleware and other components in the network. This approach increases the reusability of the components and the operations of the system are more intuitive as it is well structured and the heterogeneity can be easily hidden[34].

4.3.2 AUTOSAR Structure

The architecture of AUTOSAR is shown in figure 4.10. In the figure, it can be seen that there are three parts within an AUTOSAR system, which are AUTOSAR software, basic software and AUTOSAR middleware. The AUTOSAR middleware is done by the RTE with the help of interfaces. The software components are classified by their nature into two categories.

AUTOSAR software

AUTOSAR software, which is also known as software component is the application software that running on the system. The software components are accessible throughout the network. It can be further divided into two categories, application software and actuator/sensor software. An application software component is hardware independent and can be relocated on any of the ECU in the system without change of code. Actuator/sensor software, however, is needed to stay with the same ECU. This is because of this type of software component can only interact with the hardware locally within the ECU. If it needs to access to hardware on other ECU node, it has to be done with the help of intermediate software component.

Basic software

Basic software is hardware dependent and can never be relocated to another node. a basic software can interact with the hardware that is locally located through the ECU abstraction layer. Some of the basic software, like operating system, services and communication are fixed, which means that they require a interface to become usable in the node as each node has different hardware component.

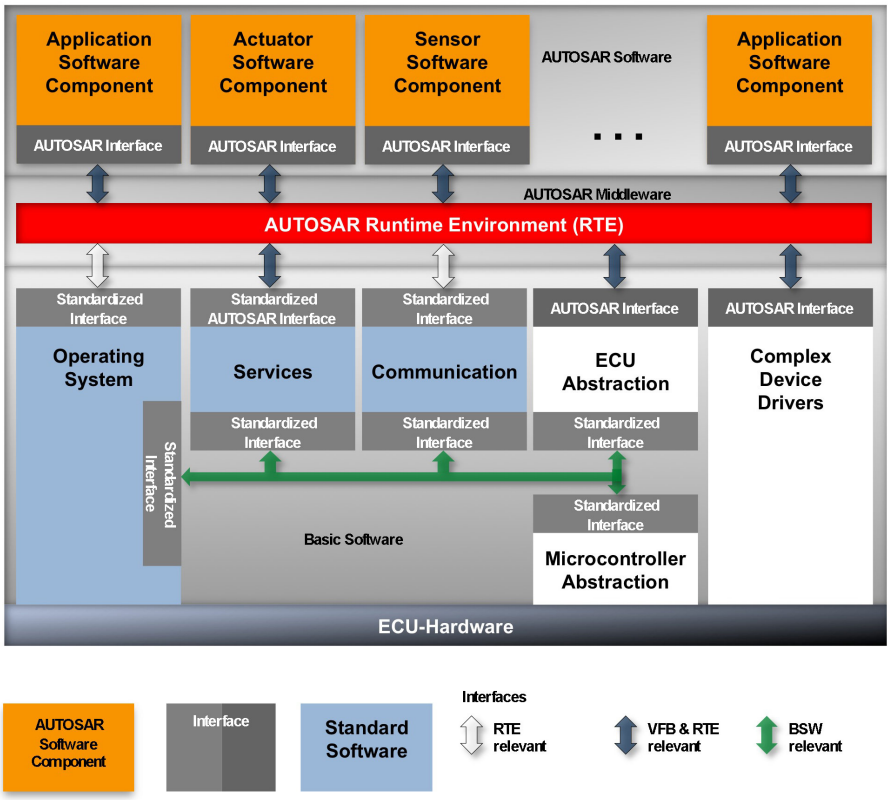


Figure 4.10. AUTOSAR Structure[32]

4.3.3 Communication in AUTOSAR

All communications that involve software component is handled by the RTE. A software component can be communicating with another software component located within the nodes or in the global network. But a software component can only access to the local basic software that has ports and runnable entities. The communication can be intra-task, inter-task, inter-partition or inter-ECU[32]. The higher level of communication may involve context switch or affect memory protection. In inter-ECU communication, it may even be unreliable. The communications need to be done through ports, which is static and connected through the RTE. Communication can be categorized into sender-receiver type or client-server type. The AUTOSAR RTE from all the nodes works together and forms a Virtual Functional Bus (VFB) that handles the communications.

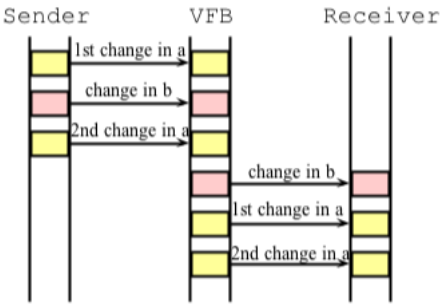


Figure 4.11. Sender-Receiver Communication

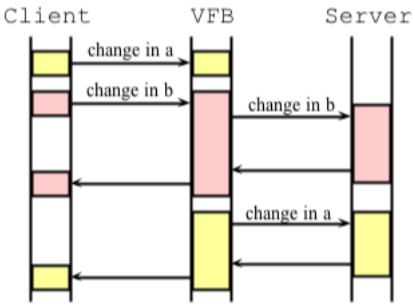


Figure 4.12. Client-Server Communication

4.3.4 Synchronization Protocol in AUTOSAR

The sender-receiver interface of AUTOSAR aims to allow a software component to transmit or broadcast data or information. This type of communication acts like a message passing communication. It supports both one-to-many and many-to-one communication[32]. The approach of this protocol is illustrated in Figure 4.11. In this graph, different colors represent the instance of communication of different data. From the figure, it can be seen that the AUTOSAR VFB is asynchronously communicating with the sender and the receiver and can receive concurrent requests from the sender instead of handling one data transfer at a time. D. Wang et al.[35] point out that the VFB reserves the history order of a data but not the order of requests. J. Hyun et al.[36] also state that AUTOSAR does not guarantee that the messages sent in many-to-one communication are distributed at the same time, and it does not guarantee that the messages received at the same time by receivers in one-to-many communication.

The client-server interface is aiming to allow software component to call for remote operation that is provided by another software component. Client-server communication acts like a function call. Unlike sender-receiver, this protocol supports only many-to-one communication[32], many different clients can call meaning a function at the same time. An illustration of this protocol is shown in Figure 4.12.

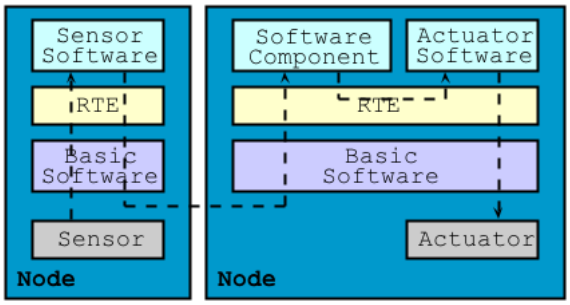


Figure 4.13. Data Flow Example

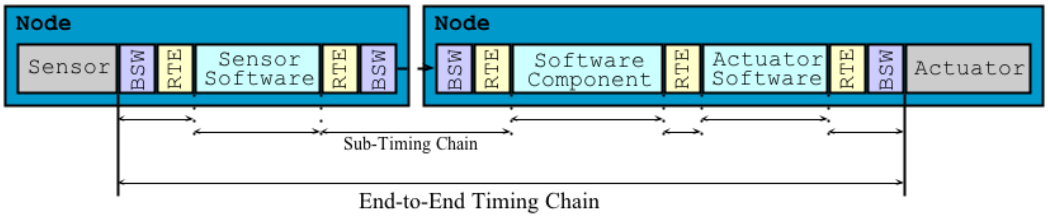


Figure 4.14. Timing Chain Example

It can be seen that this protocol requires AUTOSAR VFB to have synchronous communication with the server, but not the client. And as stated in [35], it does not reserve the order of the operation.

4.3.5 QoS support in AUTOSAR

R.Ernst[37] state that AUTOSAR use mainly client-server communication, which would create a hidden timing dependencies. Figure 4.13 shows the example of data flow in a sensing to actuating operation. The data flows from a sensor to a sensor software component, then to the software component that handles the operation and finally to the actuator software component and the related actuator. Such data flow can be seen easily in automobile, like sensing of brake paddle (sensor) to control the brake (actuator).

The timing chain of the example can be unfolded as shown in Figure 4.14. It is seen that the end-to-end timing chain is composited with different sub-timing chain. It can result in a low predictability of the timing, and thus a poor guarantee of QoS. To address this problem, AUTOSAR introduced timing extensions[38] in AUTOSAR 4.0 that provides timing models to allow specifying the timing constrain of the system in the start of the project, and the timing specification then act as timing requirement for the developer. AUTOSAR can then guarantee the real-time performance.

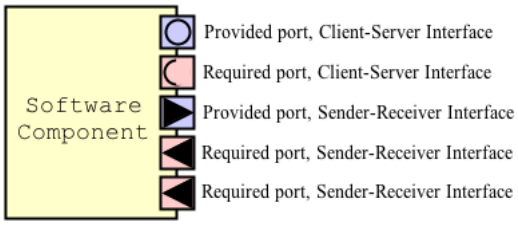


Figure 4.15. Example of Connection Ports of a Software Component

4.3.6 Interface in AUTOSAR

To separate the infrastructure of the system and the software application, AUTOSAR uses software component to construct the application. It is aiming to increase the independent of the software to the structure of the system[32]. The software component has to be designed based on the software component template[39].

A software component template encapsulates the software component and it provides standard port prototypes for the software component to connect to the AUTOSAR RTE and other software component. The connections are further supported through interfaces to specify the type. A software component with the connection ports is shown in Figure 4.15. Basically, there are two basic types of the port prototypes for the software component, which handles the provided communication and required communication respectively, and there are interfaces for either client-server communication or send-receiver communication to specify the type of the communication that can be done in the port.

Each software component is defined with a type. The type specifies the port setting of a software component. Developer can make use of the types or define own types to encapsulate the software component and use as a reference for other components that it communicates with.

The AUTOSAR has been mapped to support C, C++ and Java at the moment[40], and a software component is transparent to the type of the ECU, and the location of the component software that it communicates with[39].

4.3.7 Fault-tolerant in AUTOSAR

AUTOSAR uses a redundant activation approach to guarantee fault-tolerant[41]. Figure 4.16 shows the mechanism of the redundant activation. It is basically a voting scheme that monitors the results of replicated software running on different ECU, the end result is the majority of the common value. Since the change for more than half of the software components fail are low, the results are expected to be right.

This approach can act as the monitoring of the status of the component too. When there is a software component not agreeing to the end result, the system can isolate that component and do correction on it.

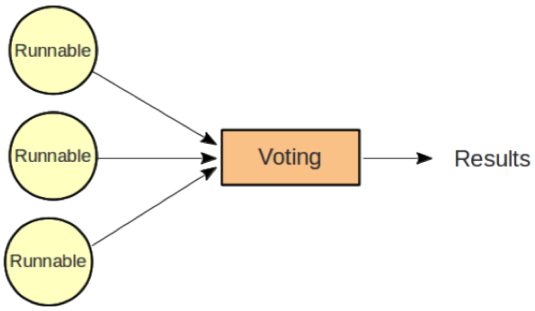


Figure 4.16. Redundant Activation

Besides that, J. Kim et al.[42] implement a fault-tolerance algorithm into AUTOSAR. The software component template is reconstructed to include an Automotive Safety Integrity Level (ASIL) that is used to state the degree of safety that the software component should be maintained. This ASIL determines the replication policy used for the component.

The software component, including the runnable entity as a whole, is replicated. The tasks that run on the ECU that is corresponding to the software component need also to be duplicated to the ECU that the replica resides. The replica of the component would maintain the data and state synchronized using a cold standby approach, meaning that the replica of the component is powered off with only the memory part is active to achieve data consistency.

In additional to that, a health statues module is added to the ECU to report the status of the ECU to the system. The health statues module report any failure of ECU and AUTOSAR can then activate the replica of the runnable in the software component.

Such implementation is tested and simulated in [42] that the replicas can save more than 60% of processors and, therefore, give a satisfying reliability and fault-tolerance to AUTOSAR.

4.3.8 Consistency control in AUTOSAR

Data consistency mechanisms in AUTOSAR can be derived from the software component template[39]. There are many strategies to maintain the consistency of data suggested in [32] that can be derived in AUTOSAR, including sequential scheduling, interrupt blocking, copy strategy, cooperative runnable placement etc. Here, the cooperative runnable placement strategy is discussed.

Cooperative runnable placement strategy is to allow a runnable to specify the cooperative restriction, such that a task containing the runnable that is marked as protected under this strategy cannot preempt another task that has a protected runnable. But all other tasks without protected runnable can preempt them as in a normal priority scheduling algorithm. In short, in this strategy, the tasks are executed according to the priority, but tasks that are having runnables that are

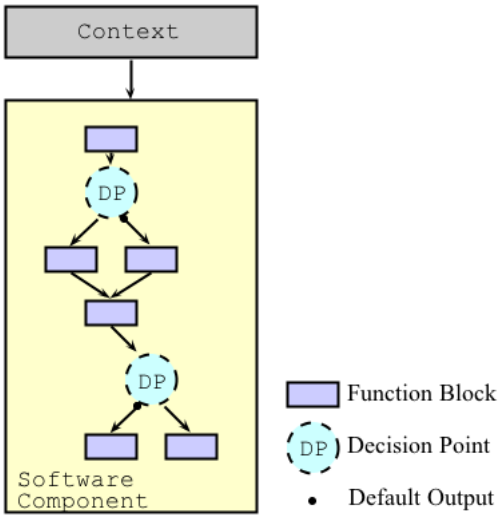


Figure 4.17. Example of a Software Component under Policy-Based Configuaration[43]

marked protected cannot preempt each other.

4.3.9 Additional feature in AUTOSAR

Unlike most of the middleware, Run-time Environment (RTE), which is the middleware of AUTOSAR is specific for the system, and cannot be reused in system that have different structure and software components. Which mean that the RTE has to be generated every times the system change. To address this problem, AUTOSAR provides a RTE generator [32]. The developer has to specify the system and ECU configuration, constrains, software component, hardware, etc., and the generator would then generate the RTE accordingly.

4.4 Related work: DySCAS

DySCAS[43] is an autonomic middleware framework targeting to automotive system. It uses policy-based configurations and provides high degree of flexibility and reconfigurability to the system.

4.4.1 Policy-based middleware

Policy-based configuration aims to provide a versatile system that provides dynamic reconfiguring according to the context in the system. The behavior of the software component in the system can be changed during runtime. Figure 4.17 shows an example of how a component can be constructed under policy-based configuration[44].

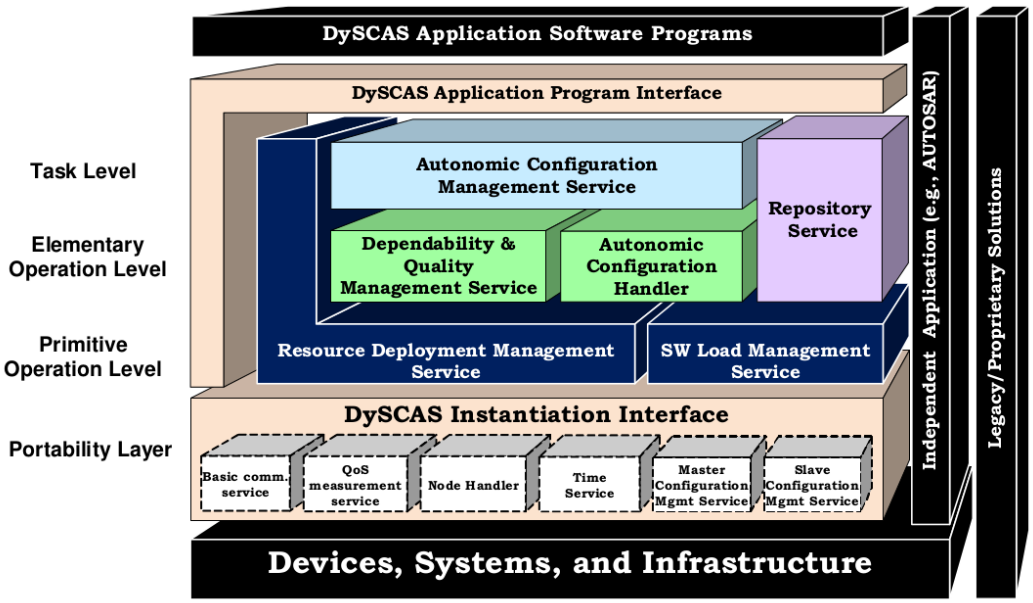


Figure 4.18. DySCAS Structure[43]

From the figure, it can be seen that two basic blocks, which are static function block and decision point, compose each software component. A software component uses the status of the environment and resources to make decisions. In each of the decision point, the context act as the variants that determine which branch the component should execute next according to the decision. In policy-based configuration, software components are constructed in static structures. The function blocks are static, but the policy in the decision point can be dynamically changed, making a higher flexibility to the system. To fault-prove the system, a default control route can be set in the software component, such that in case of failure, the software execute the default control route and return to default outcome.

4.4.2 DySCAS Structure

The structure of the DySCAS structure is shown in Figure 4.18[43]. The middleware is consists of two categories of services, which is core services and interface services. Core service is the six service blocks that operate in task level, elementary operation level and primitive operation level. They are the backbone of policy-based configuration and QoS support. The services in the portability layer are interface services. They are not mandatory, but they support the interaction for the system to the hardware structure.

The task level in DySCAS handles the request of configuration change. In this level, the impact to the system is evaluated, and if the change is valid, it would create a scheme and configure the software component. Services in elementary

operation level monitor the QoS of the system and determine if there is a need of configuration. And in primitive operation level, the services allocate the resources for supporting the monitoring and configuring and they also receive the feedback from the hardware and the application and report as exception.

In addition, DySCAS can run side-by-side with other independent applications and legacy solutions and they can access to the hardware structure directly without passing through DySCAS.

4.4.3 Communication in DySCAS

The communication of DySCAS is done by LINX[45], which is a communicating protocol that is independent to the hardware, operating system and the network connection of the system. In LINX, the communication is done directly from the sender to the receiver. Currently, LINX supports the communication protocol of Ethernet, TCP/IP for communication across machine and, in addition, shared memory across CPU within a node.

In LINX, the connections are dynamically created during runtime. LINX hunt for the targeting communication by the Communicator ID (CID) and the name of the communicator to establish connection. The receiver can filter the message that it wants to receive by specifying the message ID that it intends to receive in a specific time. The actions on handling the received message can be determined by the type of the signal in the message. M. Christofferson[46] states that this allows the receiver to be designed in a finite state-machine approach.

4.4.4 Synchronization Protocol in DySCAS

LINX protocol supports a non-blocking send and both blocking and non-blocking receive[45], which means that the system support asynchronous send and receive and non-blocking send and receive. The non-blocking send and receive protocol does not guarantee the message is sent, but it can avoid deadlock or blocking due to waiting for the receiving end. LINX is a direct message-passing tool, it does not go through middleware and therefore protocols like the middle-tier server blocking protocol that is used in TAO cannot be implemented directly. However, the direct point-to-point communication this can reduce the latency of the communication.

4.4.5 QoS support in DySCAS

The Quality of Service (QoS) of the DySCAS is supported by feedback of Dependability and Quality Management Service (DQMS)[44]. The idea of this approach is to choose the QoS level and to migrate the function according to the statues of the resources. DQMS are distributed over the network, each of the node have a local DQMS that can handle the optimization of the resources in that node. When the local DQMS fails to allocate resources to a task, it would file a load balance request to the global DQMS in the master node. The global DQMS can choose to relocate the task or redeploy the task to the same node.

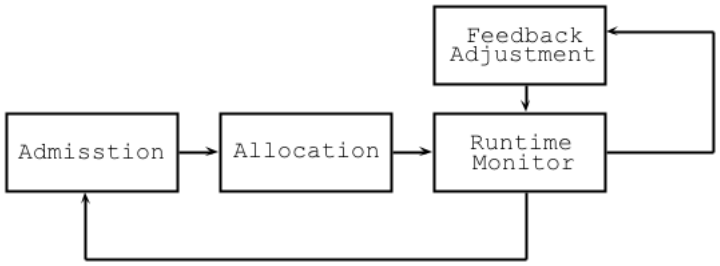


Figure 4.19. Operation Flow of DQMS

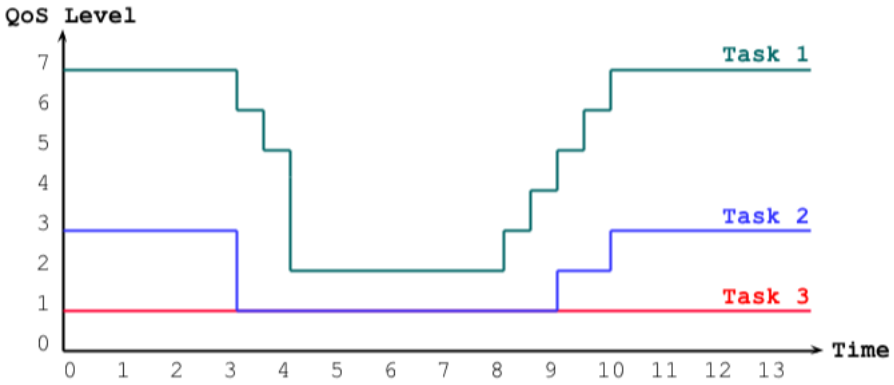


Figure 4.20. Example of Change in QoS Levels

DQMS can dynamically manage the resources to applications. The tasks need to be able to run in different levels of QoS in order to be supported. The local DQMS uses a feed forward loop to measure the runtime resource usage and thus adjust the QoS[43]. The operation flow of the DQMS is shown in Figure 4.19. The admission control decides if a feasible schedule is possible if there is a task. DQMS then either rejects the task or chooses the optimal allocation according to the current resources usage of the node. It keeps monitoring the usage and controls the QoS of the running tasks by a feedback adjustment. An example of the QoS level for tasks running on the system is shown in Figure 4.20. It can be seen that some tasks are running in a higher QoS comparing to other tasks. This decision is made by the importance of the task to the system. When a failure occurs, like at the 3rd second in the Figure, the DQMS would lower the level of QoS of tasks to an utilization that is affordable by the node. When the system recovers at the 8th second, DQMS then assigns a higher QoS to the tasks and back to the usual operation.

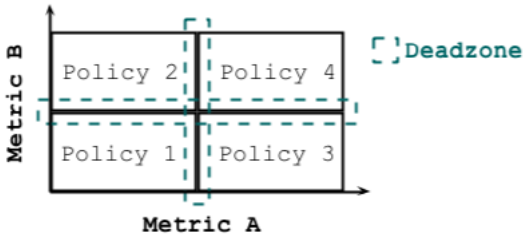


Figure 4.21. Self-Adaptive Policy

4.4.6 Interface in DySCAS

The policies in DySCAS are written in Agile policy expression language[47]. The idea of Agile is to put the decision of the behavior of the system to as late as possible. This is the backbone of the policy-based configuration. Agile policy script is written in an XML format. It supports self-adaptive policy behavior, which means that policies with different configurations of the same type are handling the behavior of the application depending on the current behavior of the system.

An example on the self-adaptive policy behavior is shown in Figure 4.21[48]. The current behavior of the system is monitored by a meta-policy, and depending on the behavior, the meta-policy determines the policy to be used in the next time instance. Agile uses dead-zones that avoid the policy changing rapidly when the behavior is at the border of the thresholds. In this example, four different policies are possible, depending on the two conditions that it is monitoring.

R. Anthony[48] states that the self-adaptive policy behavior brings in great flexibility and a better control over a complex hierarchical system.

4.4.7 Fault-tolerant in DySCAS

The fault-tolerant in DySCAS is built on the self-adaptability of the system. As mentioned in the QoS support, DySCAS can dynamically relocate the tasks running on a node. The relocation of the tasks is done by a load balancer and a resource manager[43].

In case of failure, the component that detects failure will trigger an event to the load balancer. The load balancer would gather the information on the resources and then send a request to the resource manager. The resource manager would calculate all the possible schedules and return the information of the resources. The load balancer then decides how the tasks should be assigned to other nodes base on the information on schedule and the information on resources.

4.4.8 Consistency control in DySCAS

The concurrency control design in DySCAS shares the same concept as CORBA and TAO. For details, please refer to Section 4.2.8.

4.4.9 Additional feature in DySCAS

Because of the flexibility given by the policy-base configuration, DySCAS provides many autonomic features, including self-configuration, self-healing, self-optimization, self-protection and self-defining[44], which lower the need of involvement from users.

4.5 Summary of middleware architecture

The three middleware each have different focus. TAO is aiming at to provide a general solution to distributed system. It tries to balance all the different aspects of the middleware and to tackle all the areas. AUTOSAR is focusing on automobile platform. To increase the real-time performance of the system and to have a better control over hardware components, AUTOSAR generates a specific system structure for each individual system. DySCAS focuses on providing autonomous ability, and the architecture and the functionalities of the middleware are built on the flexibility the autonomous of the system.

Chapter 5

Discussions on the Middleware

5.1 Summary of architecture comparison

In the previous chapter, the three related architectures of middleware are studied. is compared . Table 5.1 gives a general summary of the architectures of the three middleware: TAO, AUTOSAR and DySCAS. In the following sections, the middleware are compared in different perspective. This chapter answers the second question in the thesis:

What different protocols and techniques can be used to develop a DRE system?

5.2 Embedded concerns

TAO supports both minimum CORBA[49] and TAO subsetting[50] that can effectively reduce the memory footprint for the embedded system. By minimum CORBA, some features can be removed from the specification. They include dynamic skeleton interface, dynamic invocation interface, repository and other none crucial functionalities. TAO optimized the IDL compiler and, with TAO subsetting, it is shown that the overall footprint is reduced by more than 10% and the compilation time is reduced by 50%[51]. The principle of the IDL compiler is to try actively finding common features and build a single binary for common ones, avoid heap memory allocation and breaks the source code into smaller pieces and uses static link of the object.

AUTOSAR does not originally optimized in size. Though it is supported by project COMPASS[52], which the project is a cooperation of Decomsys GmbH, University of Technology Vienna and University of Applied Sciences Technikum Wien. The project is aiming to extend the component-based architecture of AUTOSAR by applying the component paradigm to the middleware itself through defining standards[33]. This approach encapsulates and transforms the communications into predefined communication primitives and further unifies the system and can successfully reduce the memory footprint of the system by 30%[33].

DyLite is an implementation of DySCAS aim to reduce the footprint of the mid-

Table 5.1. Architecture of TAO, AUTOSAR and DySCAS

Aspect	**TAO**	**AUTOSAR**	**DySCAS**
Approach	Object-oriented	Component-based	Policy-based
Synchronization Protocol	Asynchronous Method Handling	Client-Server and Sender-Receiver	Asynchronous communication and non-blocking send and receive
QoS Support	Dynamic Scheduling	Timing Extension	DQMS
Interface	IDL	Software Component Template	Agile
Fault-tolerant	Replication Manager	Redundant activation, Automotive Safety Integrity Level	Self-Adaptability
Consistency Control	Hierarchical Locking	Cooperative Runnable Placement Strategy	Hierarchical Locking
Additional Features	Leader/Followers Thread Pool Server	RTE Generator	Autonomous

dleware. It further simplified the reconfiguration and Quality of Service (QoS)[44]. When there is an additional application to start, DyLite remove the applications by the order of increasing resources efficiency until there is sufficient resource. It then removes the applications that have dependencies with the applications removed in previous stage and finishes with assign the QoS level of the new application[53]. This method allows reconfiguration under limited memory and resources and is proved that the functionalities of QoS optimization, fault-tolerant and load balancing in DySCAS can be preserved.

5.3 Real-time performance

Dynamic scheduling in TAO allows a high flexibility in scheduling. Developer can choose different scheduling algorithms for scheduling of tasks. On top of that, the asynchronous method handling, the communication protocol that is used by TAO allows the middleware not to be blocked by the client, which gives a higher QoS to the system. In addition, the leader/followers thread pool approach used in TAO tackles the priority inversion problem that can cause missing of deadlines for higher priority tasks by reducing the locking overhead and the hierarchical locking can help avoiding deadlock comparing to simple locking mechanism. It can be seen that TAO have lots of measures to ensuring the real-time performance to be met.

AUTOSAR uses timing extension as the specification to the QoS and allow the end-to-end latency of the system to be more predictable and therefore provide a better QoS. Also, AUTOSAR supports priority ceiling protocol, which avoids priority inversion and deadlocks and gives a better real-time performance. However, AUTOSAR does not have an active approach in guaranteeing QoS. Comparing to AUTOSAR, TAO and DySCAS have more optimizations that can be done in runtime.

DySCAS has a good QoS control that is done by dependability and quality management service. It monitors on the resources available and manage the QoS and function migration accordingly. The real-time property is supported by load balancing such that the load is not exceeding the capability of the available resource. Like TAO, the hierarchical locking can avoid deadlock occurs. The QoS provided in a DySCAS is dynamic and can be adapted to different situations. The acceptance of tasks provides the basis of tackling hard real-time tasks.

5.4 Accessibility transparent

The objects in TAO are addressed as an object through the IDL stub and IDL skeleton. On top of that, TAO provides naming and trading service to let the applications to locate the remote object. The dynamic skeleton interface and dynamic invocation interface also allows the client to invoke the remote object without knowing the exact configuration of the object. All these give a good support on accessibility of the system.

AUTOSAR uses a flat design in both application software and the basic software. The communications and locating of components are done in a unified way. And the software components are all encapsulated by using the software component template, such that the components can interact with each other always in the same manner.

LINX in DySCAS provide some degree of accessibility transparency to the system. The users need to specify the IP address or the hardware address of the target node, LINX would then search through the network and locate target. Comparing the three, AUTOSAR and TAO give better accessibility transparencies, as DySCAS

is limited by the functionality provided by LINX and the communications need to be handled by the users.

5.5 Migratability transparent

In TAO, the migration of functionalities is done by disabling and enabling the object replicas, and the whole process is handled by replication manager. The object replicas are consistent under strong replica consistency, which the frequency of consistency can be configured by specifying the replication style and can be as frequent as every method done. This means that migrating from one node to another can be done in a fine-grained manner. In theory, any tasks in TAO can be migrated into any other nodes as long as there is a factory in that node that can create the object replicas.

The fault algorithm constructed by J.Kim et al. in AUTOSAR can also be used as a base for task migration. The replicas of the runnable entity synchronized in a cold standby approach, like in TAO, the replication policy in AUTOSAR can be specified. However, limited by the hardware, not all the AUTOSAR software can be migrated to another ECU since components like sensor/actuator software components are hardware-dependent, this gives a lower migratability.

DySCAS uses self-adaptability for load balancing and task migration. DySCAS actively monitor the status of the system and apply load balancing to the system, which requires low human interference to the system. On top of that, the QoS level of each tasks can be changed to adopt a new task, which makes the migration become more possible if it is needed.

Moreover, TAO, AUTOSAR and DySCAS are all having certain degree of migratability transparency. The applications need not to know or direct the migration. Task migration is done in the system without influencing other applications running in the system.

5.6 Concurrency transparent

TAO uses asynchronous method handling, such that the sender and the middleware itself would not be blocked when waiting for the results from the server. This protocol provides a better concurrency transparent, as the applications are unaware of the number of concurrent tasks running in the system. The request sent by a sender to the middleware would not be stalled and there is no need of waiting for the whole communication is completed to send another message. When there is conflicting access to common resources, TAO protected the resources by hierarchical locking which can reduce the overhead of locking a composite object, and therefore to resolve the conflict of two concurrent requests would be faster.

AUTOSAR uses client-server and sender-receiver approaches for communication. For client-server communication, the middleware need to communicate with the server synchronously. Which means that the tread handling the communication

need to wait for the reply from the server. In heavy load situation, requests from clients may be unable to be handled as the threads in middleware are handling other communications, which results in either lost packet or blocking of the client application. AUTOSAR uses cooperative runnable placement strategy to avoid the conflicts by specifying the protected resource of a task. This can preserve partially the priority of the task without sacrificing the consistency of the resource.

DySCAS uses LINX as communication tool. The communication does not involve the middleware, which in this way, can greatly reduce the overhead for communication and the traffic in middleware does not affect the performance of the communication. However, this leaves the responsibility to the programmer to take care of the synchronization between client and server. The capacity for the server to handle requested tasks is supported by creating buffer instead of by the middleware but the linking and connecting the communication sockets need to be taken care of by the programmer.

Among these three, TAO gives a better support for concurrency. The synchronization protocol is more sophisticated. DySCAS gives more flexibility to the user. However, it means that the involvement needed from the developer is higher.

5.7 Scalability transparent

All of the three middleware can be configured to suit on smaller systems as discussed in Section 5.2.

TAO optimized the communication throughput and adopted asynchronous method handling and demultiplexing in locating operations. The lower of these overheads can benefit the scalability of the system such that when the system scales, the applications would not be suffered from large overhead due to the summation of each of the overhead.

AUTOSAR is generally scalable to different sizes of the system. However, the synchronization protocol used in AUTOSAR somehow lower its scalability, as when the system scales, the communication can be blocked much more easily.

DySCAS bypasses the middleware in case of actual data transfer. As the handling of communication is done directly in client and in the server, if multiple clients request the server, the blocking time would increase greatly. In AUTOSAR and TAO, the communications are done through middleware, when the system scales up. There are more resources to handle the communications. However, in DySCAS, scaling up would only results in more traffic to the servers, the buffer can easily be full and the performance of the system would be deteriorated. So by comparison, TAO is the most optimized for scalability transparency.

5.8 Failure transparent

Replication manager, fault detector and fault notifier support the failure transparency of TAO. The fault detector actively checks the status of the object replicas

and report fault if there is any. The strong replication consistency can guarantee the object replica can take over the job in case of the original object fails without larger overhead to resume the job. In addition, as an object-oriented middleware, the failure of an object can notify the system and the nodes by exception[8]. So the client can be notified and take actions accordingly in case there is no resources for running certain operation.

AUTOSAR uses replication and redundancy activation for maintaining failure transparent. The safely level of a software component can be specified using ASIL and the system is monitored by a health statues modules.

In DySCAS, on top of replication, it uses self-adaptability for handling failure. DySCAS can dynamically change the level of QoS of a task during runtime. The system is monitored and the trends of availabilities of resources are tracked, and the configurations and the QoS levels of tasks can then be changed accordingly. This gives a higher flexibility to the fault-tolerant handling. The failure in resources can be handled by lowering the total QoS of the tasks running on the node instead of migrating to other node on every failure. The transition process is done seamlessly.

Comparing to task migration, the approach of lowering QoS level used by DySCAS is having a lower overhead. Also, for task migration in AUTOSAR and CORBA, if the remote nodes are busy and do not have available resources for the migration, the task needs to be aborted. However, with self-adaptability, the tasks work cooperatively to lower their QoS levels such that all the tasks can be running and important tasks can be allocated with more resources to meeting requirements. With this comparison, DySCAS performs better for failure transparency.

5.9 Reconfigurability

The application software for all TAO, AUTOSAR and DySCAS can easily be reused in other system running the same middleware. Despite of that, the middleware of AUTOSAR is constructed particularly for the system configuration, including the hardware and the software component. Which means that the middleware of AUTOSAR needs to be regenerated for every implementation. This is supported by the RTE generator of AUTOSAR.

TAO, in contrast, is independent to the hardware, and the middleware can be run on any system configuration. This reduces the time for configuration and the middleware can be used as it is. The dynamic invocation interface and dynamic skeleton interface allow some flexibility in putting the software application to be implemented in a latter stage of design. Moreover, the logic flow and implementation has to be fixed before the system is delivered.

DySCAS postpones the actual implementation to the latest stage, the system can specify the policy by changing the policy code in Agile policy configuring language. Which means that the user of the system can reconfigure the system on-site without reconstructing the whole system. And therefore the reconfigurability of DySCAS is the highest among the three.

5.10 Summary of discussions

Table 5.2 summarizes the performance of the middleware. In summary, TAO gives is having good accessibility, concurrency and scalability. The asynchronous method handling, hierarchical locking and dynamic scheduling are some of the good features in TAO. AUTOSAR perform well in real-time performance and accessibility but due to the communication protocol, it is not scalable and concurrency is not as good. The priority ceiling protocol, unified system and cooperative running placement strategy are useful in the system. For DySCAS, it is good in real-time performance, migratability, failure handling and reconfigurability. The autonomic system in it, like policy-based configuration, self-adaptability and self-configuration are the main features of DySCAS.

Table 5.2. Comparisons of TAO, AUTOSAR and DySCAS

Metric	**TAO**	**AUTOSAR**	**DySCAS**
Embedded concerns	Supported by setting	No specific measures, but supported by implementation in COMPASS project	Originally targeting for embedded system. SHAPE and DyLite implementations further reduce the footprint
Real-time performance	Hierarchical locking to prevent deadlock, dynamic scheduling	Time extension for guaranteeing QoS. Support priority ceiling protocol	Hierarchical locking to prevent deadlock, DQMS provide better QoS
Accessibility	Accessibility provided by IDL and naming and trading service	Accessibility by unified system	Limited accessibility by Linx
Migratability	Migratable with replication manager	Migratable with support of ASIL	Migratable with self-adaptability, multiple QoS level
Concurrency	Reduced overhead by hierarchical locking and asynchronous communication	Overhead for client-server communication. Resources are protected by cooperative runnable placement	Direct data transfer minimize overhead, programmer take care of synchronization
Scalability	Optimized for scalability	Scalable but limited by blocking communication in large system	Point-to-point communication maybe hard to handle in larger system
Failure handling	Task replication	Task replication and redundant activation	Self-adaptability, dynamic level of QoS
Reconfigurability	Reusable middleware and flexibility supported by DII and DSI	Not reusable and not reconfigurable	Reconfigurable by the changing of policy

Chapter 6

Specification for System Design

6.1 System specification

The goal of this project is to design the architecture of a distributed real-time embedded system. With the requirements listed in chapter 2 and the comparison done in chapter 5, the general approach in constructing the system can be defined as the blueprint of the system.

The system should address all the requirements stated in the introduction and the performance metrics of DRE system listed in chapter 3. However, not all of the metrics are as important for the design of the system. The focus of this project is to find a transparent solution, while maintaining reasonable degree of real-time, embedded and distributed performances.

6.1.1 Approach

The three approaches discussed, which are object-oriented, component-based and policy-based are excelled in different aspects. With the consideration of transparency in terms of portability and accessibility of the system and the simplicity of system construction, a component-based approach is more preferred in the system. The system should have functionalities encapsulated in components. This approach hides the divergence of the implementation and the communication within the network can be done in a unified manner.

Related requirements: *SW_REQ_5, SW_REQ_8*

6.1.2 Communication

Naming and trading mechanism should be available in the system to locate the targeting node to benefit the accessibility of the system. In addition, in DRE system, the communication is usually done by message passing, this allow the message to pass through network to systems that have different hardware configurations and operating systems. However, shared memory is a much faster way of communication, although it cannot be done across node. The middleware of the DRE system should

support both and depending on the two ends of the communication, the middleware should be able to redirect the communication using shared memory if it is possible, and the process should not need the interference from the developer to increase the transparency of the locality as well as allowing a high performance in local transmission.

Related requirement: *SW_REQ_3, SW_REQ_8*

6.1.3 Synchronization protocol

The asynchronous method handling protocol benefits the capacity of the middleware to handle the data transmission. As discussed in Section 5.7, this helps enhancing both the scalability and the real-time performance of the system. However, the point-to-point communication by LINX is having a very high performance. Therefore, the system should support both of the protocols and depending on the load on receiving node and choose the communication protocol, such that in case the receiving node is busy and the buffer queue is full, the middleware would act as the temporary storage of the data and wait until the receiving node is ready to do the transmission. In additional to that, the memory requirement on the server nodes can be reduced to handle the communications.

Related requirements: *SW_REQ_1, SW_REQ_2, SW_REQ_3, SW_REQ_4*

6.1.4 QoS support

The system should have a guarantee on the QoS. Dynamic scheduling and DQMS, which are used in TAO and DySCAS with respectively, provide a good direction on how the system should support the QoS. The system should be able to schedule the tasks dynamically and be adjusted according to the load of the nodes, but the scheduling algorithm needs not to be changed in real-time. The DQMS used in DySCAS requires a support of policy-based configuration and autonomic properties of the system. However, the idea of multiple levels of QoS for handling excessive load is very promising. The system should be able to adjust the QoS levels of the tasks in case of task migrating.

Related requirements: *SW_REQ_1, SW_REQ_2, SW_REQ_6, SW_REQ_7*

6.1.5 Fault-tolerant

The replication manager of TAO can be reused for providing fault-tolerant, combining with the cold standby approach used in AUTOSAR to save energy and therefore reduces the power consumption in the concerns of embedded system. The need of replication should be able to be specified when the component start and the replicas of the system should be maintained with strong replica consistency, and the failure should be detected and reported automatically without need of user interference.

Related requirement: *SW_REQ_6*

Table 6.1. System Specification

	Specification	**Addressed Requirements**
Approach	Component-based	SW_REQ_5 SW_REQ_8
Communication	Naming & Trading units, Support local communication in shared memory	SW_REQ_3 SW_REQ_8
Synchronization Protocol	Asynchronous method handling Point-to-point non-blocking communication	SW_REQ_1 SW_REQ_2 SW_REQ_3 SW_REQ_4
QoS Support	Dynamic scheduling Task migration depends on load Multiple QoS levels	SW_REQ_1 SW_REQ_2 SW_REQ_6 SW_REQ_7
Fault-tolerant	Strong replica consistency Fault detectors Cold standby	SW_REQ_3 SW_REQ_6
Consistency Control	Hierarchical Locking	SW_REQ_9

6.1.6 Consistency Control

To achieve the reliability, the system should be able to use the hierarchical locking mechanisms by TAO and DySCAS. The common resources should be able to main consistent without lost update. Although the cooperative runnable placement strategy by AUTOSAR is useful and has a better control over deadlock, it would greatly increase the complexity of the schedule and therefore is not preferred in the design.

Related requirement: *SW_REQ_9*

6.2 Summary of the system specification

The system specification is summarized in Table 6.1.

Chapter 7

Design of the middleware

7.1 Overview

This work designs a distributed system with a middleware based on the results is concluded from the academic study in chapter 6. The design is shown in figure 7.1. This design uses a component-based approach. Each of the components is wrapped by an interface and can communicate with other components through that. Depending on the type of the component, the interface used by the object can be different. The components are categorized into three types: system object, wrapper object and task object, and their corresponding interfaces are system interface, wrapper interface and task interface.

As shown in the figure, nodes in a distributed system under this design can be divided into two types, namely main node and client node. Each client node contains a wrapper object, called client handler, and the binary executables that perform tasks. The binary executables in this design are called task objects. The main node is unique in the system and it contains system objects that handle the operations and provide supports in different areas in the system. Like client nodes, the main node can also provides and executes task objects. The functionalities of each of the objects are described in the following sections in this chapter.

The design chooses LINX[45] as the main communication tools. LINX is a message passing tools developed by ENEA Software AB, which is suitable for communications in a distributed environment. The main reason of using LINX in this design is that it can support Linux and OSE system and it provides some support in fault detection and locating target. The in-depth investigation on the tools is done by J. Larsson in XDIN internal documentation.

Although the current design cannot fulfill all the specifications listed in the previous chapter, it provides basic supports for real-time, accessibility, consistency, fault-tolerance and scalability. The detailed analysis of these supports is investigated in chapter 8. In this following chapter, the third question stated in the first chapter is answered:

How does the middleware handle the coordination of the system?

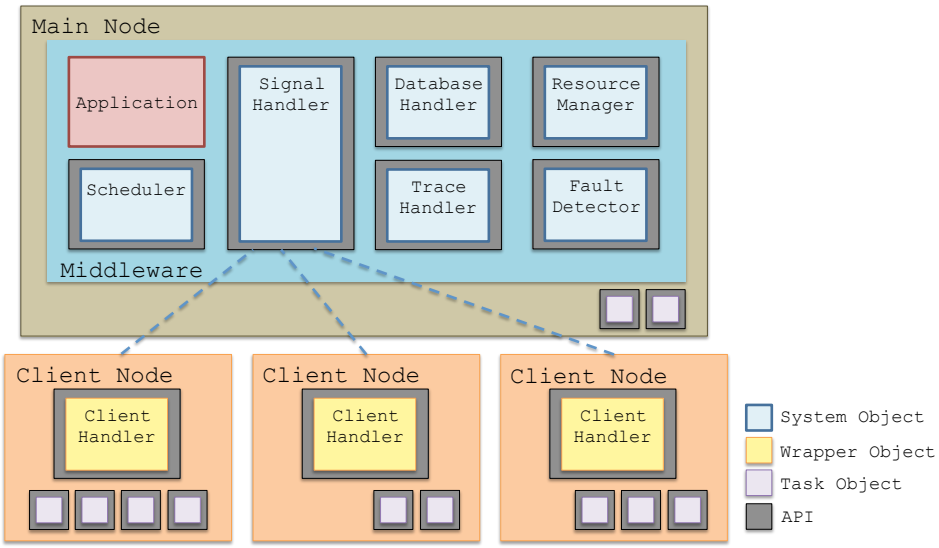


Figure 7.1. Overview of the distributed system

7.2 Operation in the system

In figure 7.2, the usual operation of the system is shown. The application starts with starting starting tasks through scheduler. The started task would then search for communication by the naming service from database handler or by trading service from resource manager. After knowing the ID of the target, the task can start communicating. The details of how the scheduler starts a task and how the trading is done is introduced in section 7.3.3 and section 7.3.5 respectively.

7.3 System object

There are many different system objects to maintain and support the system, and to provide communication medium in this design. They are all located in the middleware and is using system interface as the interface. In the current design, there are six different types of system objects in the middleware, namely: database handler, scheduler, signal handler, resource manager, fault detector and trace handler. System objects are unique in the system and only one copy of each system object can be run concurrently.

7.3.1 System interface

System interface (*mw_ipc*) is the interface for system objects. It hides the complexity of operations related to LINX. Through the system interface, the system objects can register in the database, send or receive messages, trade or find a task and trace the activities. System interface is having the highest degree of freedom among the

Step	Action
1	Scheduler starts a task according to the application.
2	Tasks find a target through naming service of the database handler to locate another task.
3	Tasks communicate through signal handler.

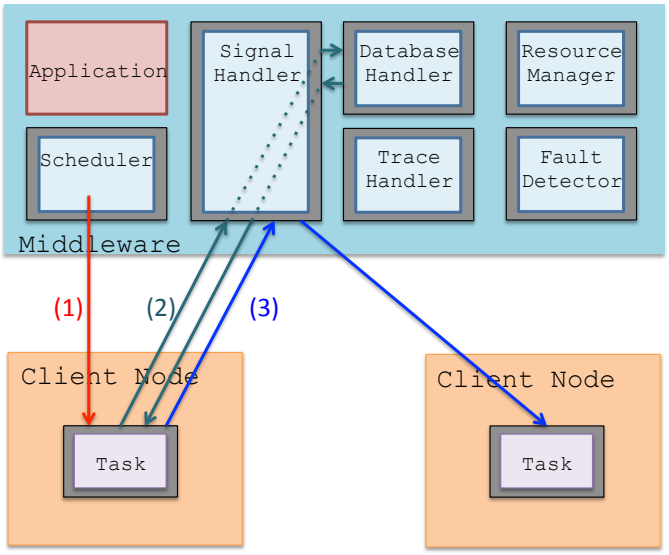


Figure 7.2. General communication in the system

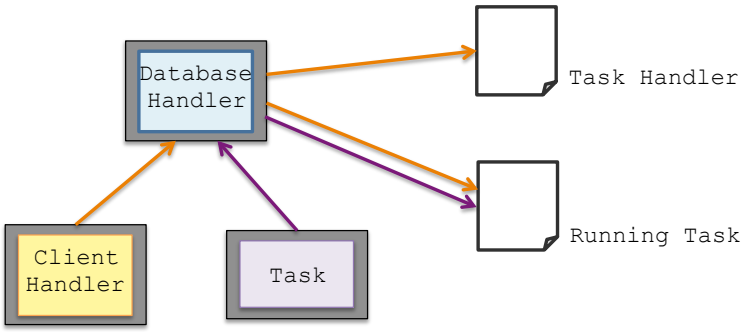
three interfaces and it can alter some of the system operation. This interface should only be used by designers of the middleware.

Functions provided in this interface is mainly divided into five area, initialization, trace, accessibility support, scheduling and communication function. For initialization functions, they initialize the component in the system including to open a socket for communicating and registering to the database handler. Trace functions provide functions that are used in order to keep track of the communication and data transmittio within the system, it is useful for debugging purposes. Accessibility functions allow the system object to locate a task. It can be any task that is registered in the database. Communication functions are used to send and receive messages without concerning how the communication is done. The functions provided in this interface is listed in Appendix A.

7.3.2 Database handler

Database handler (*db_handler*) records the tasks that are currently running in the system, it also stores the information of the capability of each node in the system.

Figure 7.3 shows the registration of tasks to the database with the help of database handler. There are two databases that are handled by database handler. One of which records the tasks that are currently running in the system, and it



Task Name	ID	Machine Name	Unit Type	Status
DB_HANDLER	65567	RPi1	DATABASE	READY
MIDDLEWARE	65568	RPi1	MIDDLEWARE	READY
NODE2	65580	RPi2	CLIENT_HANDLER	READY
RearRightWheelSensor	65581	RPi2	SENSOR	NOTREADY
NODE1	65584	PC	CLIENT_HANDLER	READY
.	.	.	.	.
.	.	.	.	.

Running Task

Program Name	Client List	WCET	WCCT
pedal_sensor	NODE1, NODE2	15, 20	5, 5
wheel_sensor	NODE2, NODE3	20, 10	5, 7
brake_controller	NODE1, NODE2, NODE3	17, 30, 20	5, 5, 7
.	.	.	.
.	.	.	.

Task Handler

Figure 7.3. Registration through database handler

is called running task database. The other one is called task handler database, it records the task that can be handled by each of the nodes. In the starting of the system, all of the components, including system objects, wrapper objects and task objects need to register themselves in the running task database, however, only client handlers need to be registered in the task handler database.

Running task database stores each task with task name, ID, machine name, unit type and the status. The task name is used for handling finding of tasks and the ID is used for communication using LINX. They are both unique in the system, if a task is trying to register itself using an existing task name without specifying it is re-registering, the database handler would reply with a failure message. The unit type and the status determine some of the operations in the system. For instance, the database uses the unit type to determine which order to terminate the components in the system and it uses the status to check if it is necessary to terminate or pause the system.

Task handler database stores the client nodes in which a task is runnable. It also

Step	Action
1	Scheduler reads the next task to be executed, and identifies the actual executable binary that handler the task through lookup table.
2	Scheduler sends a trade signal to database handler to obtain a list of client nodes that can run the executable binary program.
3	Scheduler identifies an available node and sends the task to that node.
4	Client handler in the receiving node receives the task and runs the executable accordingly.

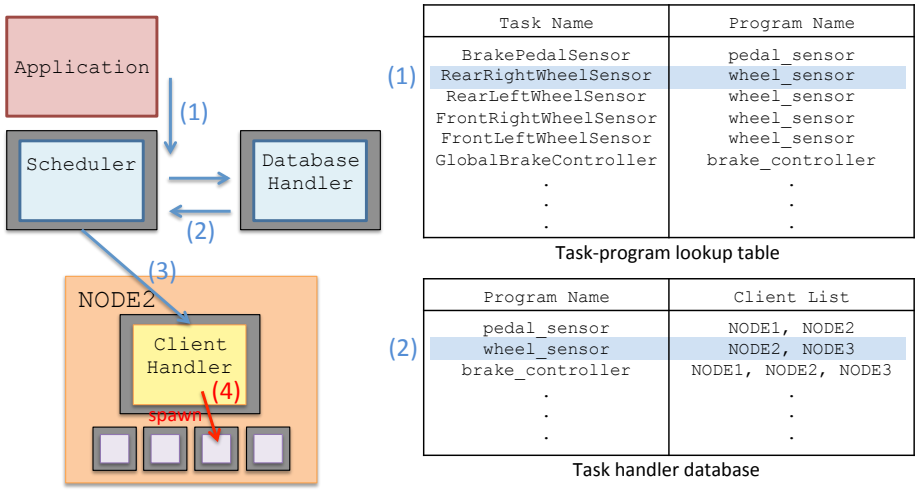


Figure 7.4. Example of distributing a task to a client node

stores the worst case execution time (WECT) and worst case communication time (WCCT) of the task running on a specific client node which is provided together with the register signal when the client handler register itself in this database.

With the information stored, database handler can handle the trading of client nodes and finding of tasks. The uses of trading of client nodes are shown in the examples in section 7.3.3 and section 7.3.6.

7.3.3 Scheduler

The scheduler (*scheduler*) implemented currently in the system is considered to be performing a static scheduling. The scheduler does not provide a timing scheduler, but it distributes the tasks among the client nodes. The scheduling of the tasks in each of the client nodes depends solely on the scheduler of the node, which may or may not be a real-time scheduler.

The distribution of tasks is done at the start up of the system. Figure 7.4 shows an example of how a task is deployed to a client node. Each deployment involves four steps. First, the scheduler loads the task to be run from the application list. Through a lookup table, the scheduler identifies the actual executable binary that executes the task. In this example, the task *RearRightWheelSensor* is to be run. From the task-program lookup table, it can be identified that this task is actually

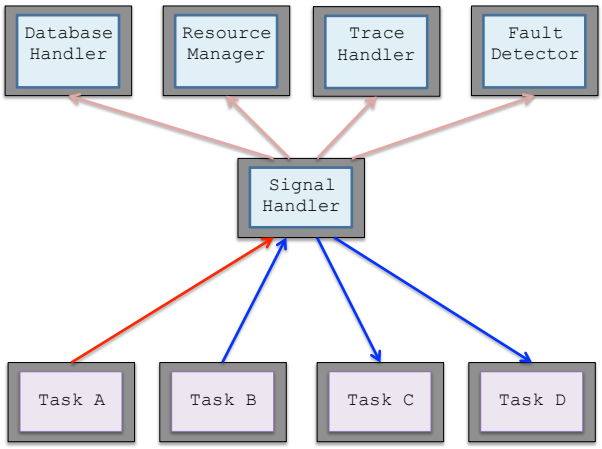


Figure 7.5. Signal handler

run by a program named *wheel_sensor*. It can be noted that *wheel_sensor* is the program for all *RearRightWheelSensor*, *RearLeftWheelSensor*, *FrontRightWheelSensor* and *FrontLeftWheelSensor*. The second step is to trade for the list of client nodes that can run this program by sending a trade signal to database handler. The reply signal from the database handler contains a list of client nodes that can handler the task. In this example, both *NODE2* and *NODE3* can run the task. Then, the scheduler calculates the utilization of the client node by the WCCT, WCET and the period of the task. It distributes the task to a client node that with a manageable utilization from the list by sending a task signal to the client handler. Finally, the client handler receives the signal and spawns the task in the local machine.

The scheduler guarantees that the task is runnable by the client node and the task is schedulable if the client node supports real time scheduling.

7.3.4 Signal handler

Signal handler (*signal_handler*) is the bridge between the task object to the middleware and it acts as the intermediate medium between two tasks. Figure 7.5 shows the path of communication from a task to system objects and from a task to other tasks.

As the figure illustrated, a task can communicate with the system object through the signal handler. In a distributed environment, it is not easy to locate a target component. In this design, the task objects only need to attach to the signal handler, and they can transmit the signals by sending the signal to signal handler and specifying the target that the signal is needed to send to. The same idea applies when a task is trying to communicate with another task.

Thanks to the help of the signal handler, in the current design, task objects are able to send out a message to a group of targets or receive a group of messages

Step	Action
1	Task sends a trade signal to resource manager to look for a task that is available ad is having the desired task type.
2	Resource manager receives the signal and identify a task that is suitable. It then locks the targeted task.
3	Resource manager replies to the trading task with the ID of the targeted task.
4	The task can communicate with the targeted task through the signal handler by specifying the ID.

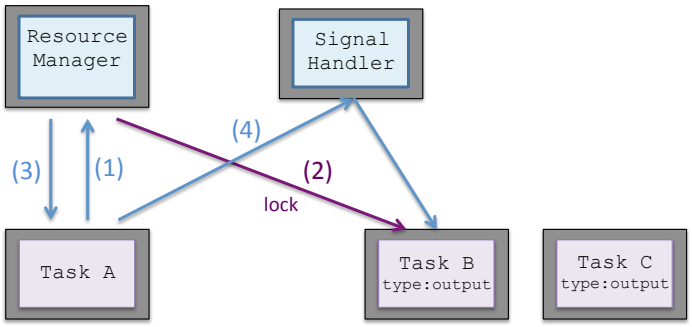


Figure 7.6. Example of trading and locking

from more than one source. They are called group send and group receive. The performance of group send and group receive are evaluated in the next chapter.

7.3.5 Resource manager

Common resources can be protected by resource manager (*resource_manager*). If a resource is registered to the resource manager, it can be found by other tasks through trading and can be locked by the resource manager.

The procedures of trading and locking a resource are shown in figure 7.6. To trade for a task, the trading task need to send a trade signal to the resource manager by specifying the type of task that it needs to search for. In this example, *Task A* wants to trade for a task that is with the type *output*. The resource manager would look for a task that is having this specific type and is available, it then labels the task by the status *LOCKED* depending on the operation that the trading task wants to perform. It then replies the ID of the locked task to the trading task and the task can then communicate with the target. In this example, *Task B* is the target, but *Task C* is equally qualified. So depending on the status of the tasks, trading can results in getting a different target. The target would be labeled as locked until the trading task releases the lock.

Though, currently, only a simple lock is implemented in the design, and it requires all application to follow the rule of checking the status of the targeting tasks before sending signals. If a task does not follow the rule, the protection would be invalid and if the application writer needs to take care of the possibility of deadlock manually.

Step	Action
1	Fault detector receives a signal indicating a task goes into zombie state.
2	Fault detector sends the information of the zombie task to scheduler for re-invocation.
3	Scheduler trades for a list of client that can run the task through database handler.
4	Scheduler identifies an available node and sends the task to that node.
5	Client handler in the receiving node receives the task and runs the executable accordingly.
6	Fault detector sends information of redirecting message from old task to new task to signal handler.

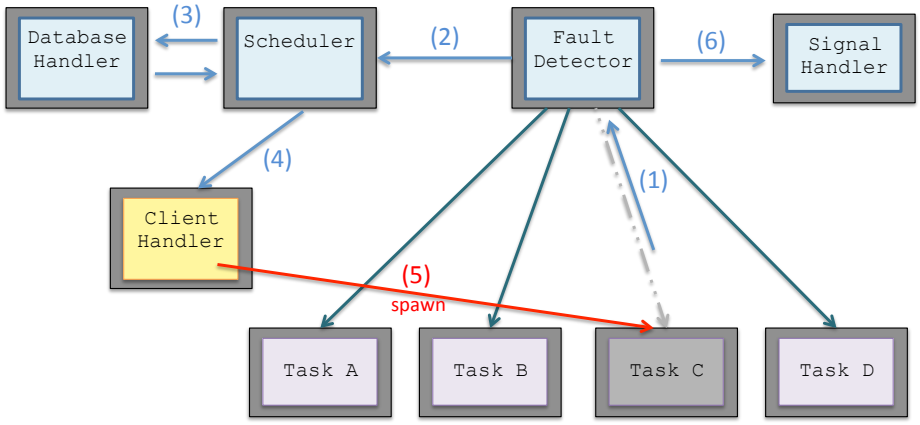


Figure 7.7. Example of fault handling

7.3.6 Fault detector

Fault detector (*fault_detector*) provides fault tolerance to the system. It attaches to all of the task objects that are in the system in order to detect the failure of tasks. Figure 7.7 shows an example of fault handling initiated by fault detector.

The fault detector attaches to task objects. If a task objects is terminated unexpectedly, a signal indicating the task object is in zombie status would be sent to the fault detector. The fault detector then passes the information of the failing task to the scheduler. Scheduler, like what it does in the initial scheduling state, trade for the list of client nodes that can handle the task. It then sends a task signal to a client node that is suitable. The client node receives the signal and spawns a task to continue with the operation. Once the new task is registered again in the system, the fault detector would send information on redirecting messages from old task to new task to signal handler.

In the current design, the fault detector is able to identify the failed tasks and re-invoke the task. Though, the states of task objects are not saved and the new task can only start from the beginning of the task. This design is currently only useful for periodic tasks that do not have states.

7.3.7 Trace handler

Trace handler (*traces*) is used to log the operations and the signal sent in the system. It provides timing information of the application and captures the activities within the system. The tracing can be configured to record actions in different level and in different nodes. This provides some timing information of the system and records all the traffic and information of the system. This component is designed by J. Larsson from XDIN AB.

7.4 Wrapper object

Wrapper objects are used to perform operations related to the hardware and operating system. They touch system command in order to run a task or, in future work, even to schedule the tasks. There are two types of wrapper objects, which are middleware and client handler. They use a different interface comparing to system objects.

7.4.1 Wrapper interface

Wrapper interface (*mw_wrapper*) provides functions for wrapper objects to register themselves, receive and spawn tasks and trace the activities. At the current stage, it does not provide any message sending functionality to wrapper objects as the wrapper objects are not supposed to be functional objects. The idea of wrapper interface is to adapt the client object to the hardware.

Wrapper interface hides the LINX and system commands from the wrapper objects. Since it is used for adapting hardware and operating system, so for wrapper objects running on different machines, they should have different wrapper interface. Though at this stage, only wrapper for Linux is developed.

Functions provided in this interface is mainly divided into four area, initialization, trace, task spawning and termination. For initialization and trace functions, they provide similar functions as that in system interface. Task spawning functions allow the wrapper object to receive tasks from scheduler and to start a task in the local node. Termincation function is a function that check if the application is ended or not. The functions provided in this interface is listed in Appendix B.

7.4.2 Middleware

Middleware (*middleware*) is the wrapper object of the main node. It is responsible for starting all the system objects in the system. It also provides some fault tolerance to the database handler, such that if database handler failed, it would re-invoke the database handler.

7.4.3 Client handler

Client handler (*client_handler*) is the wrapper object of the client node. It is responsible for registering the tasks that are runnable in the client node. It receives tasks and start running them accordingly. In the future plan, client handler would be responsible also for scheduling the tasks locally.

7.5 Task object

Task object is the object that actually runs the application. It can use the resources of the distributed system through the task interface. Task objects do not have much freedom in controlling the system, but they can send or receive messages, find or trade for a task and trace the activities.

7.5.1 Task interface

Task interface (*mw_port*) provides an interface for the task objects to communicate to the middleware and to other tasks. Like other interfaces, it hides the complexity of LINX. The aim of this interface is to provide an easy to use API for application writers to write their application.

Functions provided in this interface is mainly divided into five area, initialization, trace, timing, communication and resource managing. For initialization and trace functions, like wrapper interface and system interface, they initializes the component to establish a connection port, registers the component in the system and sending trace signals. Timing functions provide some support of measuring WCET and WCCT of a task and provide a sleep function for periodic tasks. Communication functions allow the sending, receiving messages and group communication in the tasks. For resource managing functions, tasks can trade or find other tasks or manipulate the lock of a common resource. The functions provided in this interface is listed in Appendix C.

7.6 Summary of the design

This chapter introduces and explains the functionalities of different components in the system. Different components play unique roles to the system. System object provides the main support and creates the middleware for the system. Wrapper objects isolate the programs from hardware heterogeneity and task objects perform the actual functionalities.

The three different types of components use different interfaces. The interfaces are the tools to develop the system and the applications. For application developers, only the task interface is relevant. The system interface is used for system developing, and to work with different machines, the wrapper interface is used.

The current design is functional but more supports and better supports are yet to be implemented. This thesis acts as the basis for the future development of this system.

Chapter 8

Implementation

8.1 Overview

The design is tested with different requirements through a few small test cases and a brake-by-wire system use case. This chapter, the results of the tests are presented.

The requirements are based on the list of requirements stated in chapter 2. This thesis respond the requirements on real-time support, scalability, fault tolerance, consistency and accessibility. Each of the test cases is supported with simulation results or graphs. This chapter answers the final question of this thesis:

How should the middleware be constructed in order to maintain high scalability and transparency?

8.2 Real-time support

To achieve real-time performance in a distributed system is not easy. It requires all the machines, the communication networks and the operating system to be real-time. This thesis does not give a solution on performing real-time applications, but it aims to provide basis that can be used in future development to realize real-time.

Related requirement: *SW_REQ_1, SW_REQ_2*

8.2.1 Task distribution

The real-time support provides in this design is the distribution of tasks. It is known that whether a task is schedulable in a machine depends on the utilization of that machine. In a single core machine, if the utilization is less than 100%, EDF scheduling can guarantee a feasible schedule exists. This thesis takes advantage of this point to distribute the tasks base on the utilization of client nodes. The idea is that each client handler registers the WCET and WCCT of each task that it can run to the database. The scheduler would calculate the utilization of a machine if the task is deployed to a it. As long as the utilization can be maintained lower than 100%, the task would be distributed to that machine.

Table 8.1. Task distribution simulation result

Task (Period/Deadline)	**Client1**	**Client2**	**Client3**	**Client4**
BrakeTorqCalc (50000)	14000	14000	10000	14000
LDM_BrakePedal (10000)	5000	N/A	N/A	N/A
GlobalBrakeController (50000)	18000	18000	15000	18000
ABSatRearRightWheel (50000)	10000	10000	8000	10000
LDM_RearRightBrake (10000)	N/A	2000	N/A	N/A
WheelTorqCalc (50000)	15000	15000	10000	15000
LDM_RearRightSensor (10000)	N/A	4000	N/A	N/A
WheelController (50000)	10000	10000	8000	10000
SpeedSimulator (50000)	40000	40000	35000	40000
SpeedAtRearRightWheel (10000)	2000	N/A	N/A	N/A
Total Utilization	**98%**	**96%**	**52%**	**80%**

8.2.2 Simulation and results

The simulation of this requirement is done by launching four client handlers, each have different capability of running certain task with different execution times. The simulation should give results such that the utilization of each client handler is less than 100% and only runnable tasks are allocated.

The registration information of each of the client handler and task to be run and the simulation results are listed in table 8.1. The task models that are used in this example is the task model of the brake-by-wire system. Here the period of a task and the deadline of the task is assumed to be the same. The numbers shown in each of the columns of the client nodes indicates the summation of the WCCT and WCET of that node. These numebrs are measured prior to the simulation in different client nodes, it varies from machine to machine. The highlighted cells are the targets that the tasks are deployed to. The utilization of each of the core is calculated by: .

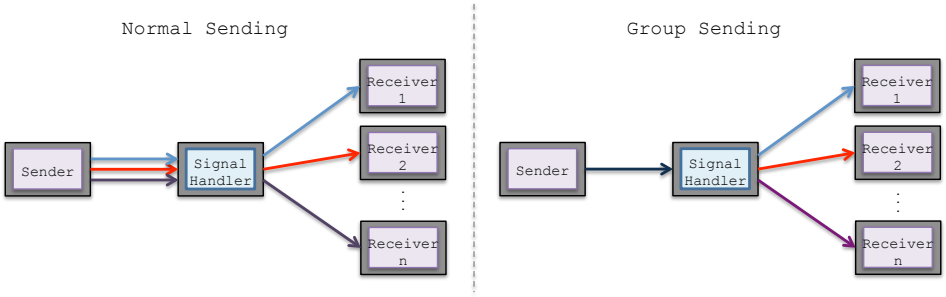


Figure 8.1. Comparision of normal sending and group sending

$$Utilization = \sum \frac{executiontime}{period}$$

It shows that all of the client handlers are having utilization of less than 100%, all of the tasks are distributed and all tasks are distributed to a runnable node. It can be concluded that the scheduler is able to distribute the tasks sensibly, it does not give the system a real-time performance, but it is the basic support that a real-time system would need.

This functionality can act as the basis to support a future development, such that EDF can be used in each of the local core. Note that the WCCT and WCET need to be profiled for each individual node. Without the actual timing information, real-time cannot be realized.

8.3 Scalability support

As stated in the goal of this thesis, scalability is one of the focuses in this system. The performance of the system should not result in a severe deterioration of performance. In this design, group sending and asynchronous method handling are used to address this issue.

Related requirement: *SW_REQ_3, SW_REQ_4*

8.3.1 Group sending

Group sending allows the task to send just one signal to the signal handler, and signal handler would broadcast the signal to the interested targets. This could reduce the communication traffic, especially when the system is large. The group sending is done by each task registering the group sending prior to the execution of the main task. During the execution, a task need to send a group sending signal to signal handler, the signal handler would then distributed the message to the registered target.

The difference of group sending and normal sending is illustrated in figure 8.1. It shows that in normal sending, the number of messages to be sent by the sender

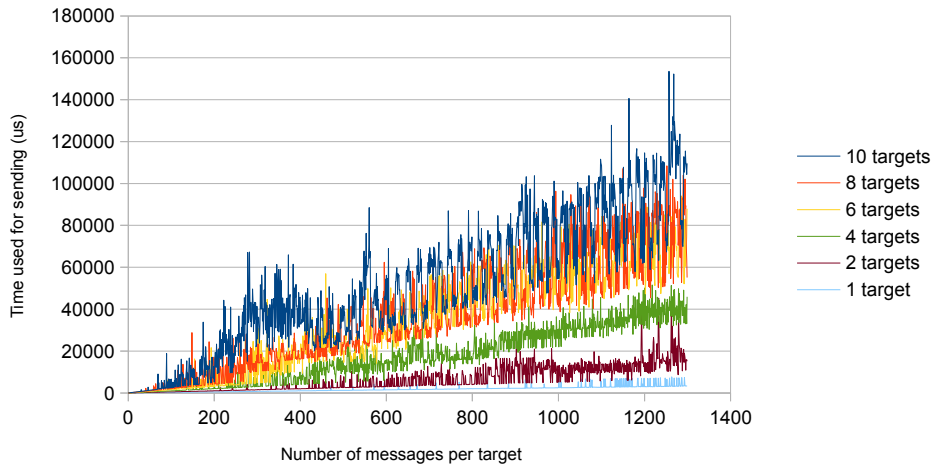


Figure 8.2. Time used for sending message in normal send

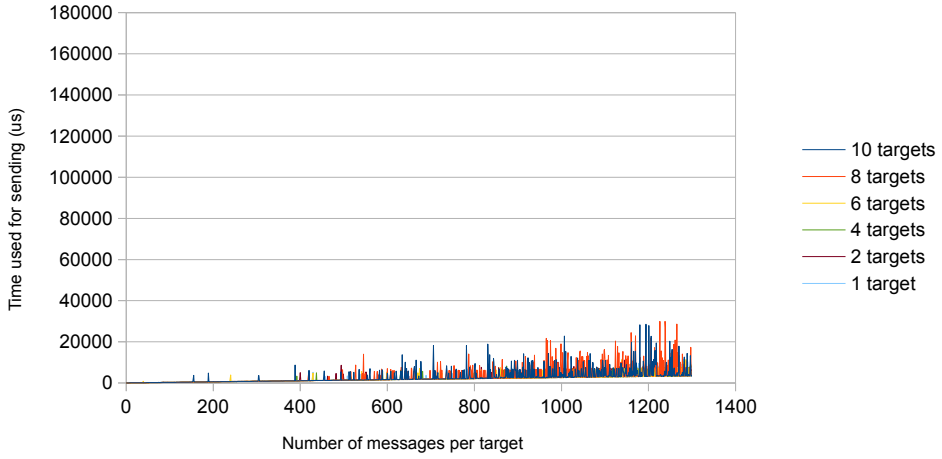


Figure 8.3. Time used for sending message in group send

is proportional to the number of targets. In group sending, however, the number of targets does not affect the number of sending in the sender. In any case, only one message is needed to send to the signal handler.

8.3.2 Simulation and results

There are two metrics to be investigated in the comparison of normal sending and group sending. They are the time used for the sending messages and the buffer usage.

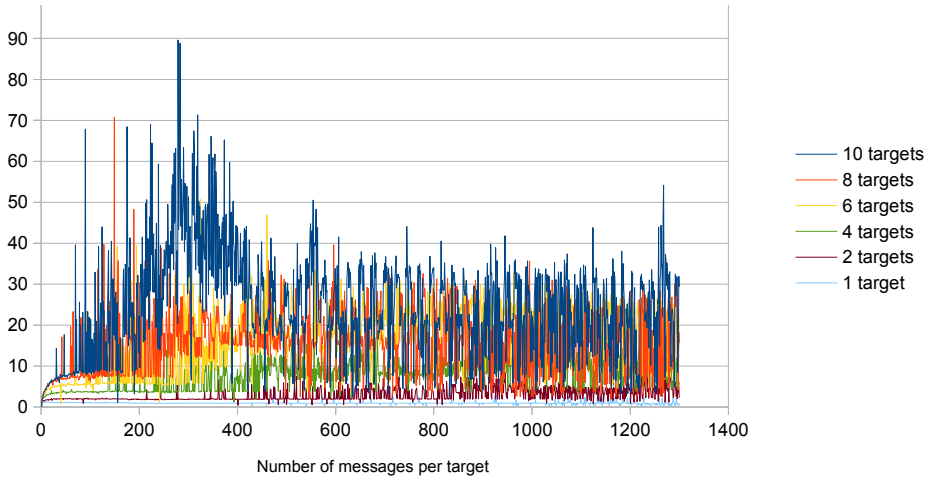


Figure 8.4. Ratio of time used for normal send to group send

Time performance

To simulate the time used for sending message, the sending task is set to be sending to a range of targets, from one target up to ten targets. The messages are sent from the sender to the signal handler, and the signal handler redirects the messages to the targets. The sending task loops to send an increasing number of messages to all of the targets every cycle, i.e. in cycle one, it sends out one message to each of the target; in cycle two, it sends out two messages and so on. The time needed to send the messages are recorded and are plotted in figure 8.2 and figure 8.3.

Figure 8.2 shows the time used for sending messages by using a normal way of sending. It means that if there are ten targets paired with the sender task, the sender task needs to send ten times for every message. As show with the graph, with more targets paired, it would need a long time to send out messages. It is logical to assume the time used for sending certain number of messages to more targets need a long time as it involves more send operations from the sender to the signal handler. The time needed is proportional to both the number of messages to be sent and the number of targets.

When the same setup is compared to using group send. It can be seen even with different numbers of targets, the time used ramps up in about the same rate. It can be explained by that the number of send operations from the sender to the signal handler does not depend on how many targets. Therefore, the time needed to send the messages is solely proportional to the number of messages to be sent.

It is logical to assume that the ratio of time needed to send under normal send to group send is equal to the number of targets, since with the same number of messages per target, the normal send need to send out one time for each target while group send send out one time for all target. Though, this is not entirely true. The ratio of time needed to send under normal send to group send is plotted in

Table 8.2. Average ratio of time needed for normal send to group send

Number of targets	Average ratio
1	0.97
2	3.01
4	8.16
6	12.20
8	15.99
10	24.46

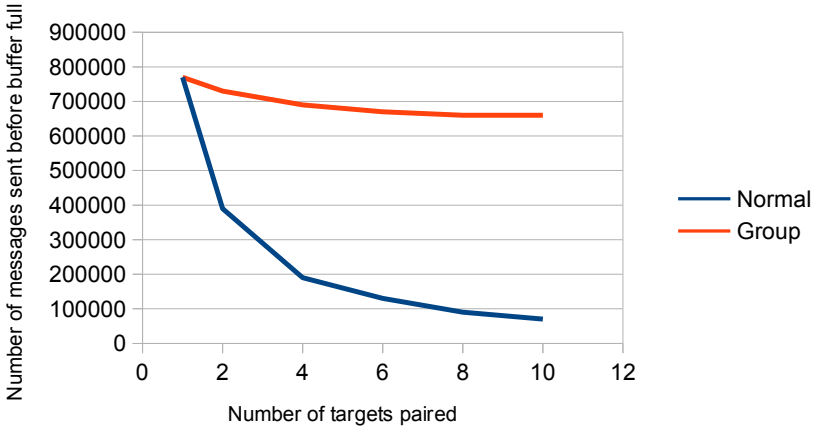


Figure 8.5. Maximum message can be sent for normal send and group send

figure 8.4.

It can be seen that as the number of targets increases, the ratio increases further. The average ratio is listed in table 8.2. This can be argued by the increase number of send operations may queue up in the outgoing buffer of the sending task. Thus, it can increase the delay and therefore the ratio would be larger than the number of targets.

This shows that with the help of the group send, a task that sends out message would not be stalled by the sending operations when the number of communicating targets increases. It can be concluded that this measure provides a better scalability in the sense that the performance of a task, at least for the sending part, would not be deteriorated greatly when the system scales up.

Footprint

When the receiving component is contented in a communication, the messages queue up in the buffer and wait for receiving. The buffer size and the rate of injection determine the capability of the receiving component. As explained previously, the

Table 8.3. Comparison of maximum message between normal send and group send

Number of targets	Normal Send	Group Send
1	770000	770000
2	390000	730000
4	190000	690000
6	130000	670000
8	90000	660000
10	70000	660000

number of messages to be sent from a sending task to the signal handler using a normal send would be a multiple of that in group send. Therefore, it would be interesting to investigate the capability of the signal handler on: how many massages per target can be sent before the buffer of the signal handler is full.

The simulation is done with the default setup of the LINX. In both normal send and group send, the same setup is used. The limits before the system complains about buffer full are recorded and are plotted in figure 8.5. The y-axis in this graph indicates how many messages per target can the signal handler handle. By varying the number of targets paired to the sending task, it can be seen that for the normal send, the number of messages that the signal handler can handler drop significantly when the number of targets increases, while that for group send remain quite steady. The rough figures are listed in table 8.3. These figures vary from simulation to simulation due to the indeterministic of the network and task execution, but the rough numbers are obtained. By comparing the numbers, we can conclude that the group sending reduces the footprint or buffer needed due to scaling up of the system, and it can be said to support some degree of requiring less resources for sending messages.

8.3.3 Asynchronous method handling

As mentioned in section 4.2.4, the asynchronous method handling can increase the delay of a send-and-receive operation. The different of a synchronous communication and the asynchronous method handling is illustrated in figure 8.6 and figure 8.7. The different colors represent the different communication sets that are handling by the unit.

In this example, both *Sender A* and *Sender B* want to send a message to the receivers and wait for a reply. In the case of synchronous communication, the request from *Sender A* cannot be handled by the middleware until the request of *Sender B* is completed. This delay can be extended if there are multiple groups of senders and receivers, and the total delay would be at least the summation of all the execution times of the receivers.

In the same situation, the asynchronous method handling can react to the request once the previous request is transferred to the receiver. If there are multiple

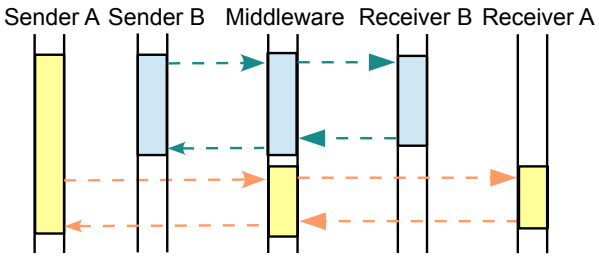


Figure 8.6. Example of synchronous communication

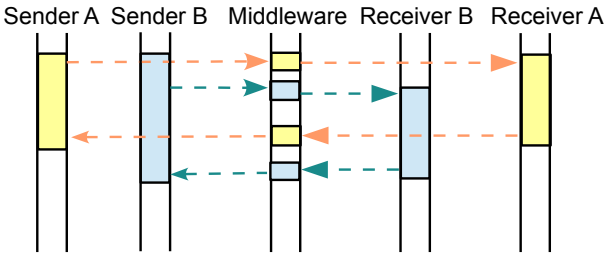


Figure 8.7. Example of asynchronous method handler

groups of senders and receivers, it can be expected that the total delay would not be much greater than the longest execution time among the receivers.

8.3.4 Simulation and results

The simulation of this requirement is tested by simulating the maximum delay from a sending task sending a message to the time it receives the respond. In the setup, all of the sender tasks send out the requests at the same time. By varying the number of concurrent communication pairs and the execution time of the receivers, the maximum delays are recorded and are plotted in figure 8.8 and figure8.9.

The results, as shown, reflect the reasoning stated in previous section. In a synchronous communication, the maximum delay is proportional to both the number of concurrent communicating pairs and the execution time needed for the receiver. The delays stack up as the number of pairs increases. For the asynchronous method handling, the delay is proportional only to the execution time of the receivers. The increase of concurrent tasks does not affect the delay much. It can be concluded that with the asynchronous method handling, the system is able to scale up without scarifying the performance, unlike the synchronous communication does.

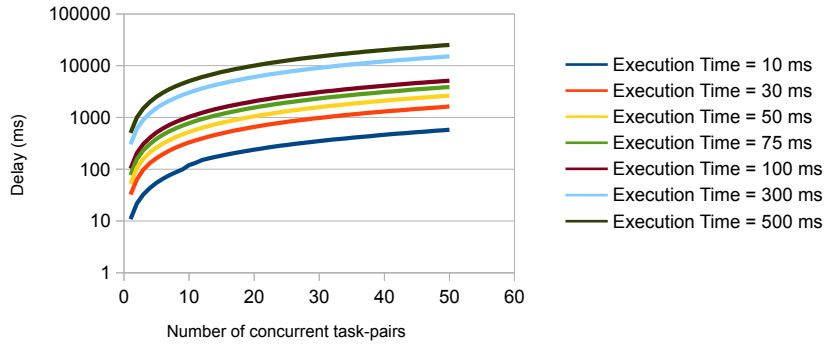


Figure 8.8. Delay in synchronous communication

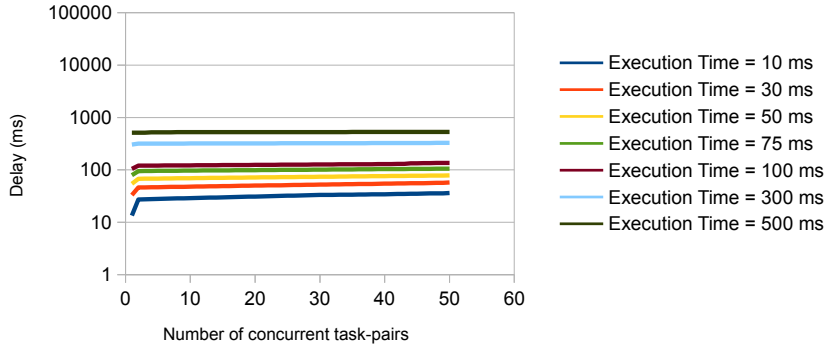


Figure 8.9. Delay in asynchronous method handling

8.4 Accessibility transparency

Another focus of this design is to provide a high transparency. The task object should be able to communication throughout the system regardless the location of the communicating partner. In this design, this is supported by trading, naming and centralized communication.

Related requirement: *SW_REQ_8*

8.4.1 Trading and naming

The principle of trading in this system was shown in section 7.3.5. The naming function is sharing a similar idea. A target can be searched by specifying the task name of the target. The application does not need to concern about the location of the task, whether of not it is located in the same node or in remote node. This, thus, provide a better transparency.

```
TASKB3: with IP: 65604
TASKA3: with IP: 65606
TASKB1: with IP: 65601
TASKB2: with IP: 65603
TASKA2: with IP: 65602
TASKA1: with IP: 65605
TASKA1: locates a receiver with IP: 65603
TASKA2: locates a receiver with IP: 65604
TASKA3: locates a receiver with IP: 65601
TASKB2: receives from 65605
TASKB3: receives from 65602
TASKB1: receives from 65606
TASKB2: receives from 65605
TASKB3: receives from 65602
TASKB1: receives from 65606
TASKB2: receives from 65605
TASKB3: receives from 65602
TASKB1: receives from 65606
...
```

Figure 8.10. Simulation output of trading

8.4.2 Centralized communication

By centralizing the communication in the signal handler, the client nodes do not need to know the connection information, like IP address or hardware address, of target nodes. The only node matters to client nodes is the main node. As long as the client nodes are attached to the main node, it is able to communicate with all other nodes in the network.

8.4.3 Simulation and results

The simulation of this requirement is done by starting two types of task object, one of which is the sender and the other type is the receiver. In total, there are three senders and three receivers. The senders do not know the IP address nor the task names of the receivers. The simulation should show that the senders are able to locate the receivers and do the communications successfully.

Figure 8.10 is the console output of simulation. The tasks with names *TASKAX* are the sender tasks and the tasks with names *TASKBX* are the receiving tasks It can be seen that all senders are all able to locate receivers. The output from *TASKA3* and *TASKB1* are highlighted to explain the results. From the beginning, it shows that *TASKA3* is having ID 65606 and *TASKB1* is having ID 65601. *TASKA3* locates a receiver with ID 65601, which is the ID of *TASKB1*. Therefore, in the execution below, it can be seen that the *TASKB1* is only receiving messages from 65606, which is the ID of *TASKA3*. Similar observations can be seen in other pairs.

This shows that with the resource manager, tasks can trade for tasks by spec-

Task Name	Node	Original ID	Final ID
DB_HANDLER	MAIN_NODE	65660	65660
SCHEDULER	MAIN_NODE	65656	65656
RESOURCE_MANAGER	MAIN_NODE	65657	65657
SIG_HANDLER	MAIN_NODE	65658	65658
FAULT_DETECTOR	MAIN_NODE	65659	65659
MIDDLEWARE	MAIN_NODE	65655	65655
NODE1	RPi1	65662	65662
NODE2	RPi2	65663	65663
DUMMY	RPi2	65664	65664
BrakeTorqCalc	RPi1	65665	65665
LDM_BrakePedal	RPi1	65666	65666
GlobalBrakeController	RPi1	65671	65679
ABSatRearRightWheel	RPi1	65667	65667
LDM_RearRightBrake	RPi1	65668	65668
WheelTorqCalc	RPi1	65672	65678
LDM_RearRightSensor	RPi1	65669	65669
WheelController	RPi1	65673	65677
SpeedSimulator	RPi1	65670	65670
SpeedAtRearRightWheel	RPi1	65674	65676
STARTER	RPi1	65675	65675

Running task list

Index	Source ID	Target ID
[0]	65674	65676
[1]	65673	65677
[2]	65672	65678
[3]	65671	65679

Message transfer list

Figure 8.11. Results for handling of faulty tasks

ifying the type to determine the communicating partners. This can be useful in many applications. For example, if the system is used in an infotainment system, when the system starts, it would try to trade for hardware controller of a specific hardware, like a speaker. It does not require the information to be hard-coded in the program and it allows certain degree of reconfiguration

8.5 Fault tolerance

Depending on the use of the system, fault tolerance can become critical in a distributed embedded system. Tasks should be guaranteed to be running in order to deliver desired results. A good system should be able to identify faulty tasks and recover from the failure without intervention of the user.

Related requirement: *SW_REQ_6*

8.5.1 Fault detection and signal redirection

The basic mechanism of the fault handling in the design was discussed in section 7.3.6. The general idea is to detect the failing task, re-invoke the task and redirect the messages.

8.5.2 Simulation and results

Two simulations are designed to test this requirement. For the first one, a number of inter-communicating tasks are running in the system. Some of the tasks are killed intentionally to see if the messages can still be transmitted or received in the system. The transfer table in the signal handler and the running task list in the database handler are compared to see if the redirection is done correctly. The comparison of one of the simulation is listed in table 8.11.

Task Name	Original Node	Original ID	Final Node	Final ID
DB_HANDLER	MAIN_NODE	65735	MAIN_NODE	65735
RESOURCE_MANAGER	MAIN_NODE	65732	MAIN_NODE	65732
SCHEDULER	MAIN_NODE	65731	MAIN_NODE	65731
SIG_HANDLER	MAIN_NODE	65733	MAIN_NODE	65733
FAULT_DETECTOR	MAIN_NODE	65734	MAIN_NODE	65734
MIDDLEWARE	MAIN_NODE	65730	MAIN_NODE	65730
NODE1	RPi1	65737	---	---
NODE2	RPi2	65738	RPi2	65738
DUMMY	RPi2	65739	RPi2	65739
GlobalBrakeController	RPi1	65741	RPi2	65755
BrakeTorqCalc	RPi1	65740	RPi2	65756
LDM_BrakePedal	RPi1	65744	RPi2	65760
ABSatRearRightWheel	RPi1	65742	RPi2	65754
LDM_RearRightBrake	RPi1	65747	RPi2	65758
WheelTorqCalc	RPi1	65749	RPi2	65753
WheelController	RPi1	65743	RPi2	65752
SpeedSimulator	RPi1	65745	RPi2	65761
SpeedAtRearRightWheel	RPi1	65746	RPi2	65759
STARTER	RPi1	65748	RPi2	65751
LDM_RearRightSensor	RPi1	65750	RPi2	65757

Running task list

Index	Source ID	Target ID
[0]	65748	65751
[1]	65743	65752
[2]	65749	65753
[3]	65742	65754
[4]	65741	65755
[5]	65740	65756
[6]	65750	65757
[7]	65747	65758
[8]	65746	65759
[9]	65744	65760
[10]	65745	65761

Message transfer list

Figure 8.12. Results for handling of faulty node

It can be seen that the running task list updates the ID of the faulty tasks successfully and the transfer table gives a correct redirection. The four highlighted tasks are the tasks got terminated intentionally and re-invoked. The changes in the running task list in database handler are coherent with the message transfer list in signal handler. It cannot be shown by figures or tables, however, the tasks are able to communicate after the re-invocation of the tasks.

In another simulation, the setup is similar to the previous one, but instead of running all of the tasks in a client node, the tasks are distributed to two different client nodes. Also, instead of killing certain tasks, a client node is physically removed from the network by unplugging the LAN cable. A similar comparison table of the simulation is listed in table 8.12.

It can be seen that the tasks from the unplugged node *RPi1* successfully move to another node *RPi2*, except for the task *NODE1*, which is the wrapper object for the failing node. Like the previous simulation, the message transfer list is coherent with the running task list. The communications are resumed after the re-invocation of the tasks.

From these two simulations, it can be concluded that the fault detector is able to provide fault tolerance to the system without intervention of users. Though, in the current design, it must exists capable nodes for the faulty tasks in order to resume the operations. For tasks that communicate with sensors or actuators, the moving of a task from one client node to another client node may not always be possible. Also, if a client node is removed from the network, it cannot be reregistered to the system during runtime and the node can no longer be used unless the system restarts.

```
RESOURCE_MANAGER: resource info: [0] 65560, NOTLOCKED, owner is :0
RESOURCE_MANAGER: resource info: [0] 65560, LOCKED, owner is :65559
RECEIVER: receives from 65559
RECEIVER: receives from 65559
RECEIVER: receives from 65559
RECEIVER: receives from 65559
RECEIVER: receives from 65559
RESOURCE_MANAGER: resource info: [0] 65560, NOTLOCKED, owner is :0
RESOURCE_MANAGER: resource info: [0] 65560, LOCKED, , owner is 65561
RECEIVER: receives from 65561
RECEIVER: receives from 65561
RECEIVER: receives from 65561
RECEIVER: receives from 65561
RECEIVER: receives from 65561
RESOURCE_MANAGER: resource info: [0] 65560, NOTLOCKED, owner is :0
RESOURCE_MANAGER: resource info: [0] 65560, LOCKED, owner is 65556
RECEIVER: receives from 65556
RECEIVER: receives from 65556
RECEIVER: receives from 65556
RECEIVER: receives from 65556
RECEIVER: receives from 65556
RESOURCE_MANAGER: resource info: [0] 65560, NOTLOCKED, owner is :0
```

Figure 8.13. Simulation output of resource locking

8.6 Consistency

The last requirement tested in this thesis is the consistency. In a distributed environment, resources are shared and can be commonly used. If the usage of a resource is not controlled, faulty output can occur. In this design, the consistency is done by resource locking.

Related requirement: *SW_REQ_9*

8.6.1 Resource locking

Resource locking is supported by the resource locking using the resource manager. The procedures of the locking is to request of locking a target by sending a signal to the resource manager and check the reply signal to see if the lock is available for the requesting task. It is quite similar to the trading procedures described in section 7.3.5.

8.6.2 Simulation and results

To test with this simulation, the system is set such that there are three sender tasks, each trying lock a common receiving task then send five messages to it. The receiving task print out the source of the messages once it receives any. The resource lock would lock the receiving task until the lock is released by the sending task after all five messages are sent.

Figure 8.13 shows the console output from the receiving task and the information printed out by the resource manager. *RECEIVER* is having ID 65560, and the three sender tasks are having ID 65556, 65559, 65561. It shows that when the lock is

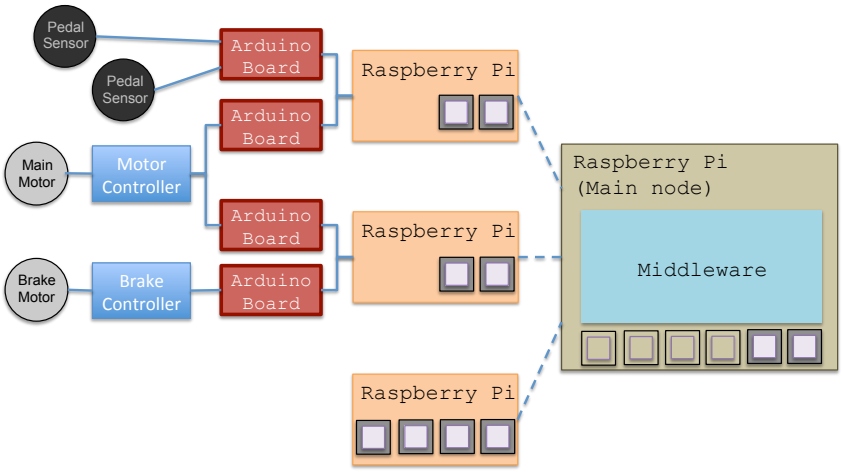


Figure 8.14. Overview of the brake-by-wire system

applied, only the owner of the lock can access the locked task, and after the lock is released, other tasks can try to own the lock. This shows that the resource manager is able to provide a support of locking and avoid contention in an object. Though, again as stated in 7.3.5, all tasks must follow the rules to guarantee the locking is valid.

8.7 Use case: brake-by-wire system

The system designed is tested with the application, brake-by-wire system. This application is simple to be distributed, yet it requires much communication, fault tolerance, or even real-time performance.

8.7.1 Introduction of brake-by-wire system

Brake-by-wire system is a system aim to replace mechanical braking component with electrical sensors and actuator [54]. In a brake-by-wire system, tasks are communicated interactively. By sensing the posistion of the brake pedal and the current rotational speed, the system determine how much torque should be supplied to the brakes of each of the individual wheel.

Anti-lock Braking System (ABS) is part of the brake-by-wire system that uses information of the rotational speeds to determine if the wheel is under slipping, and thus control the appling torque accordingly.

Since a braking system requires deterministic behaviour, the tasks within the system have hard deadlines and the system should be robust and fault-tolerant.

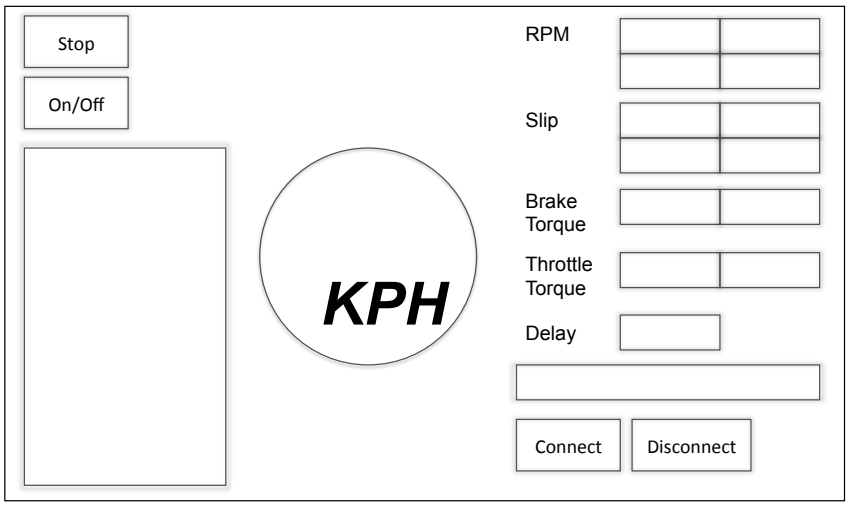


Figure 8.15. Layout of the Android control and display device

8.7.2 Simulation setup of the use case

The demonstration is a collaboration of six students' works. Besides the author of this thesis, the group includes: J. Larsson, A. Kouhestani, J Pérez, R. Hultman and T. Lindblad, with the supervision of B. Claesson, D. Scholle and help from S. Ullström, D. Skoglund and M. Saadatmand of XDIN AB.

This use case is simulated with the designed system. On top of the brake-by-wire system, a simple task that controlling a motor by a throttle pedal is also implemented in this use case. The system setup is shown in figure 8.14. As shown in the figure, the Raspberry Pi is connected to the sensors and motor controllers through Arduino boards. These boards act as the interface and provide analog to digital conversion and generation of Pulse-Width Modulation (PWM) signals to read in from sensors and to control the controllers. The demonstration of this system consists of two actual motors, two motor controllers, a wheel, two pedals and two potentiometers as the sensors.

In this setup, the use case is run by four Raspberry Pi: one for the controlling of sensors from pedal and the actuator for the main motor, one for the sensing of wheel rotation speed and the actuator of the brake, one for some of other calculation tasks and one for the middleware and backup of non-hardware related tasks.

The system connects also to an Android device for controlling the starting, stopping and pausing, and displaying information of the system. The communication between the system and the android device is through Mosquitto[55], an implementation of MQTT[56], which is a message passing protocol through publish and subscribe. This part is not part of the system, but is only for the demonstration purpose of the brake-by-wire system. The layout of the Android device is not designed by this thesis, but is shown in figure 8.15. This Android device shows the information, like the sensor values, actuator values, delay and all important information in

the system.

8.7.3 Challenges of brake-by-wire system

The brake-by-wire system consists of many functional components, each performs a simple calculation. The traffic in the system is heavy and requires responsive middleware for handling the communication. Since each of the task objects has only one port and they need to communicate with more than one other task. Some of the measures are needed to be taken care of in order to have a efficient communication.

As a real-time system, the tasks should not be blocked by the waiting for messages and sending messages, and the components in the system should be able to set up all the configuration, including starting communication with the hardware peripherals, locating communication targets, etc. prior to the execution of the function part. Also, in this system the tasks should be able to receive from more than one source with the knowledge of where the received messages are from. Though not necessary, the system also requires certain fault handling to become more robust.

To address these challenges, on top of the support that the system provides, there are functionalities developed in the system specifically for this use case. They are two-phased execution and group receiving.

8.7.4 Two-phased execution

Two-phased execution is to divide the execution of the application in the two parts, the second part is a normal execution of the periodic tasks, the first part is used for a set up prior to the normal execution.

In a distributed system, each of the tasks can be started at different point of time, and before entering the main loop, different tasks may need different amount of time to setup. The time point of every tasks are "ready" in the system is not precise. Therefore a synchronous starting is implemented to divide the execution into two phases.

In the first phase, which is called initial phase, tasks register themselves into databases, locate tasks to communicate with, set up the target of group sending and source of group receiving and configure the hardware to communicate with. Once a task has finished all the setup, it issues to the database handler indicating that it is ready. The database handler gathers all the ready signals and once all the tasks are ready, it would send out signals to the system to initiate the starting of the execution phase. This aim to provide a closer start times throughout the system and guarantee every task is ready to commit.

8.7.5 Group receiving

As mentioned, the tasks need to receive messages from more than a source. The messages can be ready at different points of time and with different orders every time. Also, if the receiving task is executed slower than expected, the messages may

queue up in the receiving buffer of the task. In a long run, this delay may queue up and results in a delayed reaction in the end actuators.

Group receive is used to handle these problem. The idea of the group receive is that the receiver registers the sources that it would receive from in a predefined order during initial phase. When the system enters the execution phase, the task send a requests to the signal handler and the signal handler would then reply with the messages that it gathered in the order of the registration. In this way, the order of the receiving messages represents the source of the messages, and since the signal handler is the only object that sending messages to the receiver, the order would be determined.

Another thing this group receive does is that it keeps the most updated values of the messages. For instance, if a receiving task is delayed for a cycle, two messages from each of the sources would have sent to the signal handler. The signal handler would not queue up the messages, but it would overwrite the old messages with the new ones. In this sense, the receiving end can already get the most recent values that the signal handler obtained, and the problem of stacked delay can be eliminated.

8.7.6 Results of the use case

The role of this use case is to validate some of the design decision. The use case is a good example of the designed system. Each node is running a copy of client handler to receive and spawn tasks. The client nodes have no knowledge of other client nodes in the rest of the system. Through communicating with middleware and using the accessibility supports provided, all the client nodes are working together and form the brake-by-wire system.

From the simulation, it shows that with the tasks are deployed according to the registration of the client nodes. The tasks communicate efficiently and effectively in the system. The system is responsive without noticable delay. This simulation also demostrates the fault tolerance that the system provide. During the runtime, a client node is removed from the system. The fault detector successfully detects the faulty node and respawns the tasks in a backup client handler.

The brake-by-wire system is tested with expected functionalities. The system, at the moment, cannot achieve a real-time performance, but it can demonstrate the responsiveness, fault tolerance and successful communications. It can be concluded that this system provides basic supports and functionalities to run actual applications. There is more work to be done in order to achieve real-time, but this design acts as a good start to an actual distributed real-time embedded system.

8.8 Conclusion on the implementation

In this chapter, the performance of different areas of the system is tested and the results are evaluated. It can be concluded that the system is a functioning distributed system that takes into account of transparency, scalability, real-time support, fault

tolerance and consistency. The design is not yet a complete system, but it is able to demonstrate support in different aspects, and it can run actual applications with correct functionalities. In summary, this system is a good start for building an actual system.

Chapter 9

Conclusion and Future Work

9.1 Conclusion on the project

This project covers both the academic study and actual design and implementation of middleware and distributed real-time embedded system. In the academic study, through analyzing different existing distributed system, a long-term specification is determined with the concerns of different requirements, focusing mainly on different transparencies of the system.

The project starts reaching the specifications by building up a basic distributed system with concerns in the requirements that the project is targeting on. The current design from this project is successful and it demonstrates the different supports, functionalities and potentials in the system. It is a good starting point for further developments.

A use case is used to validate the functionalities of the system. Through that, it can be concluded that the system is able to provide basic supports of reducing memeory footprint, communication and fault tolerace.

9.2 Limitations

As a first step of building a complete system, there are lots of limitations in the system that need to be overcome. In this report, some of the limitations are mentioned. Here, they are listed together with other limitations known in the system:

- The system does not provide real-time solution. Real-time performance requires all the machines, operating systems and communication network to be real-time and/or deterministic.
- The scheduling is an offline scheduling.
- Point-to-point communication cannot be done.
- Fault handling requires an available client node to re-invoke a task, a faulty node cannot restart and resume in the system.

- The re-invocation of a task does not inherit the states of the old task.
- Task migration is not possible.
- Consistency requires the tasks to be written with certain rules.
- Locking of resource is done only by a simple lock.
- The system does not detect or take care of deadlock.
- QoS support is not yet implemented.

9.3 Future work

Based on the limitations and other concerns, a list of future work is identified:

- Implement real-time scheduling in client node.
- Implement online scheduling and online profiling of execution time.
- Implement a deterministic communication platform.
- Use cold standby to save the states of the tasks
- Allow adding of new nodes during runtime.
- Provide support of point-to-point communication.
- Implement hierarchical locking.
- Actively check the locking status of a task.
- Provide end-to-end QoS support.

Appendix A

System interface

Initial functions

```
int gethostname(...)  /* get the name of the machine */
int mw_ipc_opensocket(...) /* open a socket for LINX communication */
MW_SPID mw_ipc_find_server(...) /* find the db_handler or trace_handler */
int mw_ipc_register(...) /* register to database handler */
int mw_ipc_unregister(...) /* unregister from database handler */
MW_SPID mw_ipc_getSpid(...) /* get the ID of the target, the ID is local */
int mw_ipc_findAndAttach(...) /* find and attach to a task */
int mw_ipc_attach(...) /* attach to a task */
```

Trace functions

```
struct timeval mw_ipc_get_timestamp(...) /* get the current time stamp */
int mw_ipc_set_tracelevel(...) /* set the trace level */
int mw_ipc_trace_info(...)  /* info tracing */
int mw_ipc_trace_profile(...) /* trace the wcet and wcct */
int mw_ipc_trace_error(...) /* trace error messages */
int mw_ipc_trace_group(...) /* trace group sending messages */
int mw_ipc_set_trace(...)  /* set trace signal */
int mw_ipc_trace(...) /* send a trace signal */
```

Accessibility supporting functions

```
void mw_ipc_trade(...) /* trade for a list of handler for a task */
MW_SPID mw_ipc_sender(...) /* get the ID of the sender of a LINX signal */
MW_SPID mw_ipc_findTask(...) /* find a task by name */
```

Scheduling functions

```
int mw_ipc_spawn_middleware(...) /* restart the middleware in case if it fails */
int mw_ipc_sendTask(...) /* send a task to a client node */
```

Communication functions

```
struct MW_MESSAGE mw_ipc_receiveMessage(...)  /* receive a message */
int mw_ipc_sendMessage(...) /* send a message */
```

Appendix B

Wrapper interface

Initial functions

```
int gethostname(...)  /* get the name of the machine */
int mw_wrapper_opensocket(...) /* open a socket for LINX communication */
MW_SPID find_server(...) /* find the db_handler or trace_handler */
int mw_wrapper_findAndAttach(...) /* find and attach to a task */
int mw_wrapper_registerTask(...); /* register the task runnable in this node */
int mw_wrapper_register(...); /* register to db_handler */
int mw_wrapper_unregister(...); /* unregister from db_handler */
```

Trace functions

```
struct timeval mw_wrapper_get_timestamp(...) /* get the current time stamp */
int mw_wrapper_set_tracelevel(...) /* set the trace level */
int mw_wrapper_trace_info(...)  /* info tracing */
int mw_wrapper_trace_profile(...) /* trace the wcet and wcct */
int mw_wrapper_trace_error(...) /* trace error messages */
int mw_wrapper_set_trace(...)  /* set trace signal */
int mw_wrapper_trace(...) /* send a trace signal */
```

Task spawning functions

```
int mw_wrapper_receiveTask(...) /* receive a task from scheduler */
int mw_wrapper_spawn(...) /* spawn a task locally */
```

Termination function

```
int mw_wrapper_checkEnd(...) /* check if the application is terminated or not */
```

Appendix C

Task interface

Initial functions

```
int gethostname(...)  /* get the name of the machine */
int mw_port_opensocket(...) /* open a socket for LINX communication */
MW_SPID mw_port_find_server(...) /* find the db_handler or trace_handler */
int mw_port_findAndAttach(...) /* find and attach to a task */
int mw_port_register(...) /* register to db_handler */
int mw_port_unregister(...) /* unregister from db_handler */
```

Trace functions

```
struct timeval mw_port_get_timestamp(...) /* get the current time stamp */
int mw_port_register_trace(...) /* register to trace_handler */
int mw_port_set_tracelevel(...) /* set the trace level */
int mw_port_trace_info(...)  /* info tracing */
int mw_port_trace_profile(...) /* trace the wcet and wcct */
int mw_port_set_trace(...)  /* set trace signal */
int mw_port_trace(...) /* send a trace signal */
```

Timing functions

```
int mw_port_wcet(...) /* record the execution time */
int mw_port_wcct(...) /* record the communication time */
int mw_port_period_sleep(...) /* sleep until next period starts */
int mw_port_startSync(...) /* synchronize the starting of all tasks */
int mw_port_checkPause(...) /* check if the application is paused or terminated */
```

Communication functions

```
int mw_port_setGroupReceive(...) /* set the sources of group receive */
int mw_port_setGroupSend(...) /* set the targets of group send */
int mw_port_setGroupSend_wspid(...) /* set the targets of group send by ID */
int mw_port_groupCheckReady(...) /* check if messages of group receive are ready */
int mw_port_send(...) /* send a message */
int mw_port_receive(...)  /* receive a message */
int mw_port_receive_w_tmo(...) /* receive a message with timeout */
int mw_port_groupSend(...) /* sending a group message */
int mw_port_receiveGroupRequest(...) /* receive group messages */
```

Resource managing functions

```
MW_SPID mw_port_findTarget(...) /* find a task */
MW_SPID mw_port_trade(...) /* trade for a task */
int mw_port_registerlock(...) /* register as lockable task */
int mw_port_lock(...) /* lock a task */
int mw_port_lock_wspid(...) /* lock a task with ID */
int mw_port_release(...) /* release a lock */
int mw_port_release_wspid(...) /* release a lock with ID */
```

Bibliography

[1] J. Proenza and L. Almeia, "Distributed Embedded Systems: An Introduction." University Lecture, 2008.

[2] W. Emmerich, "Distributed System Principles." University Lecture, 1997.

[3] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Realtime Systems, Springer, 2011.

[4] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. Lee & Seshia, 2011.

[5] D. C. Schmidt, A. Gokhale, R. E. Schantz, and J. P. Loyall, "Middleware R&D challenges for distributed real-time and embedded systems," *SIGBED Rev.*, vol. 1, pp. 6–12, Apr. 2004.

[6] T. Bishop and R. Karne, *A survey of middleware*. PhD thesis, Towson University, 2002.

[7] F. Picioroaga, *Scalable and efficient middleware for real-time embedded systems. a uniform open service oriented, microkernel based architecture*. PhD thesis, PhD thesis, Université Louis Pasteur, Strasbourg, 2004.

[8] W. Emmerich, *Engineering Distributed Objects*. John Wiley & Sons, 2000.

[9] T. LeBlanc and E. Markatos, "Shared memory vs. message passing in shared-memory multiprocessors," in *Parallel and Distributed Processing, 1992. Proceedings of the Fourth IEEE Symposium on*, pp. 254 –263, dec 1992.

[10] P. A. Bernstein, "Middleware: a model for distributed system services," *Communications of the ACM*, vol. 39, no. 2, pp. 86–98, 1996.

[11] W. Lamersdorf, M. Merz, and K. Müller-Jones, "Middleware support for open distributed applications," in *Proceedings of the first International Workshop on High Speed Networks and Open Distributed Platforms, St. Petersburg*, 1995.

[12] D.Schmidt, "Overview of the ACE TAO Project." http://www.dre.vanderbilt.edu/~schmidt/TAO-overview.html, Accessed: 20-02-2013.

[13] "Real-time CORBA specification," 2005. Version 1.2.

[14] J. Lawson, R. Raines, R. Baldwin, T. Hartrum, and K. Littlejohn, "Modeling adaptive middleware and its application to military tactical datalinks," in *Military Communications Conference, 2004. MILCOM 2004. 2004 IEEE*, vol. 2, pp. 975–980 Vol. 2, oct.-3 nov. 2004.

[15] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time corba event service," *ACM SIGPLAN Notices*, vol. 32, no. 10, pp. 184–200, 1997.

[16] V. Fay-Wolfe, L. DiPippo, G. Cooper, R. Johnson, P. Kortmann, and B. Thuraisingham, "Real-time CORBA," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 11, no. 10, pp. 1073–1089, 2000.

[17] W. W. Eckerson, "Three tier client/server architecture: Achieving scalability, performance, and efficiency in client server applications.," *Open Information Systems*, vol. 10, no. 1, 1995.

[18] M. Deshpande, D. C. Schmidt, C. O'Ryan, and D. Brunsch, "Design and performance of asynchronous method handling for corba," in *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, (London, UK, UK), pp. 568–586, Springer-Verlag, 2002.

[19] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time corba scheduling service," *Real-Time Systems*, vol. 20, no. 2, pp. 117–154, 2001.

[20] "Common object request broker architecture specification, part 1: CORBA interfaces," 2011. Version 3.1.1.

[21] S. Vinoski, "Corba: Integrating diverse applications within distributed heterogeneous environments," *Communications Magazine, IEEE*, vol. 35, no. 2, pp. 46–55, 1997.

[22] A. S. Gokhale and D. C. Schmidt, "Optimizing a corba internet inter-orb protocol (iiop) engine for minimal footprint embedded multimedia systems," *Selected Areas in Communications, IEEE Journal on*, vol. 17, no. 9, pp. 1673–1706, 1999.

[23] A. Gokhale and D. Schmidt, "The performance of the corba dynamic invocation interface and dynamic skeleton interface over high-speed atm networks," in *Global Telecommunications Conference, 1996. GLOBECOM '96. 'Communications: The Key to Global Prosperity*, vol. 1, pp. 50 –56 vol.1, nov 1996.

[24] R.Martin, "Introduction to Fault Tolerant CORBA." http://cnb.ociweb.com/cnb/CORBANewsBrief-200301.html, Accessed: 20-02-2013.

[25] "Fault-tolerant corba specification," 2010. Version 1.0.

[26] P. Narasimhan, L. Moser, and P. Melliar-Smith, "Strong replica consistency for fault-tolerant corba applications," in *Object-Oriented Real-Time Dependable Systems, 2001. Proceedings. Sixth International Workshop on*, pp. 10–17, IEEE, 2001.

[27] "Corba concurrency service specification," 2000. Version 1.0.

[28] N. Desai and F. Mueller, "Scalable hierarchical locking for distributed systems," *Journal of Parallel and Distributed Computing*, vol. 64, no. 6, pp. 708–724, 2004.

[29] N. Sharifimehr and S. Sadaoui, "An extended concurrency control service for corba," in *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, pp. 330 –337, march 2008.

[30] I. Pyarali, C. O'Ryan, D. Schmidt, A. Gokhale, N. Wang, and V. Kachroo, "Applying optimization principle patterns to real-time orbs," *Concurrency Magazine*, 2000.

[31] D. C. Schmidt, "Evaluating architectures for multithreaded object request brokers," *Communications of the ACM*, vol. 41, no. 10, pp. 54–60, 1998.

[32] G. AUTOSAR, "Specification of RTE," *Part of Release*, vol. 4, 2009.

[33] D. Schreiner and K. Goschka, "A component model for the autosar virtual function bus," in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 2, pp. 635 –641, july 2007.

[34] G. T. Heineman and W. T. Councill, *Component-based software engineering: putting the pieces together*, vol. 17. Addison-Wesley USA, 2001.

[35] W. Dafang, L. Shiqiang, H. Bo, Z. Guifan, and Z. Jiuyang, "Communication mechanisms on the virtual functional bus of autosar," in *Intelligent Computation Technology and Automation (ICICTA), 2010 International Conference on*, vol. 1, pp. 982–985, IEEE, 2010.

[36] H. C. Jo, S. Piao, S. R. Cho, and W. Y. Jung, "Rte template structure for autosar based embedded software platform," in *Mechtronic and Embedded Systems and Applications, 2008. MESA 2008. IEEE/ASME International Conference on*, pp. 233 –237, oct. 2008.

[37] R. Ernst, "Networks, multicore, and systems evolution - facing the timing beast," in *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, p. xviii, sept. 2008.

[38] AUTOSAR, "Specification of Timing Extensions," *Standard*, 2011. Version 1.2.0.

[39] AUTOSAR, "Software Component Template," *Part of release*, 2010. Version 3.3.0.

[40] AUTOSAR, "Requirements on Methodology," *Auxiliary*, 2011. Version 1.1.0.

[41] N. Naumann, "AUTOSAR Runtime Environment and Virtual Function Bus," 2009.

[42] J. Kim, B. G, J. Rajkumar, and M. Jochim, "An autosar-compliant automotive platform for meeting reliability and timing constraints," 2011.

[43] ENEA, "Dynamically Self-Configuring Automotive Systems," 2009. Version 1.0.0.

[44] A. Vasilakos, M. Parashar, S. Karnouskos, and W. Pedrycz, eds., *Autonomic Communication*. No. ISBN: 978-0-387-09752-7, Springer, 2010.

[45] "LINX for Linux." http://cnb.ociweb.com/cnb/CORBANewsBrief-200301.html. Version 2.5.1.

[46] M. Christofferson, "LINX: an open source IPC for distributed, multicore embedded designs," 2006.

[47] "Agile Policy-Expression-Language." http://www.policyautonomics.net/. Version 1.2.

[48] R. Anthony, "Generic support for policy-based self-adaptive systems," in *Database and Expert Systems Applications, 2006. DEXA '06. 17th International Workshop on*, pp. 108 –113, 0-0 2006.

[49] "Corba for embedded specification," 2008. Version 1.0.

[50] A. Gokhale and D. Schmidt, "Techniques for optimizing corba middleware for distributed embedded systems," in *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 513 –521 vol.2, mar 1999.

[51] "ACE+TAO Subsetting." http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/ACE/docs/ACE-subsets.html/, Accessed: 2-03-2013.

[52] "Component Based Automotive System Software." http://embsys.technikum-wien.at/projects/compass/index.php, Accessed: 2-03-2013.

[53] R. J. Anthony, D. Chen, M. Pelc, M. Persson, and M. Torngren, "Context-aware adaptation in dyscas," *Electronic Communications of the EASST*, vol. 19, 2009.

[54] "What is brake by wire." http://www.brakebywire.com/brake-by-wire.html, Accessed: 22-05-2013.

[55] "Mosquitto." http://mosquitto.org/. Version 3.1.

[56] "MQTT." http://mqtt.org/. Version 3.1.

TRITA-ICT-EX-2013:155

www.kth.se