TEKNISKA HÖGSKOLAN

HÖGSKOLAN I JÖNKÖPING

Middleware for Dynamically Self-Configuring Automotive Systems

Dung Vi

MASTER THESIS 2006

ELECTRICAL ENGINEERING

TEKNISKA HÖGSKOLAN

HÖGSKOLAN I JÖNKÖPING

Middleware for Dynamically Self-Configuring Automotive Systems

Dung Vi

Detta examensarbete är utfört vid Tekniska Högskolan i Jönköping inom ämnesområdet elektroteknik. Arbetet är ett led i teknologie magisterutbildningen med inriktning inbyggda elektronik- och datorsystem. Författaren svarar själv för framförda åsikter, slutsatser och resultat.

Handledare: Alf Johansson
Examinator: Shashi Kumar

Omfattning: 20 poäng (D-nivå)

Datum: 2007-01-24

Arkiveringsnummer:

Postadress:
Box 1026
551 11 Jönköping

Besöksadress:
Gjuterigatan 5

Telefon:
036-10 10 00

Abstract

This master thesis is a portion of the DySCAS project and work is performed at Enea AB. DySCAS (Dynamically Self-Configuring Automotive Systems) is a research project funded by the European Commission.

This thesis concentrates on future vehicle electronic systems. During a life cycle of the car vehicle manufacturers desire to upgrade or add new functions into the vehicle electronic systems, this is not possible with the static-runtime environment that employed into today's car.

To tackle this difficult problem many technologies were gathered and a dynamically self-configuring automotive system was introduced by combining technologies like self-managing, service-based and middleware.

The obtained results fulfilled most of DySCAS requirements. However, the system has a few limitations and these are caused by the immature of distributed reconfigurable embedded systems in the market.

Sammanfattning

Den här magisterexamen är en del av DySCAS projekt och arbetet har bedrivit på Enea AB. DySCAS (Dynamically Self-Configuring Automotive Systems) är en forskning projekt finansierad av Europa Kommissionen.

Den här magisterexamen koncentrerar sig på framtida fordon elektronik system. Under livstiden av ett fordon vill fordonstillverkarna kunna uppgradera eller lägga till nya funktioner i systemet. Detta är inte möjligt med dagens bilar som är statiskt konfigurerad.

Många teknologier har samlat för att tackla detta svåra problem. Ett dynamiskt konfigurerbar system har utvecklat genom att kombinera självhanterande, tjänster baserad och mellanprogramvara.

Resultatet uppfyller de flesta kriterier för ett dynamiskt konfigurerbar system. Men systemet har några begränsningar och detta beror på en omogen marknad för distribuerad konfigurerbar inbyggda system.

.

Acknowledgements

Many thanks to Detlef and Barbro for their efforts, also thank to Mr Homuth for a full version of TASKING embedded development environment that made the development possible and thank to Alf Önnestam for helped me with OSE epsilon.

Also thanks to Martin Tiselius, Magnus Gille, Mattias Säteri, Dan Jerrestam, Ola Redell and Joel Axelsson for helped me with various things.

Key words

Self-managing systems

Middleware

Link handler

CAN protocol

OSE Epsilon

Vehicle electronic systems

Registry

Policy-based

Service-based

Deterministic

Fault-tolerance

Flexibility

Scalability

Robust

Portability

Openness

Contents

List of Figures

List of Tables

List of Abbreviations

DySCAS	Dynamically Self-Configuring Automotive Systems
OSE	Operating System Embedded
ABS	Anti-lock Breaking System
CAN	Controller Area Network
CPU	Central Processing Unit
ECU	Electronic Control Unit
LRM	Local Resource Manager
CORBA	Common Object Request Broker Architecture
RT	Real-Time
TTP	Time Triggered Protocol

1 Introduction

DySCAS (Dynamically Self-Configuring Automotive Systems) is a research project funded by the European Commission. DySCAS started at June 2006 and will end in May 2008. This thesis is a portion of the DySCAS project and work is performed at Enea AB.

In the later years, the processing power has increased and this leads to that many automakers migrated functions from the hardware side to the software side. Automotive systems nowadays are complex distributed systems and future applications that will include simultaneous access to a number of devices [1]. These imposed high demand on the network capabilities and the dynamic reconfiguration of the system. A dynamic reconfigurable system is a system that will adapt its operations due to the workloads, demands and conditions.

Future automotive systems are required efficient middleware to handle the complexities added into the devices. The goal of middleware is to hide the complexities of the underlying operations from the applications including the network communications. The benefits are reliable automotive communications (numbers of devices are reduced), lower development costs, easier to maintain and comforts for the end-users.

Enea AB is world leader in embedded device software and advance systems development. The company has more than 350 professional engineers that integrate and develop advanced systems for the customers.

The company is also leading in supplier of operating systems for telecommunications industry. Millions of devices worldwide are running OSE operating systems. OSE operating systems were introduced 1984 and has become the most tested and trusted real-time operating systems.

1.1 Background to the project

Embedded electronic systems for vehicles are facing difficulties, in-car electronics have grown rapidly the recent years and it is still growing. Examples of such devices are ABS, gear control, light control, air conditioning, airbags, engine management, active suspension and central locking. All new devices that get added into the vehicle electronic systems led to bigger material cost, longer production time, more complexity and less reliability for automotive communication systems.

The static-runtime environment that employed into today's vehicle electronic systems will be a problem in the future. With static-runtime environment that means the system is defined in the beginning of the development process. During a life cycle of the car, the vehicle manufacturers are desired to upgrade or add new functions into the vehicle electronic systems. This is not possible with static-runtime environments and future vehicle electronic systems impose high demand on dynamic reconfiguration.

The advantages with this kind of systems are easier to upgrade, functionality and comforts for the users.

1.2 DySCAS overview and thesis objectives

This master thesis is part of DySCAS start-up phase. The task is cooperated with two other students to develop a dynamically self-configuring middleware for DySCAS.

1.2.1 DySCAS requirements

Future use-cases require vehicle electronic systems to be more flexible and scalable. Two major automotive manufacturers [2] have supplied the use-cases. Below is the list of the use-cases and the defined characteristics that the system shall have.

Scalability – Devices can be added or removed during runtime. Typical are portable devices are added/removed from the system. (DySCAS use-case).

Flexibility – Functions are scaled in/out depends on the environment (short-lived functions). Vehicle arrived to a hot-spot and software became downloadable (DySCAS use-case).

Portability – The system should be executable on a variety of operating systems and platforms. There are many different Electronic Control Units (ECU) interconnected in a vehicle electronic system.

Openness – Open for third-part developers to use the middleware interface, faster product to market for third-part developers because the system knowledge is not required.

Robust – The system shall detect hardware and software failures, functions are made available depended on the standing of the system (DySCAS use-case). This is often referred to graceful degradation[1].

Distributed – The information is shared between ECU (Electronic Control Unit). The user browsed and played music stores on the portable devices (DySCAS use-case).

1.2.2 Master thesis objectives

The intention of this master thesis is to develop a middleware for DySCAS platform with respect to the hardware. This thesis followed a bottom-up methodology [2]and focused more on automotive bus than the self-configuring architecture. Below is the list of the requirements for this thesis.

In a vehicle electronic system, devices are interconnected with each other using some kind of bus. The task is to specify a CAN protocol for the link handler. The protocol is used by the devices in the communication network.

To deploy a link handler using the specified CAN protocol. The link handler shall hide the complexities of the network operations. It shall also detect when nodes are added, removed or failed.

- The link handler shall send/receive signals larger than 8 bytes (fragmentation and assembly of large signals because each CAN message can transfer maximum 8 bytes)

[1] When an ECU in the network failed the system inactivated its functions

[2] With bottom-up methodology means that the system is divided into different layers. The applications layer is placed at the top and the hardware layer is placed at the bottom. Work is performed with the lowest layer first and upward.

- The link handler shall support Little Endian and Big Endian processors
- The link handler shall supervise nodes in the network
- There must be some priority mechanism for the signals (real-time requirement)

The middleware for DySCAS shall have the characteristics listed below.

- In a vehicle electronic system there are many different operating systems running on different processors, the middleware shall be effortless to port to various platforms. (DySCAS requirement of portability)
- To simplify application developments for third-part developers a common middleware interface shall be deployed. The system complexities are completely hidden from the applications. (DySCAS requirement of openness)
- The middleware shall be scalable. This can be solved by separate the link handler from other middleware modules (DySCAS requirement of scalability).

To deploy the demonstration applications for the DySCAS system, the applications shall use DySCAS middleware to communicate with each other.

Have some mechanisms to observe and monitor the activities in the bus.

1.3 The master thesis limitations

There are a few limitations in this thesis because the thesis is scheduled for only 20 weeks.

The link handler will be implemented on OSE epsilon, but the design should be capable to execute on a variety operating systems.

The CAN bus will be used to interconnect devices. However, there is a wish to add an extra bus to the communication system. The main reason for that is to visualize a car's network.

1.4 Thesis outline

This master thesis followed a fix design (a survey) methodology. In the chapter 1 the problem was formulated and the goals ware specified. In chapter 2 technologies related to the subject were gathered to deploy the design. In chapter 3 the design implementation was realized. In the last chapter the results were analysed and the conclusions were made.

2 Technologies for vehicle electronic systems

The goal of this project is to develop an architecture for vehicle electronic systems that is scalable, flexible, portable and open. To accomplish the task technologies such as self-managing systems, middleware architectures and automotive communications were studied. The problem when combined technologies from different areas are incompatibilities because different technologies were developed for different purposes.

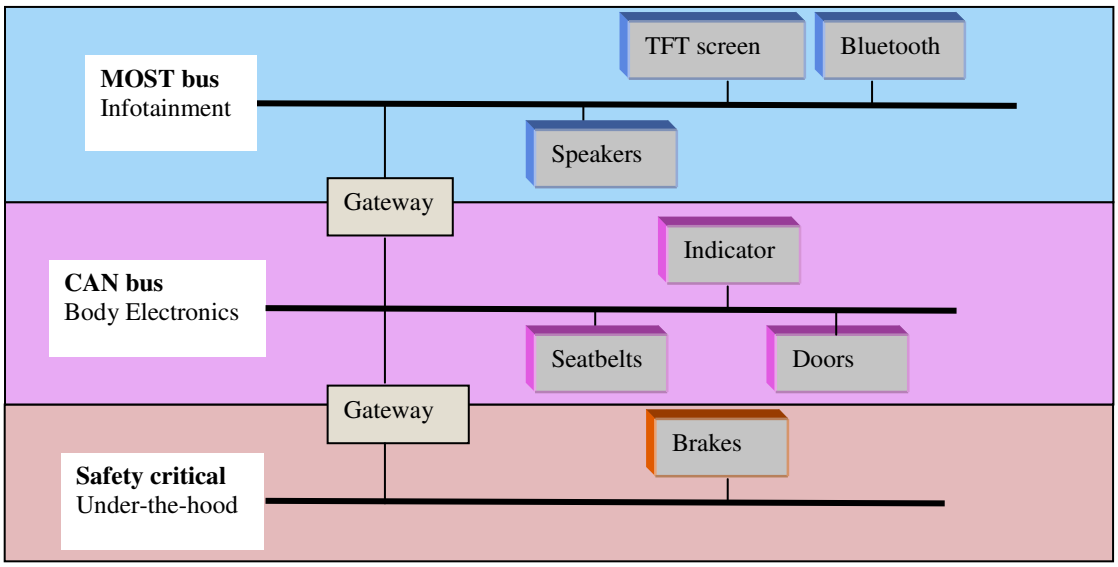

Figure 2.1: Car's network is divided into three layers

2.1 Self-managing systems

Nowadays embedded systems are commonly configured offline. Control software is stopped when new software configurations are installed onto embedded devices. In some cases stopping control systems could cause fatal systems failures. Engineers must master technologies to be able to build the systems of the future and engineers need to use the self-managing concepts to master the increased complexity of the embedded systems. Engineers are now becoming increasingly aware of the need of self-managing systems.

Many self-managing systems are developed to free up the requirement of system administrators. Most self-managing systems are policy-based systems. Policies are rules for the system to follows in a certain situation. Policies are separated from the system and because of that policy technique offers dynamic reconfiguration. This means changed policies do not need re-compilation of codes. There are three distinct approaches using policies, the static policy configuration approach, the open-loop policy adaptation approach and the close-loop policy adaptation approach [3].

The static policy configuration is the simplest approach. This means that the policies are embedded and fixed. The policies are not modifiable at runtime. To modify the policies the system has to bring offline.

The open-loop policy adaptation is when policies are modifiable between executions. The system is capable to identify the policy conflicts or identify new policies. However, users need to manually update the policies.

The close-loop policy adaptation is when policies automatically adapt its rules depending on the standing of the system. The system detects policy conflicts and automatically updates the policies. This sort of systems requires a policy server.

A policy server is where policies are kept, elements in the system are supposed to update their policies using the policy server. When new policies are discovered the policy server will roll out update information to elements in the system. When an element is started-up it automatically connects to the policy server to receive policies to follow.

Self-managing systems adapt themselves to changes in the environment. The system monitors its standing and performs environment tests to ensure that it is working properly. A self-managing system is often divided into four sub-domains [4]. These sub-domains are self-configuration, self-optimization, self-healing and self-protection.

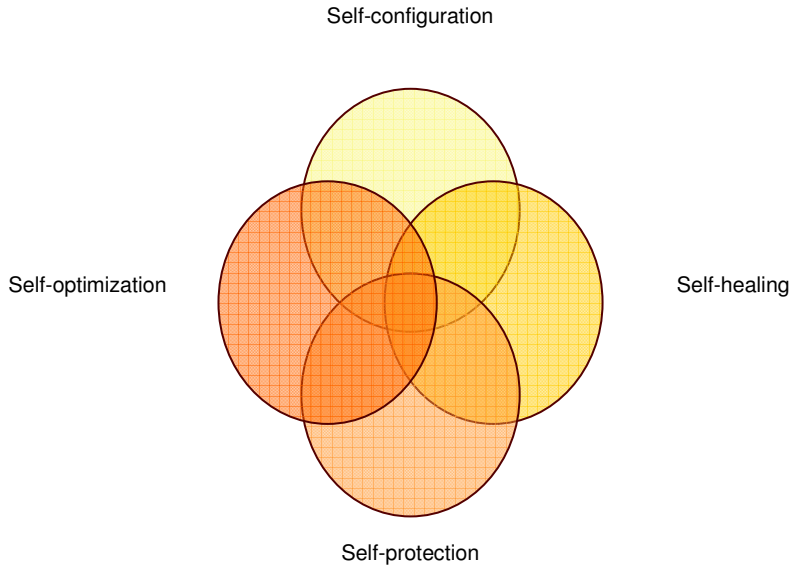

Figure 2.2: Domain of a self-managing system

Self-configuration - This is the important part of a self-managing system, when an element started up, it integrates itself seamlessly with the system and the rest of the system will adapt to its presence. The element registers itself and its capacities so that other elements in the system can either use it or modify their behaviour to it. During runtime changed in the system environment will lead to reconfiguration of the system.

Self-optimization - Self-optimization is related to self-optimization of each element. However, good behaviour of an element does not necessary be good behaviour of the system. Each element is optimized by hardware or software. In software typical optimizations are functions, algorithms or protocols. Examples of optimizations in hardware are storage or processor power. A typical system optimization is bandwidth. There is often a trade-off between cost and performance.

Self-protection - The system protects itself from external harms and internal harms. Internal harms are software failures, architecture failures or heavy loaded operations. Typical external harms are intrusions or electrical magnetic noises that force onto the system.

Self-healing – The time for a programmer to diagnose and fix a serious problem can take weeks. The system monitors its behaviours to discover the root causes and prevent it against failures. It diagnoses and repairs the discovered problems. The system as a whole should be capable to continue even if parts of it are not functioning as they should.

For a distributed embedded system to be capable to reconfigure itself, self-managing elements are needed. Each element must automatic manage its own internal behaviour with respect to the external signals and messages from other elements. A registry provides mechanisms for the elements to publish their services or subscript other element's services. When an element wants to find a service, it first contacts the registry to locate where the requested service is located on the network. The self-managing elements will discover each other to identify the communication channels and coordinate with each other to archive their mutual goals.

When an element has agreed to provide service to another element, there is a relationship between these two elements. Relationships are formed at runtime and not predefined at the deployment [5]. A self-managing system is formed as a result of relationships between elements in the network.

Self-managing system reconfigures itself to the new number of devices connected in the network. These self-managing systems can be used to fulfil the scalability and flexibility requirements that DySCAS system is required.

2.2 Middleware

To tackle the complexity in distributed embedded electronic systems middleware was introduced. Middleware creates transparency and manages the underlying operations of the system. Middleware provides common interfaces for the application. It hides the complexity of transport operations that are not necessary when developing the applications.

The term middleware is very difficult to define because opinions differ depending on the adaptation environments, but it is refer to be the software layer between the operating system, including the network communication protocols and the distributed applications that interact via the network [6].

Today middleware likes CORBA and COM are widely used for distributed object computing. But lately, middleware approach using Java homogeneous programming environment language has received much attention. There are many middleware available for distributed systems [7], but the main drawback is that they were not developed for distributed real-time embedded systems. Examples of drawbacks are the packet sizes, the memory requirements and latencies. A table comparing a few middleware systems for distributed systems is showed below.

Table 2.1: Comparison between various Middleware Systems

Middleware	Openness	Scalability	Predictability	Protocol	Transparency	Detect Connection Failures
RT-CORBA	Y	Y	Y	TCP/IP	Y	Y
J2EE	Y	Y	N	TCP/IP	Y	N
COM+	Y	Y	N	TCP/IP	Y	N
OSE-MW	Y	Y	Y	TCP/IP	Y	Y

To be able to select the right middleware for DySCAS platform the vehicle electronic system requirements must be fulfilled [8]. These requirements are time critical, embedded, fault-tolerant, distributed, bandwidth, flexibility and security. Below are the short-terms of the requirements, more information about automotive buses is available in chapter 2.4.

Time critical – A communication system provides guarantees in term of timeliness. It is possible to calculate the maximum transmission time of the messages. Examples of devices that need to be time critical are ABS, engine management and gear control.

Embedded – Many embedded systems are executed on platform with limited of resources. Examples of limitation are memory, bandwidth and CPU power.

Fault-tolerant – When the system does not behave as its specifications it is caused by faults, errors and failures. The system must be tolerated against hardware and software failures. Examples are defective circuits, line failures and architecture failures.

Distributed – Software components are often distributed, several interconnected devices in a vehicle electronic system. Examples are lights, mirrors and seatbelts.

Bandwidth – In many case high bandwidth is required in automotive systems. The trade-off between required bandwidth and cost of such bandwidth must be made.

Flexibility – The devices can be scaled in/out the network, in Time Division Multiple Access networks all message transmissions must be predetermined offline, while in Carrier Sense Multiple Access networks message transmissions are resolved online. The later is often considered more flexible than the former.

Security – Communication is reachable from outside the automotive system but unauthorized accesses are not possible.

These requirements led to that only RT-CORBA and OSE middleware can be used, because they offer deterministic communications. Therefore, many basic functions from RT-CORBA and OSE middleware will influence onto the DySCAS architecture.

Different middleware architectures have different communication approaches, like direct message passing, mailbox passing and publish/subscribe message passing. Because technologies are collected from different areas and they have different architectures or techniques the deployment of a dynamically self-configuring automotive system is very complex and tricky. Traditional middleware architectures have strived to achieve transparency, to isolate applications from underlying hardware and software changes.

To simplify the design, middleware is divided into two modules. The first module is to provide the application layer with common interface and manages the resources in each element. The second module manages the network communications (the link handler).

The link handler is most important in a distributed embedded network communication. It provides mechanisms for the information to travel from sources to destinations. The link handler acts like a post-office, deliver the letters to the right addresses.

Because middleware is a layer between the operating system and the applications it is feasible to port to other operating systems and platforms (portability requirement). The middleware consists of modules which will also solve the scalability requirement.

2.3 The Link Handler

Distribution transparency is beneficial and necessary for programming distributed applications. In a fully distributed embedded system, microprocessors are physically distributed and this must be hidden from the applications. Traditional operating systems do not support communication from an internal process to a process residing on an external node. Instead, the link handler provides support for such communication.

With transparent method that means there is no difference in sending data to a process in the local node or to a process in an external node. The sending process does not know whether that the destination process is residing on the external node or in the same node. Figure 2 shows that the signals physical path which consists of many bridges compared to the virtual path that the sending process understands.

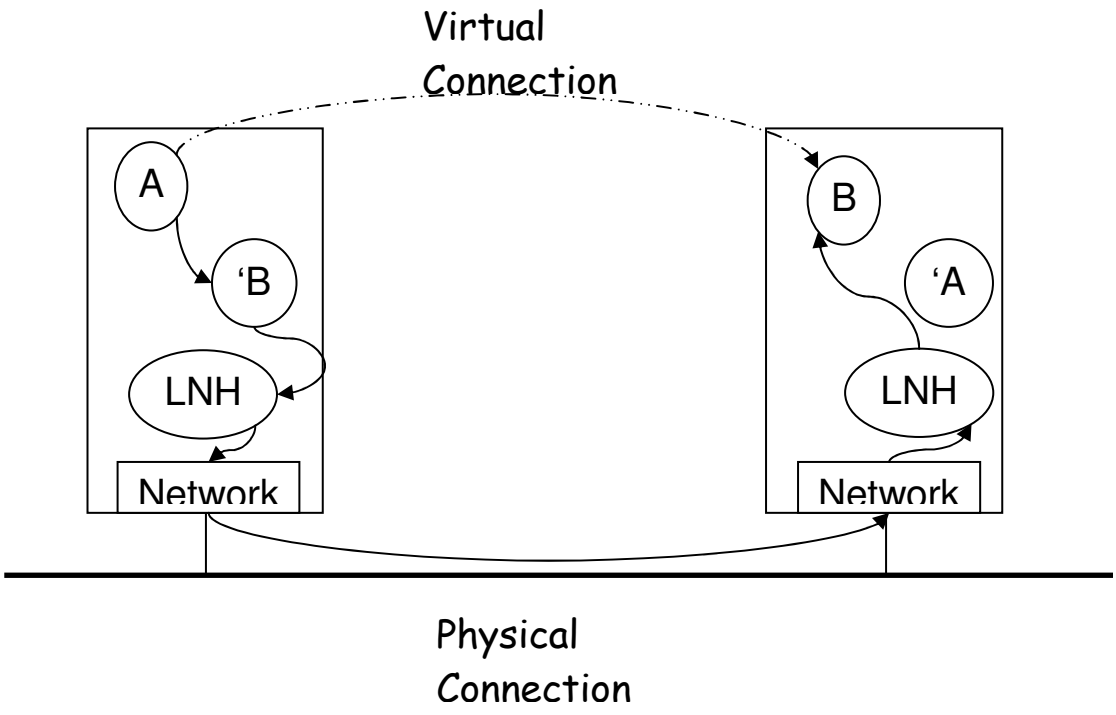

Figure 2.3: Transparency

The link handler also supervises the communication channels to identify communication failures. In a dynamic distributed embedded system nodes are added or removed at runtime and these must be detected. Detection of failures provides robustness for the communication system. There can be three kinds of failure in a distributed

communication environment, the process failure, the board failure and the network failure.

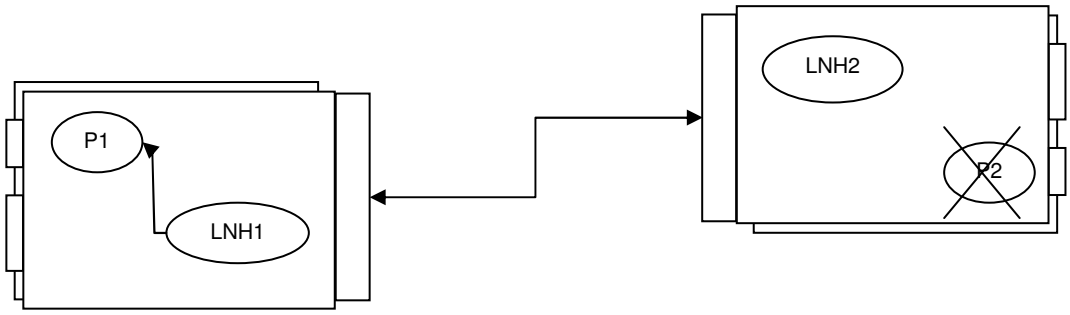

Figure 2.4: Process failure

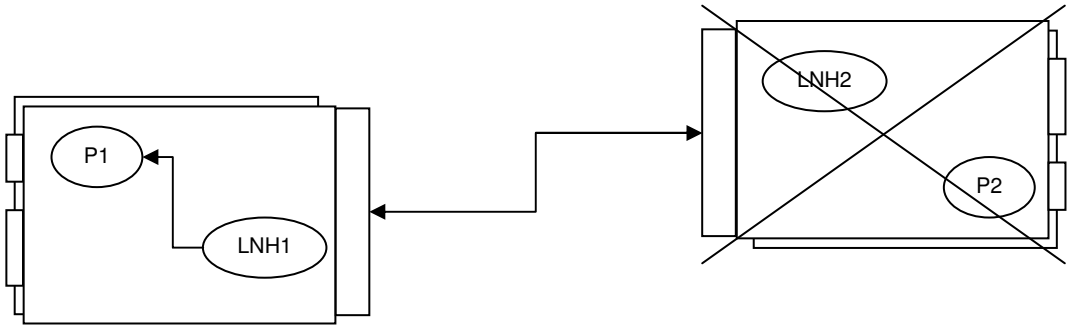

Figure 2.5: Board failure

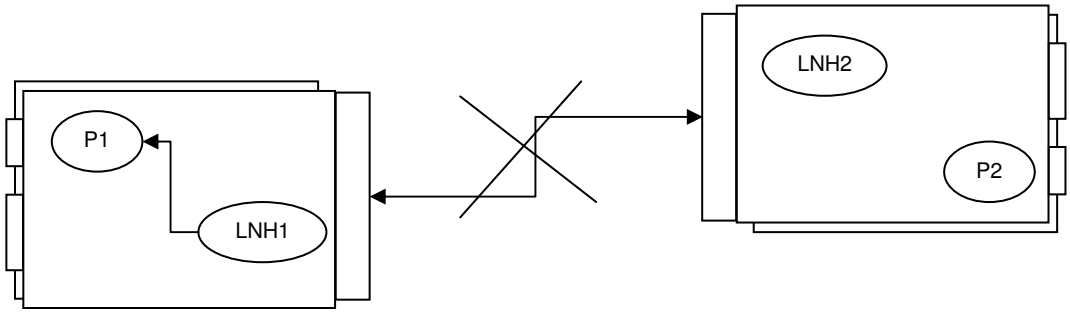

Figure 2.6: Network failure

The process failure is when the board is active but the process is unreachable. This is often caused by failed process or terminated process. The board failure is when the board is unreachable. This is not the same as network failure. In network failure is when the board can not communicate with any boards at all in the network.

Enea AB is provided two types of link handler, the link handler for OSE Delta and a link handler called LINX, which supports a variety of operating systems.

The link handler in OSE Delta which consists of three layers, the general link handler (GLH), the generic protocol link handler (GPL) and the protocol driver (PDR).

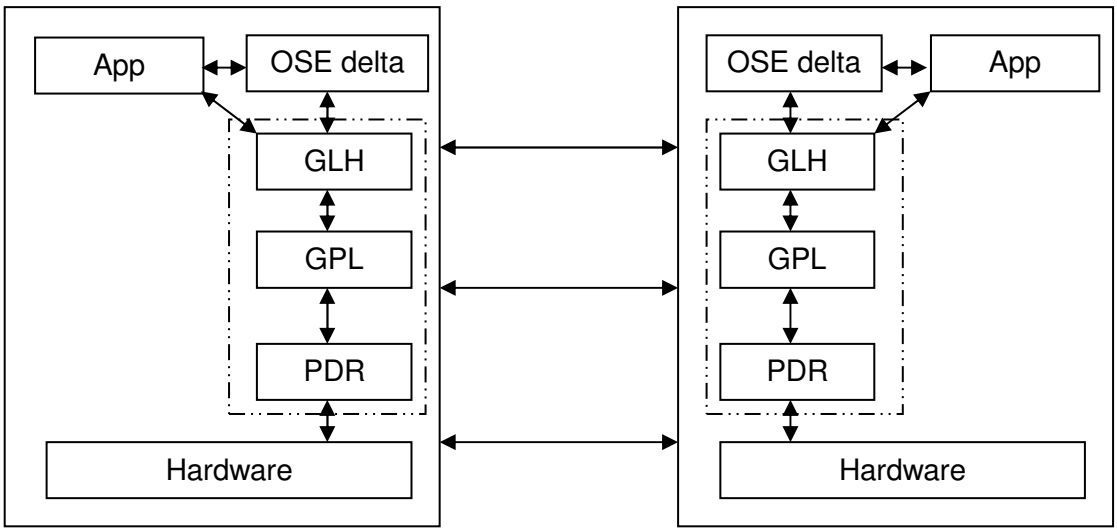

Figure 2.7: The OSE Delta link handler

The GLH is closest to the operating system and designed to be as general as possible. It integrates itself with the OSE kernel and creates two types of phantom processes[3], phantom link handlers and phantom processes. The phantom link handlers are mirrors of remote connected link handlers and phantom processes are processes on the remote link handler that have attached.

The GPL is used to make a reliable connection to the remote node. The GPL automatic detects when connection problems raised and notified the GLH. It is also responsible for fragmentation and re-assembly of signals that are too big to send as one packet.

The PDR is closest to the network. It is responsible for transporting the package over the physical layer. The PDR does not need to detect network nor system failures. It does not construct a protocol for requesting or re-sending the packages. All these are handled by the GPL. It is also possible to custom-made the PDR layer and GPL layer if the transfer medium is not supported by OSE.

The second link handler that is platform and interconnection independent, this link handler is called LINX. The LINX protocol stack has two layers, the rapid link handler (RLNH) and the Connection Manager.

[3] A phantom process is a process that redirect the incoming signals to a other process

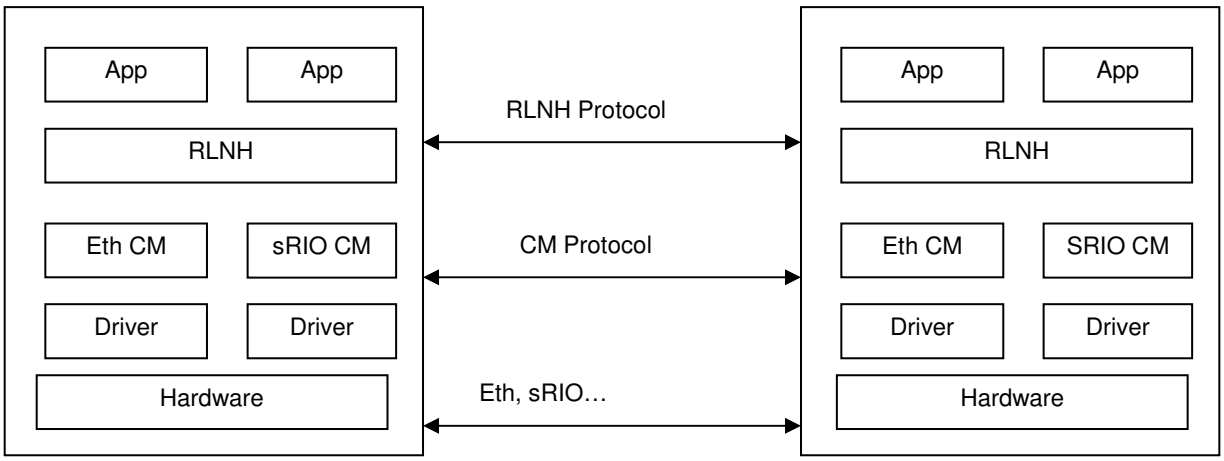

Figure 2.8: LINX layers

The RLNH provides link management, supervision, address publication and resolution. The RLNH is a layer above the Connection Manager, which makes it connection medium independence.

The Connection Manager is initialized by the RLNH. It provides a reliable transport including retransmission and fragmentation of the messages if necessary.

2.4 Automotive buses

Normally it takes more than ten years to develop a car. Technologies and functions implemented in the car are decided in the beginning of the development phase. Many times at the middle of the development process new technologies/functions appear and these technologies/functions are not possible to adapt into the car. The background of DySCAS project is to replace the static-runtime environment that is implemented into today vehicle electronic system with dynamically self-configuring run-time. The dynamically self-configuring system will allow new functions to easily be added into the system.

To be able to test and implement the middleware architecture buses that are flexible are needed. Therefore, buses that visualize a vehicle electronic network are required. In a vehicle electronic system there are many subsystems and these subsystems can be dividing into three different layers. The infotainment layer which required high bandwidth but does not necessary be time-critical, the body electronics layer which does not require as high bandwidth as infotainment layer but some devices are required to be time-critical and last the safety critical layer which requires hard real-time. This is because one error can cause catastrophically consequences.

Many buses have compared with each other, their benefits and disadvantages are used to select the right bus [9] for the DySCAS system. The goal is to develop a flexible and reconfigurable system for infotainment layer, but still, the system should be capable to fulfil the body electronics layer requirements. Therefore, DySCAS system is interested to be executable on both infotainment layer and body electronics layer.

Table 2.2: Buses

1394b	3.2 Gbps	**High speed**
IDB-1394		
1394a	400 Mbps	
MOST	45 Mbps	
TTp	25 Mbps	
Flexray/bytefly	10 Mbps	
TTCAN	1Mbps	
CAN	1Mbps - 50 Kbps	
Safe-by-Wire	150 Kbps	
LIN	<20 Kbps	**Low speed**

Typical buses for body electronics are TTP, TTCAN, LIN, CAN and FlexRay. Buses for infotainment are MOST, IDB-1943, D2B and bluetooth. Comparison between buses on body electronics layer is done in the next section *2.4.1 time-triggered vs. event-triggered.*

2.4.1 Time-triggered vs. Event-triggered

There are two types of bus for embedded systems, non real-time and real-time. In a real-time system only deterministic communication networks can be used. Deterministic networks are networks that are possible to calculate response times, latencies and deadlines. Typical deterministic networks are CAN, TTCAN, FlexRay and TTP. These buses can be of time-triggered or event-triggered.

Time-triggered buses behave like cyclic scheduling of tasks. A node is allowed to send at a fixed time (a timeslot). The sending is done in cycle-round. It is easy to determine the latencies of the packets and the buses are very often used in safety critical distributed embedded systems. Examples of time-triggered buses are FlexRay, Byteflight TTP and TTCAN.

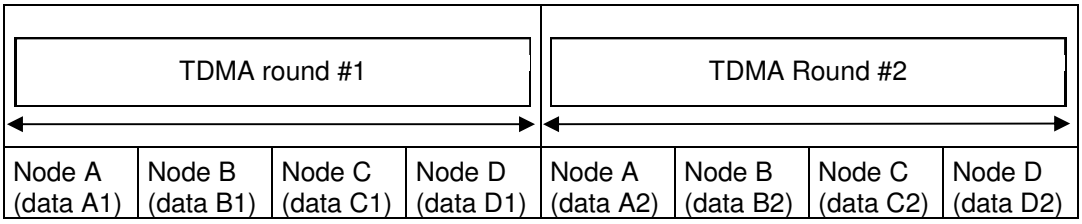

Figure 2.9: Time-triggered

Tests have done with Byteflight in article [10] and the authors concluded that FlexRay (Byteflight and FlexRay almost the same) is the new generation high-performance network. Nevertheless, buses for the future automotive systems have to be interconnected and not all nodes require high bandwidth. Even if FlexRay and TTP have higher bandwidth, this does not mean they are suitable for all systems.

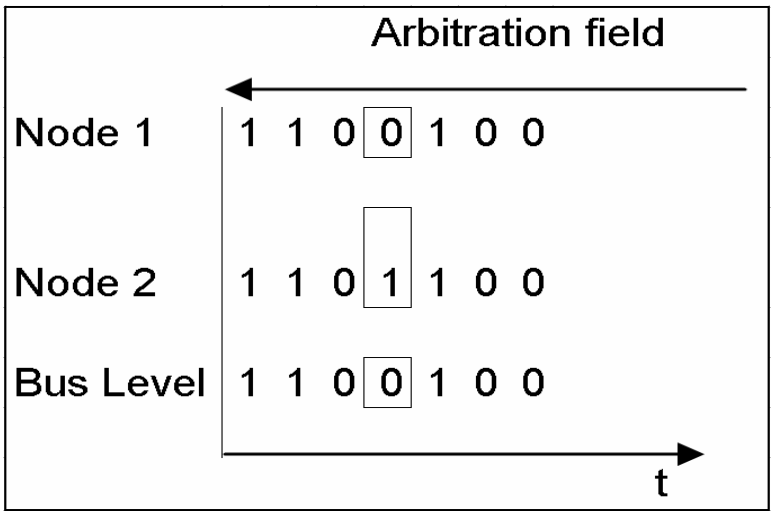

Figure 2.10: Event-triggered

In event-triggered networks every message has its identification. The identification is unique and it is the priority when the node wishes to obtain the bus. Message of higher priority will always win the arbitration. The main drawback with event-triggered buses is the large jitter and CAN is a typical event-triggered bus.

In the CAN bus the number of nodes may change dynamically without disturbing the communications of the other nodes. The bus has also advanced error handling to reduce impact from noises onto the bus. The bus is serial communications protocol CSMA/CR. This means that collision of messages is avoided by bitwise arbitration without loss of time.

Comparing these two different networks, time-triggered has smaller jitter, but it is hard to maintain and reconfiguration is needed whenever nodes are added/removed. Because of that time-triggered network is not suitable for the DySCAS system. Event-triggered buses however have large jitter but it is easy to develop, easy to maintain and have several implementation methods which are very suitable for reconfigurable systems.

To improve the embedded control devices like engine management systems, ABS, gear control, airbag and A/C. It is necessary to be able to upgrade the devices via network. Many well-known devices support programming via CAN and this is a necessary feature for future vehicle electronic systems [11].

2.4.2 CAN protocols for distributed embedded systems

To meet DySCAS requirements only deterministic networks are usable. CAN bus is widely used in industrial automation and has high reliability. Moreover, the bus is well established in car networks which are very suitable for DySCAS to visualize a vehicle electronic system.

The CAN bus is standard ISO 11898 and the bus logic used a "WIRED-AND" mechanism. The logical 0 is dominant and the logical 1 is recessive. This means that logical 0 wins over logical 1.

The bus consists of 5 different frames. These frames are the data frame, remote frame, error frame, overload frame and interframe space. The data frame is generated by a node when node wishes to transmit data. The remote frame is generated by a node to

request the data from the source. The error frame is generated by any node that detects a bus error. The overload frame has the same format as active error frame. Overload frame can only be generated during interframe space. The interframe space separates a proceeding frame from a following data or remote frame.

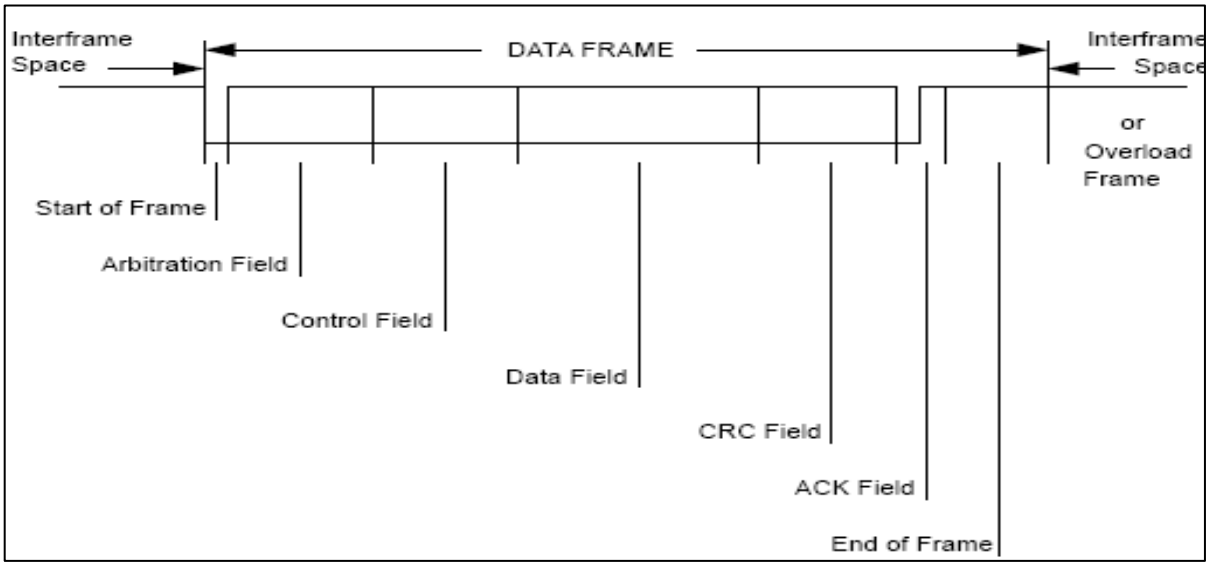

Figure 2.11: The DATA frame

The most important frame is the data frame. The frame starts by sending the most significant bit of the ID. All nodes are synchronized at the beginning of each message with first falling edge. This is called start of frame (SOF) and it is only allowed when the bus is idle.

After the SOF the arbitration begins, the lowest identification which is the highest priority will be allowed to obtain the bus. The following is the control field which consists of 6 bits. The data field which sends after the control field contain from 0 to 8 bytes, each byte contains 8 bits and the MSB is transferred first.

The CRC field contains the CRC Sequence followed by the CRC Delimiter which consists of a single recessive bit sent after the data field and finally, the ACK field which only consists of two bits, the ACK slot and ACK Delimiter.

The transmitter will send two recessive bits in the ACK field and if the receiver received a valid message it will report in the ACK slot with a dominant bit and a recessive bit in the ACK Delimiter.

There are two variants of the CAN protocol [12], the standard CAN (CAN 2.0A) and the extended CAN (CAN 2.0B). The standard CAN is 11-bit message identifications and the extended CAN is 29-bit message identifications.

Many CAN protocols for distributed systems are resided on top of the CAN protocol. It makes extensive use of the CAN identifier [13], [14] and [15]. Examples of using the arbitration are highlighted below.

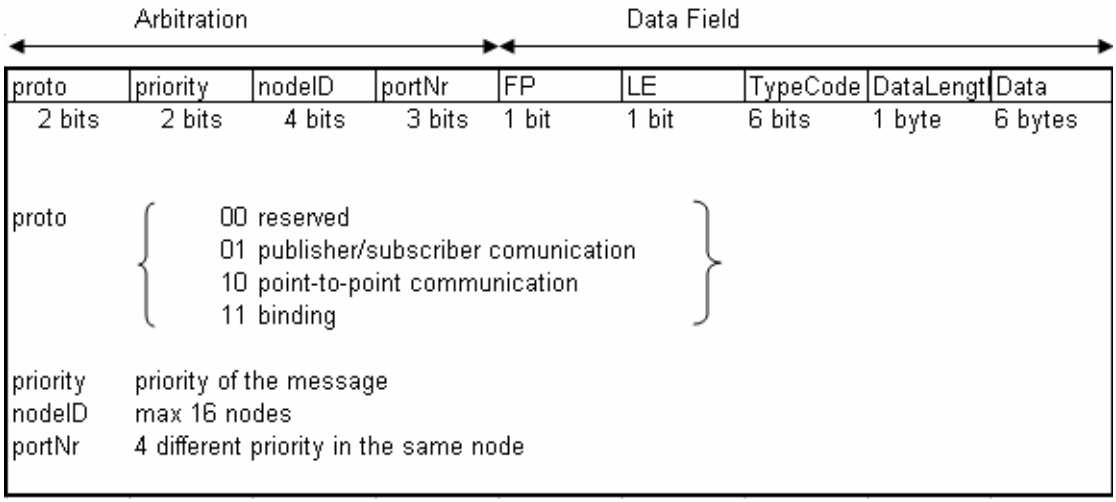

Figure 2.12: CAN for RT-CORBA

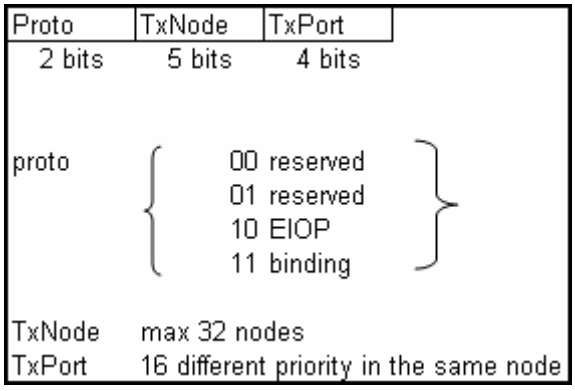

Figure 2.13: CAN for inter-ORB

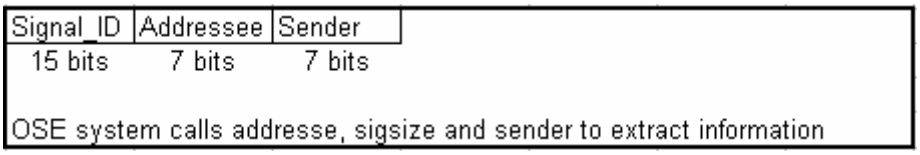

Figure 2.14: CAN for OSE

As shown the CAN identifiers are very flexible, it was used to store the information like the destinations, sources and data types. Even part of the data field is used to store information that the do not fit in the arbitration.

3 Design & Implementation

3.1 The proposal approach

A theoretical approach had sketched up for DySCAS system. The approach consists of three technologies: policy-based, service-based and middleware.

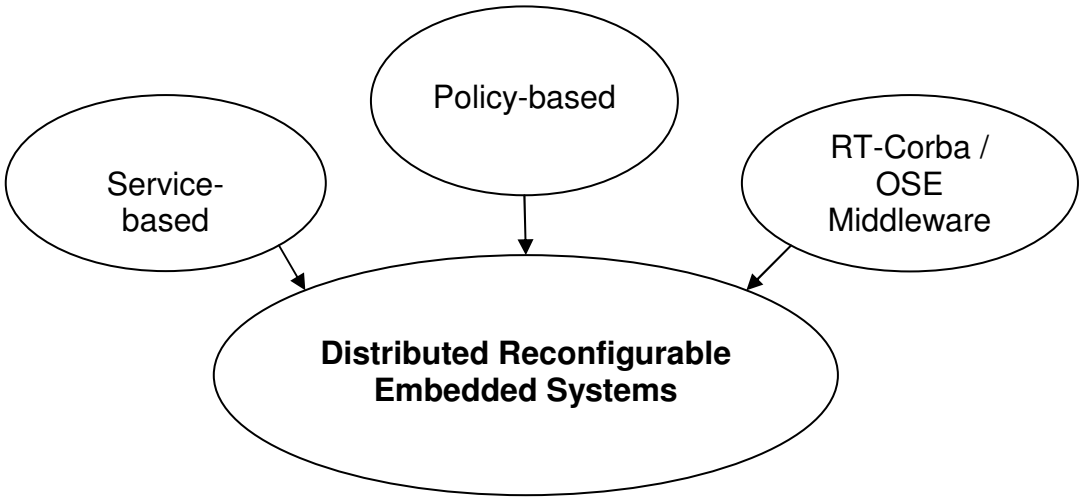

Figure 3.1: The proposed approach

To fulfil DySCAS requirements technologies relevant to the subject where gathered and compared with each other, their strengths and weaknesses are used for selection.

The benefit with policies is that changes do not need re-compilation of code. By adding policy-based methodology in DySCAS system, the system can easily change its behaviours. Future automotive system required devices to change their characteristics. By using a policy server to store policies future system will be easily to update because policies are located at one place. The policy server will be used to update devices on the system whenever new policies are added.

There are two types of decomposition, component-based and service-based. These two types of development technologies are very commonly used to design large distributed systems. A component can be visualized as a black box. It has specified interfaces that are independence of the internal operations. The benefits with component-based architectures are that the components themselves may be written in different kind of programming languages and run on different kind of platforms.

Service-based architectures are considered to be more flexible comparing to component-based architectures, because services on a service-based architecture can be loaded on demand and removed afterwards. Future vehicle electronic systems require flexibility and scalability. These requirements are fulfilled by using a service-based architecture.

To fully complete the system there is a need of an efficient middleware to hide the complexity of the system. As highlighted in *chapter 2.2 middleware* only RT-CORBA and OSE middleware support distributed real-time systems.

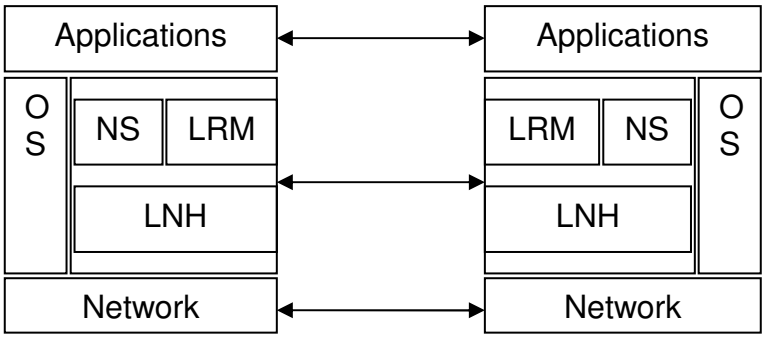

Figure 3.2: The design

The DySCAS middleware architecture was developed according the proposal approach. There is a link handler provides transparency and interconnected nodes in the network. The link handler task is to hide all the underlying network operations. It also broke up and united large signals that can not send as one message.

In the architecture there is a registry (Name Service in CORBA or Name Server in OSE) provided mechanisms for applications/services to find each other in the network. The registry allows applications/services to subscribe services in the system. Subscriptions mean that the registry will update the applications/services whenever the subscribed services are registered or removed.

Lastly, a local resource manager (LRM) provides and manages the resources on the node. All service registrations in done through the LRM.

3.2 The middleware implementation

There are two development environments available for OSE Epsilon, TASKING embedded development environment (EDE) and KEIL development tools. A full version of TASKING EDE was supplied by Mr Homuth at TASKING. Because of that all codes were developed with TASKING EDE.

3.2.1 The old platform

The implementation started out with an experimental platform. Devices on the platform are replaced, added or removed to fit the needs. The experimental platform consists of six Phytec miniMODUL boards running OSE Epsilon. The boards are connected with each other using a CAN bus. The experimental platform also contains two headlights, two indicators and three switches. These are used to visualize a car. The lights and switches are physically connected to the slots according *figure 17*.

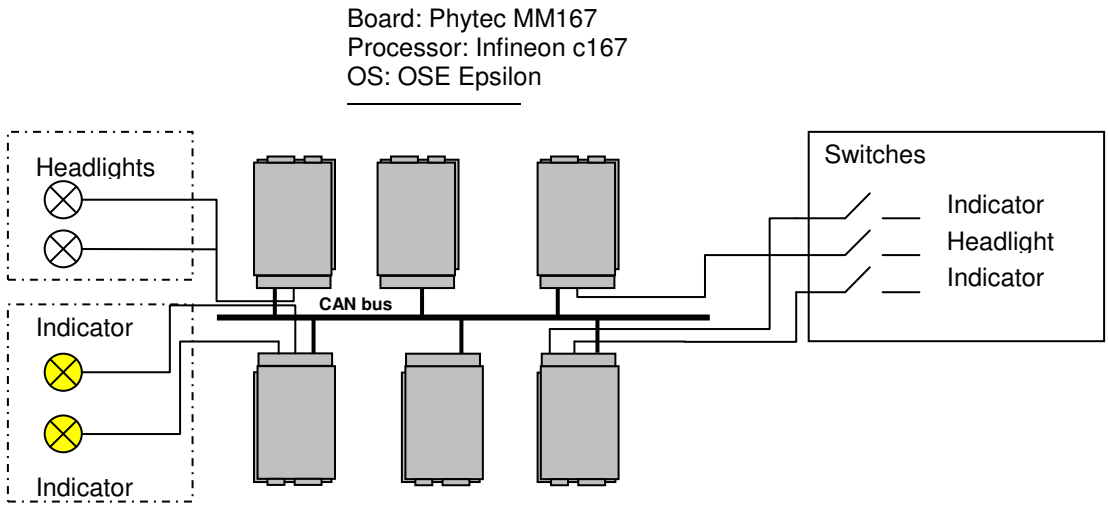

Figure 3.3: Previous development platform

In vehicle electronic systems CAN bus is widely used. The CAN bus truly envisioned the body electronics layer. There is also a desire to add a MOST bus to the car's network to visualize an infotainment layer. The MOST bus will be used to stream music, videos or for those devices that required high bandwidth. Unfortunately, due to the extended CAN version that the system used, there were no suitable CAN-MOST gateways available in the market. Resulted of that, the MOST bus was not added into the experimental platform.

The purpose of the experimental platform is to show that the system automatically adapts itself when the running environment changed. There are two types of node in the platform.

The master node type – there are two states for a master node type: The master state and the passive master state.

The master state is when the node is the master over the bus; this means it has the highest priority in the network.

The passive master state is when there is a node with higher priority in the bus; this means it is a slave but will take over the bus if it has highest priority.

The slave node type – the slave node type can never be master. The slave nodes are nodes serving the input and output interfaces.

Slave nodes should therefore be connected to slot 1, 2, 3 and 4. In the *figure 17* the headlights are connecting to the slot 2, the indicators are connecting to slot 1, the indicator switches are connecting to slot 3 and the headlights switch is connecting to the slot 4.

The master node type is not depending on the hardware, it does not read the input or the output. These two node types are introduced to visualize node with functionalities of different node. The conditions for the services to function correctly when nodes are added or removed are as follow:

The indicators: there must be at least one master node type and two slave nodes in the system, the slave nodes must be connect to slot 1 and slot 3.

The headlights: there must be at least on master node type and two slave nodes in the systems, the slave nodes must be connect to slot 2 and slot 4.

Both the indicators and the headlights: there must be at least one master node type and 4 slave nodes, the slave nodes must be connect to slot 1, 2, 3 and 4.

3.2.2 The new platform

To highlight the independence of the operating systems and independence of hardware a Phytec miniMODUL-167 board was replaced with a Phytec PhyCORE MPC554 board. The OSE ck operating system will be executed on a MPC5554 board. Another change in the platform is that two RPX-Lite boards were added. These boards will be used to execute OSE delta. OSE delta is Enea flagship of operating system comparing to OSE ck and OSE Epsilon.

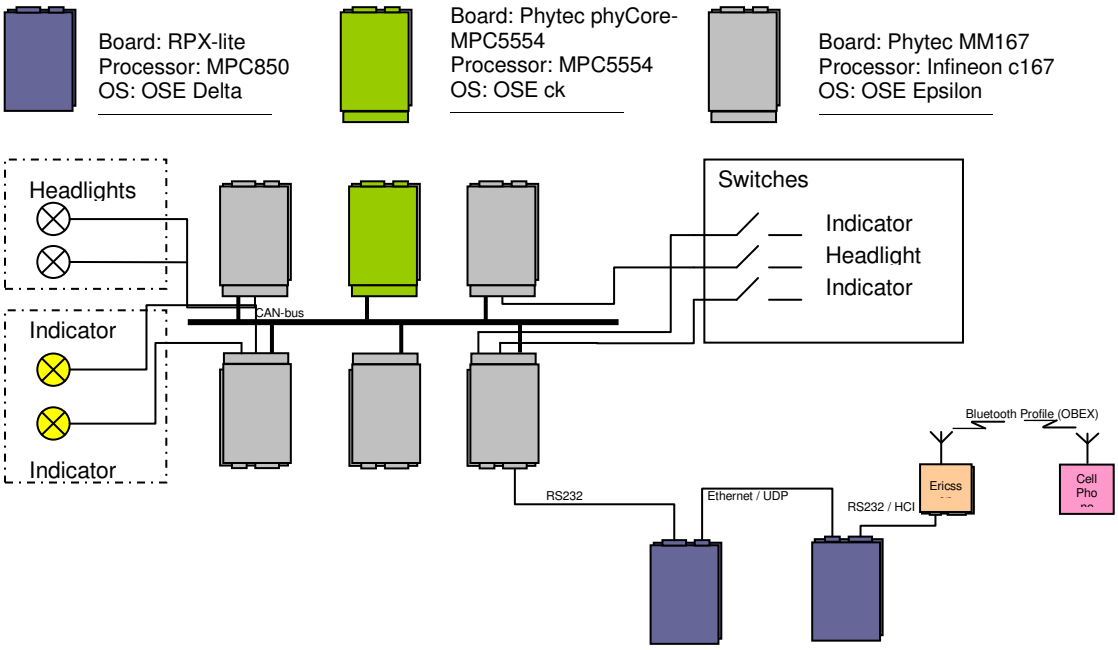

Figure 3.4: The new development platform

The *figure 3.4* showed that there are five mm167 boards, one PhyCORE MPC5554 board and two RPX-Lite boards. The RPX-Lite boards act as a CAN-bluetooth gateway to communicate with a cell phone. This is to visualize a home device integrated with the DySCAS platform.

3.2.3 The CAN protocol

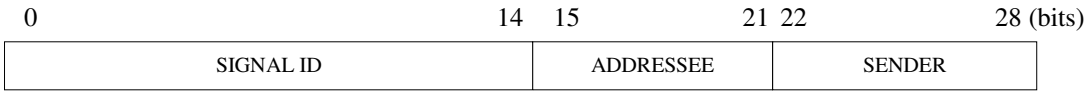

Figure 3.5: Old CAN protocol

There is also an old CAN protocol for the experimental platform that does not supports messages larger than 8 bytes. The protocol was developed very simple and used to transfer OSE signals to nodes in the network. The signals have to be identified in a process to determine the right destination for the signals. The old protocol was replaced with a more advance CAN protocol.

priority	signal ID	dnode	snode	dprocess	sprocess	LE
1 bit	7 bits	5 bits	5 bits	5 bit	5 bits	1 bit

Figure 3.6: New CAN protocol

Table 3.1: Arbitration definations

priority	this determines if the signal is short or long signal
signal id	the signal identification (7bits = 128 different signals)
dnode	destination node (5 bits = 32 different nodes), according to IEEE a CAN bus usually consists 30 nodes.
snode	source node (5 bits = 32 different processes), this means 32 processes which can communicate on CAN bus.
dprocess	destination process
sprocess	source process
LE	little endian or big endian

For better understanding of the CAN protocol the OSE communication knowledge is required. OSE communication is based on direct message passing and OSE processes use signals to send data between them. Memory for the signal is allocated by the operating system from the memory pool and returns to the pool when receiver no longer needs the signal. The signal ownership is strict in OSE, at any given time there is only one owner of signal and its content.

Table 3.2: OSE operating systems

OS	Name Server	Link Handler	ADDRESSEE SIZE
Delta	Y	Y	32
Ck	N	Y	16
Epsilon	N	N	8

The task was to develop a link handler for OSE Epsilon. The communication medium is CAN bus. OSE Epsilon is a fast, compact, real time operating system for embedded systems but there is no proper link handler for Epsilon. As mentioned, a previous version of link handler is available to use, but it does not support packets larger than 8 bytes and was created only for Epsilon. To fulfil the DySCAS requirements a new link handler is required.

Direct message passing communication makes distributed systems easier to design. The OSE link handler supports automatic supervision of communication channels. This allows the communication in the distributed system to be reliable. The link handler automatically notifies the application when problems arise.

Every processes running in Epsilon are numbered from one up to the number of processes, these numbers are also the process identifications (ADDRESSEE). When a signal is sent to an external process, the process identification is not present in the CPU. The signal will then be forward to the link handler.

The addressee in Epsilon is only 8 bits leads to that the assigned identification to a process can be maximum 255. The assigned identification can be up to 255 does not mean that Epsilon can execute 255 internal processes. The maximum numbers of internal processes for Epsilon are 127 processes, which left the rest of the numbers unused.

addressee	Sender	signal id
1 byte	1 byte	2 bytes

Figure 3.7: OSE signal

A CAN bus sends maximum 8 bytes each message and the link handler was developed to be capable to send OSE signals larger than 8 bytes. As described above the priority bit is use to determine if the OSE signal is a short signal or long signal. Short signals are messages smaller or equal to 8 bytes, and long signals are messages bigger than 8 bytes.

When sending the short OSE signal is easy, just convert the OSE signal to CAN message and send it to the destination address. All 8 bytes are used to store data, because that "0" is dominant in CAN the priority bit will be set to 0 for short messages.

Data field
8 bytes

Figure 3.8: Short signals data field

LP	signal size	data field
1 bit	7 bits	7 bytes

Figure 3.9: Long signals data field

Sending a long OSE signal is kind of complex. In the data field for long signals above the first byte of data field is used to store the OSE signal information and the rest are for the

data. The LP (last packet) is set when the last packet is sent, which indicated that this is the last CAN message. The signal size is for the receiving node to know how big the OSE signal is going to be and the largest OSE signal possible by the developed protocol is 127 bytes. This limit is because of the signal size that is only 7 bits.

It is tricky to transform an OSE signal to a CAN message, due to that some trade-off must be made. In this way, only CAN data frame can be used. Retransmission request of corrupted packets must handled by the link handler. Short signals will have higher priority and the long signals will have lower priority. At least, long signals are supported comparing to the old CAN protocol.

The priority of CAN arbitration is defined by the signal identifications. If there is more than one CAN messages with the same signal identification then the arbitration will be between the destination nodes and so on. The developed system is a control system that means most communication signals will be short signals.

3.2.4 Master node implementation

When a master node type start-up it will wait for a master heartbeat. When received the master heartbeat it checks the priority of the received heartbeat. If it has higher priority then the heartbeat it will send the master heartbeat itself. If not it will send the slave heartbeat and become a passive master (slave).

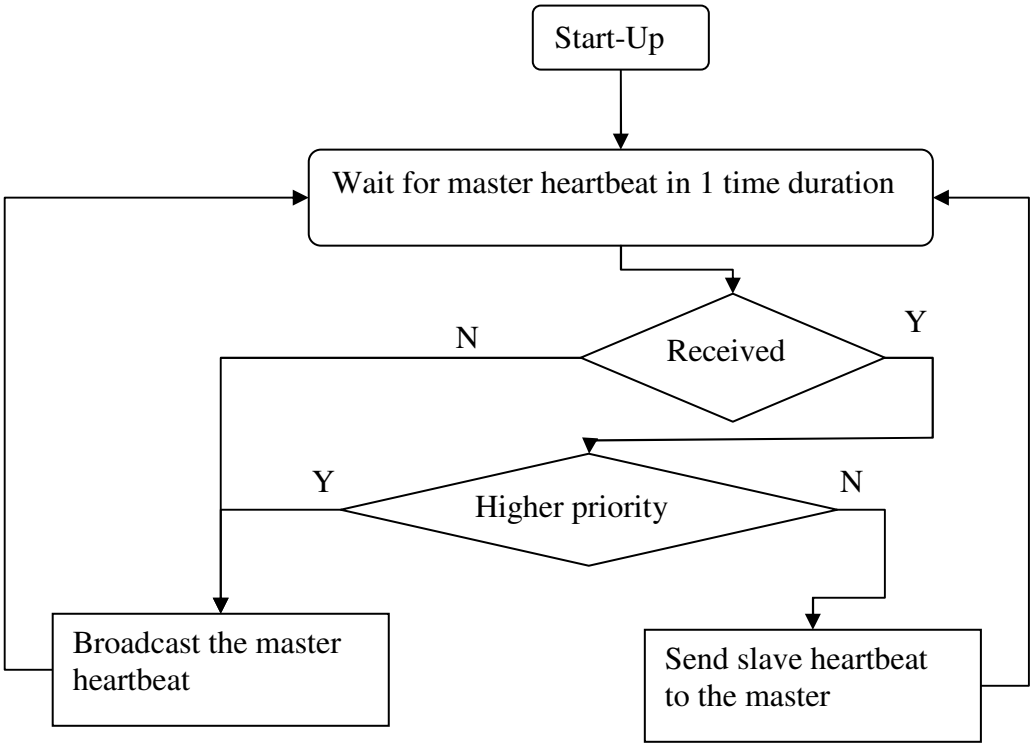

Figure 3.10: Master selection

The master node is the heart of the system, it detects if any node is added/removed from the system. The master node will broadcast its master heartbeat, in that way all the slaves will know where the master is located and all the slaves can replay by sending their slave heartbeats. The master collected and calculated the slave heartbeats to determine the number of devices on the bus.

3.2.5 Link handler implementation

The link handler is designed with three processes and one CAN interrupt process.

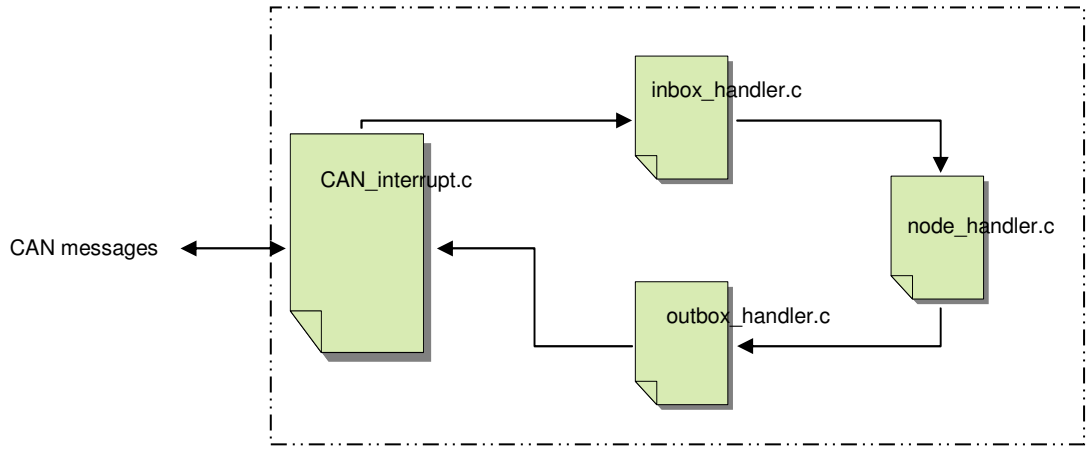

Figure 3.11: Link handler

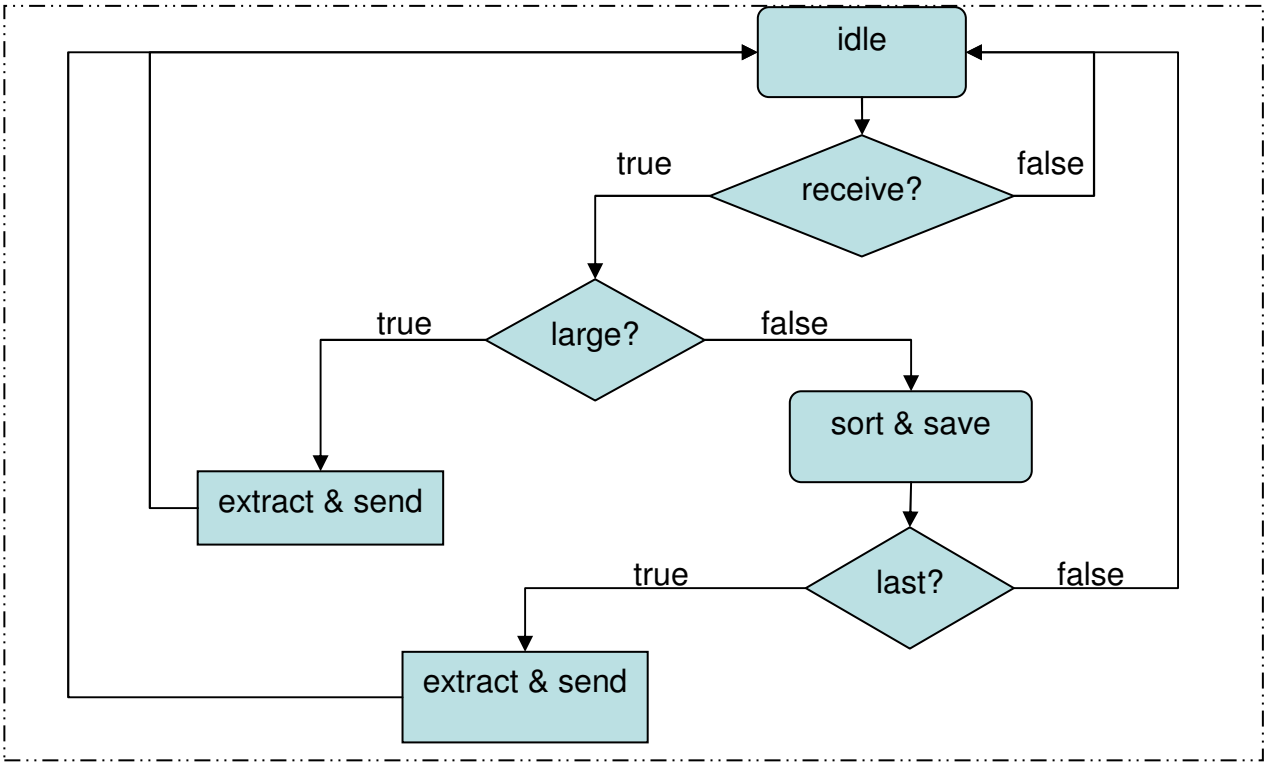

Figure 3.12: Inbox handler

The receiving process is the inbox handler, if a new CAN message is received from the CAN interrupt process, it checks if the message is a short OSE signal or a long OSE signal. If the message is a short OSE signal, the signal will be extract from CAN message and forward to the destination process.

If the CAN message is part of a long OSE signal, a temperate OSE signal will be create and add to the linking list. If all packets are received for the long OSE signal, the right OSE signal will be create and send to the destination process.

The last packet bit is set to let the inbox handler know that this is the last CAN message for the long OSE signal. If the last packet bit is set but the packet size comparing to the number of received packets is not equal, then the long OSE signal failed to receive. The temperate OSE signal will be remove from the list.

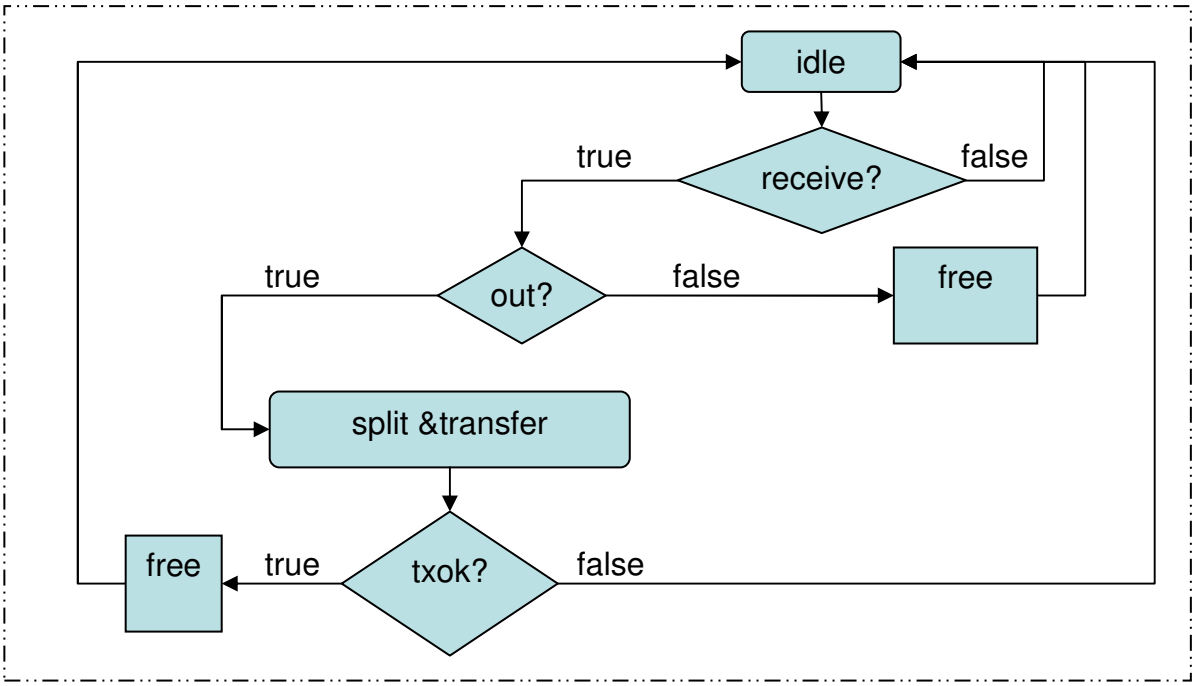

Figure 3.13: Outbox handler

The outbox handler functions are to convert OSE signals to CAN messages and send them on the CAN bus. The outbox handler receives signals which are unknown to the operating system, it checks if the OSE signal was supposed to be forward to an external node or not. If not the link handler will throw away the OSE signal.

If the OSE signal is larger than 8 bytes, the outbox handler will split and send the OSE signal packet by packet on the CAN bus until the whole OSE signal is transmitted. If the packets failed to transmit correctly the outbox handler process will receive a failed OSE signal from the CAN interrupt process.

The node handler is important for the link handler, it send the master heartbeat if the node is the master. It also receives signals from the slaves to identify the standing of the system. If node is a slave it waits for a master heartbeat and reply with a slave heartbeat to indicate that it is alive.

3.2.6 The services registry

The system is supposed to be a service-based because service-based architecture offers more flexibility than the component-based architecture.

To implement a service-based system there is a need of a registry. All slaves know where the master is located due to the master heartbeat that they received. The registry is placed at the master node. This way all nodes in the network know where the registry is located.

The registry is an important module in the middleware. It provided mechanisms for applications to search and register their services. This offers applications to automatic maintain their relationships with each other. The registry detects duplicated of services from the same process. It also supports other signals, examples are system information signal, service registration signal, service locate signal and service subscription signal.

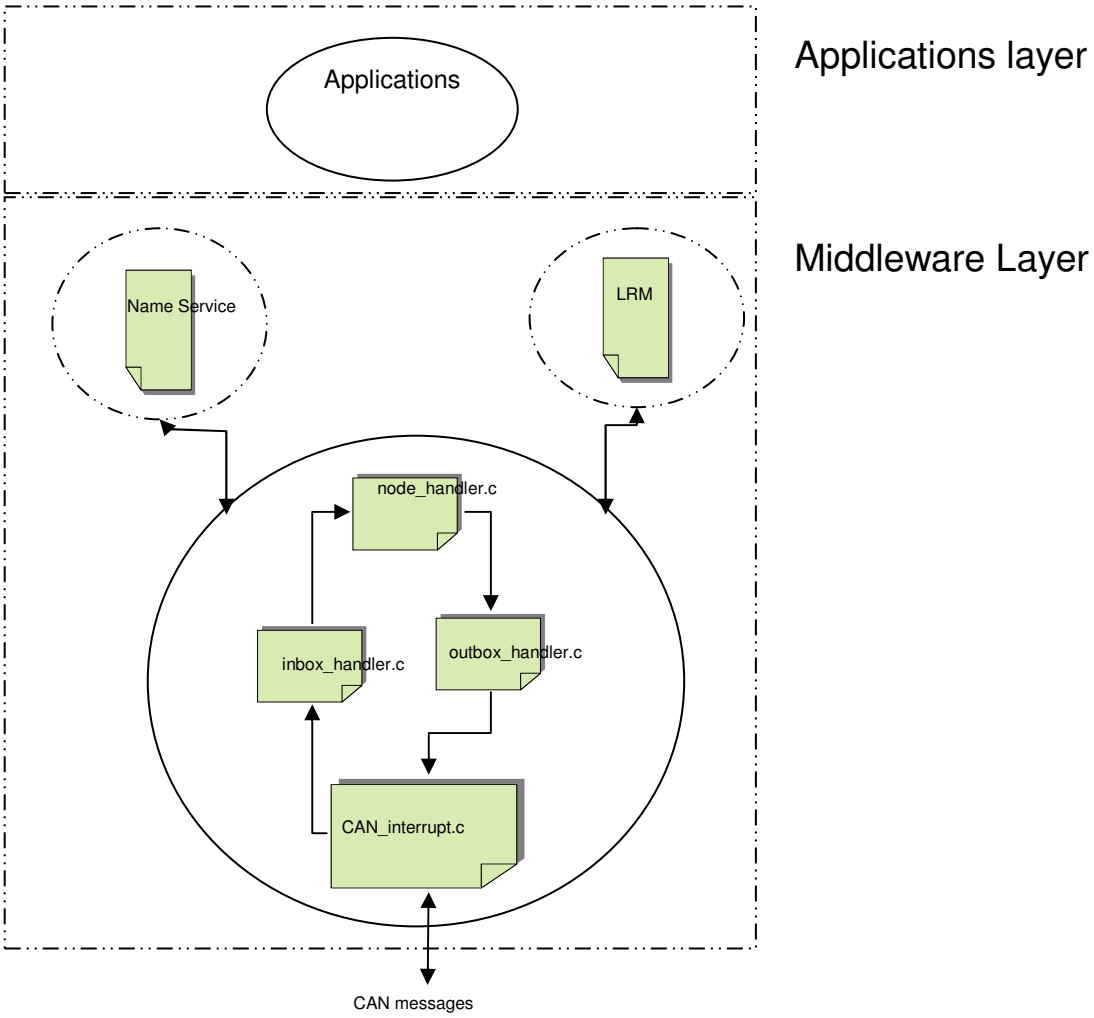

Figure 3.14: The Middleware

The purpose is to develop a system that is self-managing. Therefore, a local resource manager (LRM) for each node is necessary. The LRM collects the services published by its processes (internal processes). In that way if the master node switched place in the network or rebooted, the LRM will do registration for all the services that it has collected in the list.

There are two linking lists in the registry, the list of the services and the list of the subscriptions. The service list contains all the services in the system, a process is allows to has more than one service in the service list, but a process can not has duplicated of services in the service list.

The subscription list handle all the subscriptions, when a subscription signal is received, the registry search the service list for the service if the service does not exist, the subscription signal will be save in the subscription list.

The registry also received signals from the node handler. (Node down signal) If any node is down then the registry will remove the services belonged to the failed node. The Not master signal will be received if the node is no longer the master. The node then will remove all the services in the service list and remove all the subscriptions in the subscription list.

With this design the link handler can transfer signals up to 127 bytes. Signals with higher priority get the transmission time as they shall. The platform has been tested with 6 nodes executed simultaneously without any delay. The link handler also detects failed nodes within 1 second. More performance tests have not been done this is due to the shortage of time.

3.3 The demonstration applications

To be able to test the designed system for vehicle industry there is a need of demonstration applications. There are three different ways to turn on the indicators and the headlights in the experimental platform using the developed system.

1) An application using a blinking service: The application reads the status of the switches and sends the information to the blinking service.

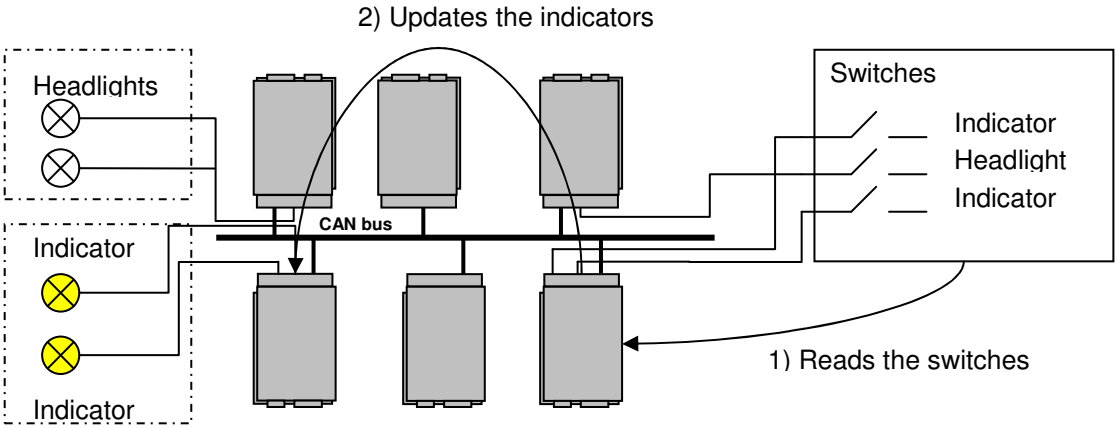

Figure 3.15: Direct service

2) An application with two services: The application send information request to the reading service, the reading service sends the requested information to the application. The application then sends the status of the switches to the blinking service.

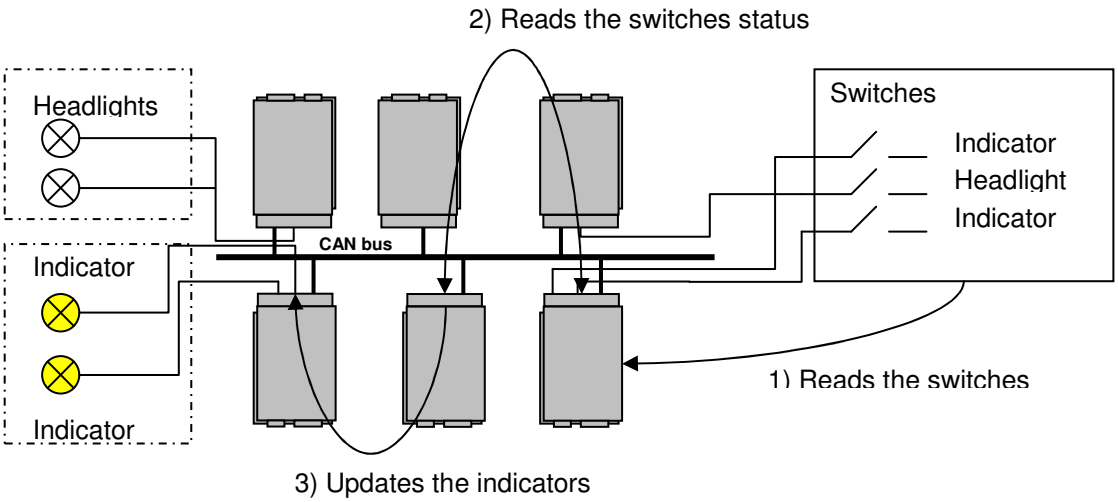

Figure 3.16: Using two services

3) An application, information collector and a service: The application reads the status of the switches and forwards the information to the information collector, the information collector then updates the blinking service.

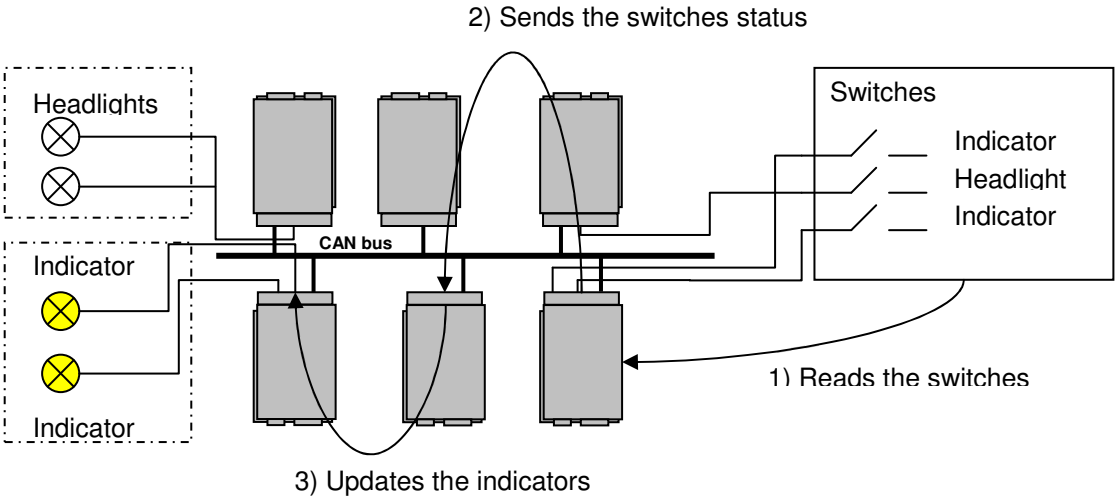

Figure 3.17: Cascading services

The last solution was selected because it highlights the abilities of a distributed reconfigurable embedded system. For the indicators to function correctly there must be at least two slave nodes and one master node in the system.

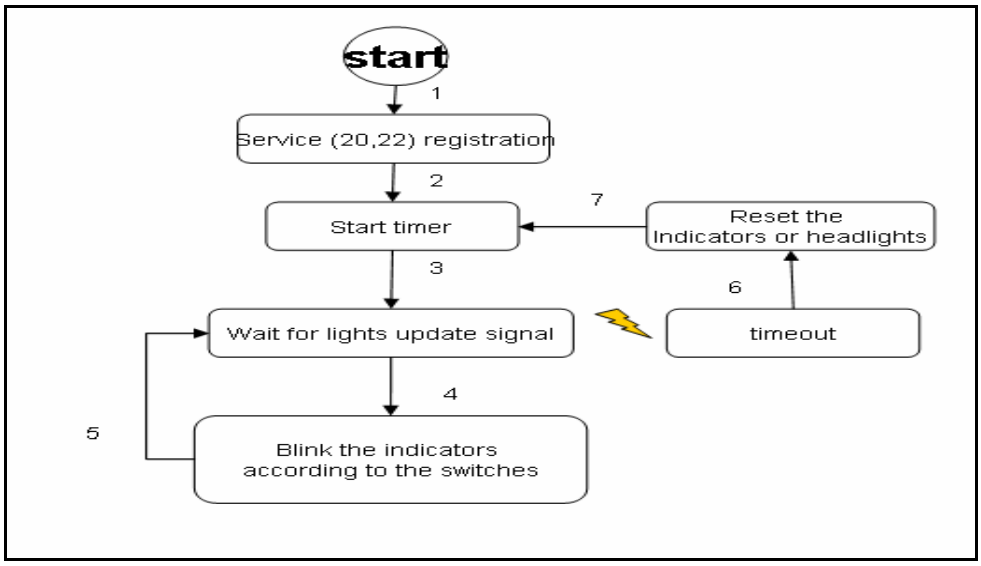

Figure 3.18: The blink service

In the *figure 3.18* the design of the blink service is showed in a state-machine. The service started with the service registration. When the service is registered the service is added to the service list at LRM and the registry at the master node. Therefore, when the master went down and up in another place at the bus. The service list at the LRM wills automatic register the services that it has in the service list.

After the registration the blink service will start a timer and wait for light update signal from any applications/server. If no signal is received within timeout the service will reset the indicators or headlights to the initial state.

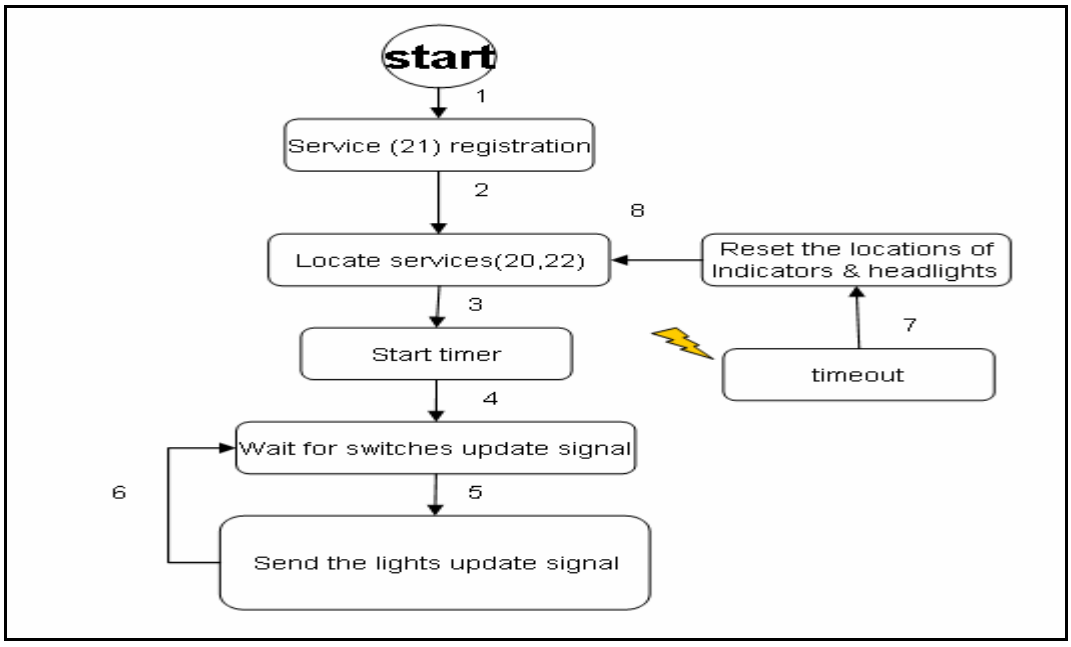

Figure 3.19: The switch service using the light service

In the same way the switch service was developed. After the service registration the switch service will locate the light services on the system. If it received the locations it will update the destinations to the light services. It will wait for the switch update signal from any applications/services. If it received the switch update signals it will send the light update signals to the light services that it found on the network. If no signal was received within timeout when locate the light services it will set the destinations to not available.

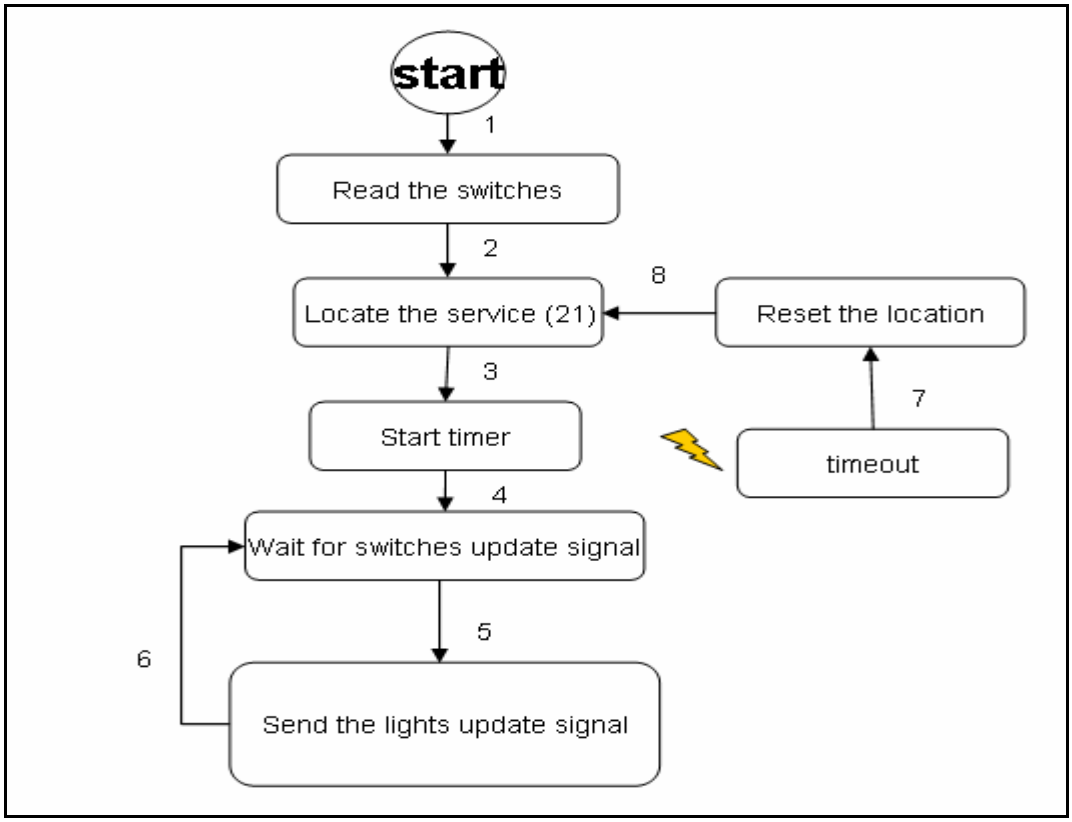

Figure 3.20: The switches application

The switches application reads the switches and locates the switch services on the network. If any switch service was found on the network it will forward the information to that service. If timeout it will set the destination to not available.

3.4 Service registration and using a service

Now that the whole system is described, a closer look at the service registration and using the service.

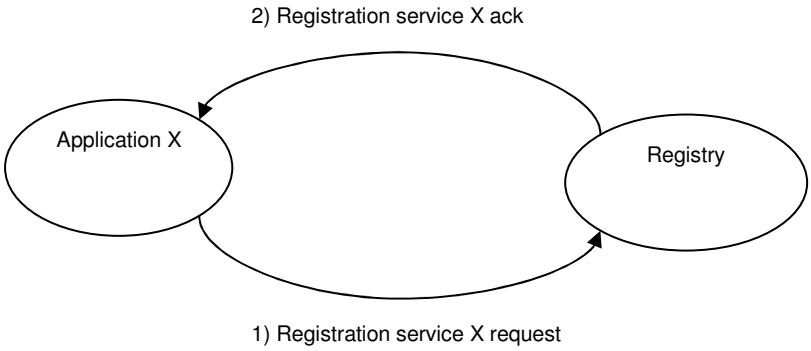

Figure 3.21: Service registration

When an application wants to publish a service, it sends a registration signal with a service ID to the registry. The registry will reply by sending an ACK to the requested process. The signal from the application will be received by the LRM and then forward to the LNH. The LNH will send the signal to the LNH at master node. When the LNH at the master node received the signal it will forward it to the registry. The registry will reply by to the LNH at the master node. The LNH at the master node then forward the signal to the LNH at the application node. The LNH at the application node will then forward the signal to the application.

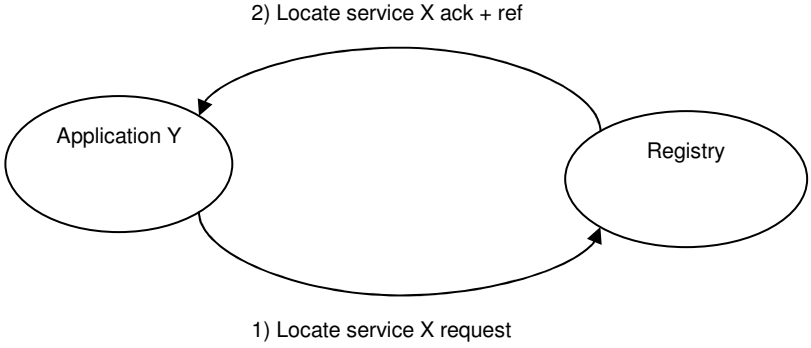

Figure 3.22: Locating a service

When an application wants to subscript a service, it sends a locate signal with a service ID to the registry. The registry sends back an ACK with a reference of the service to the requested process if the service exists in the system. The signal path acts as the same way as a service registration.

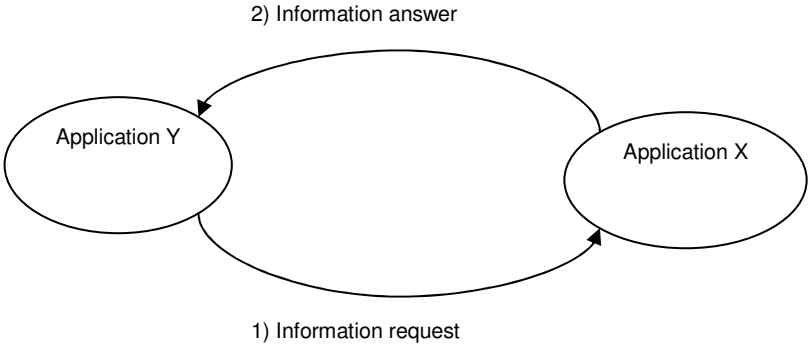

Figure 3.23: Using a service

Using a service is very simple. The application only needs to know where the service is located in the system. When is received the reference of the service through the registry, the application can immediately starts communicate with the service. The communication path is from the application y to the LNH.

3.5 The CAN monitor to observe the status of the bus

To know the status and the activities of the system there is a need of a CAN bus monitor. A previous application which used a USB-CAN gateway to monitor the CAN bus was available. However, this CAN monitor was developed for the previous architecture. Due to that, the CAN monitor was unable to identify the signals used in the developed middleware.

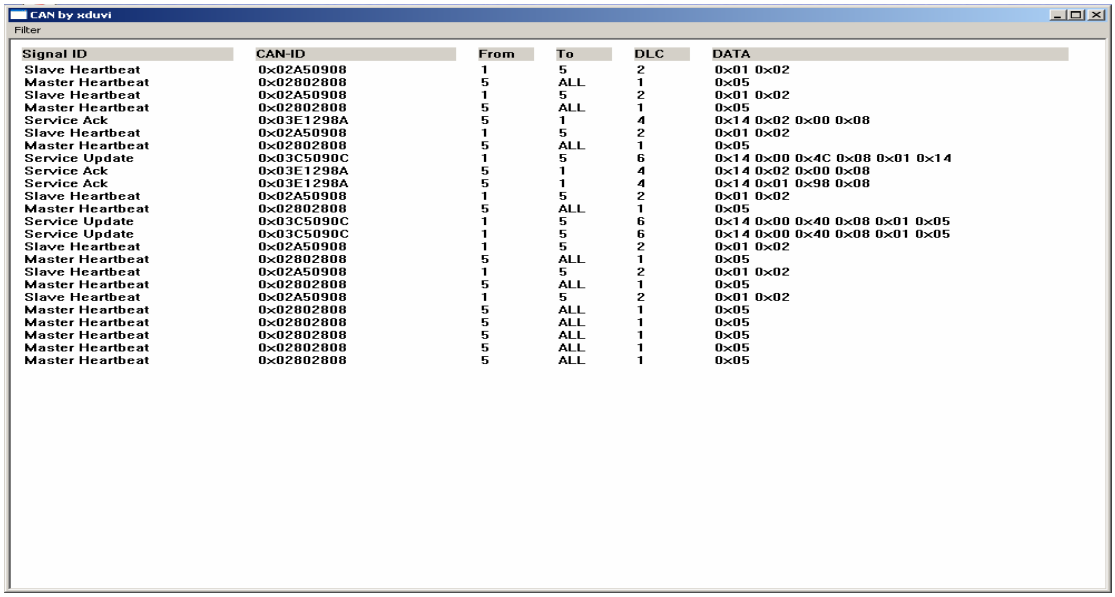

Figure 3.24: CAN monitor

To monitor the communications and activities on the CAN bus a new CAN monitor application was developed. The CAN monitor has a very simple interface. However, it still fulfilled the functions that it was created for. The monitor highlights the signal ID, CAN-ID, the destination node, the source node, how many data was sent, and what kind of data was sent. The drawback with this CAN monitor is that long OSE signals are difficult to read.
The CAN monitor is using the dll-files from SYSTEC USB-CANmodul. There are three steps to initiate the USB-CANmodul:

1. Start the USB-CANmodul hardware
2. Start CAN hardware
3. Monitor the CAN bus

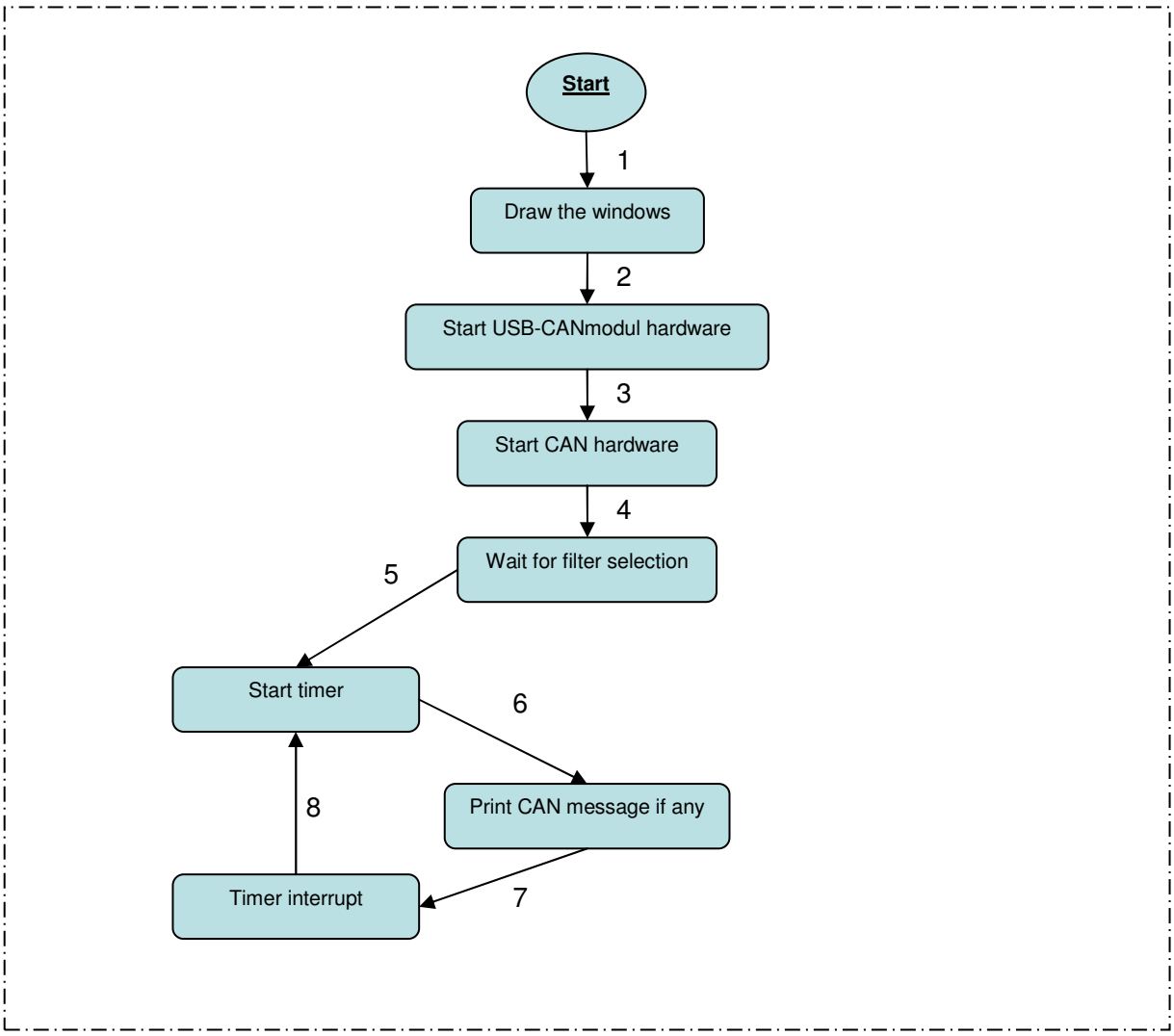

Figure 3.25: CAN monitor state chart

4 Results

The results of this master thesis are showed in the tables below. Most of the requirements are fulfilled and this showed that the proposal approach is the right path to develop a dynamically self-configuring automotive system.

4.1 The result of the requirements

Table 4.1: DySCAS results

DySCAS requirement	**Fulfilled**	**Description**
Scalability	Yes	Nodes are added or removed during runtime
Flexibility	Yes	The system is service-based, applications can easily publish/subscript services
Portability	Partly	The middleware is OSE operating systems depending
Openness	Yes	The middleware interface are developed to support third-part applications (the system complexity is completely hidden from the applications)
Robust	Partly	The middleware detects failed node, but it does not detects failed process
Distributed	Yes	The system is based on CAN bus, but bluetooth has been added into the system through a gateway

Table 4.2: Master Thesis results

Master Thesis requirement	**Fulfilled**	**Description**
CAN protocol	Yes	The DySCAS CAN protocol is developed to supports 32 nodes, signals larger than 8 bytes, Little Endian and Big Endian, Priority mechanisms
Link Handler	Partly	Detects failed node, transparent between processes, send signals larger than 8 bytes (Little Endian and Big Endian is not implemented yet)
Middleware	Partly	Portable (OSE operating systems dependence), openness (the system complexity is completely hidden from the applications) and scalable (middleware is divided in modules)
DySCAS applications	Yes	The demonstration applications are using the middleware
Status of the system	Partly	A CAN monitor application was developed using a USBCAN gateway

Tables 4.1 and 4.2 highlighted that some requirements are not completely fulfilled. This is due to the limitation of time. However, those requirements that are important to the system have fulfilled.

The scalability requirement: Nodes are added/removed during runtime, the master node detects the standing of the system. It collects the slave-heartbeats and supervises the nodes in the network.

The flexibility requirement: The system is service-based, applications can easily publish/subscript services. The master node detects failed node, it forwards the information to the registry to remove the services published by the failed node.

The robust requirement: The system detects failed node, but it does not detect failed process. The master node is the heart of the system and there must be at least one master in the system to coordinate the services. However, if the master switched place or rebooted, there is a Local Resource Manager (LRM) in each node. The LRM in each node collects the services published by its node. In that way if the master node switched place in the network or rebooted, the LRM will do registration for all the services that it has collected in the list.

The openness requirement: The middleware interface allows applications to publish/subscript services without knowledge of DySCAS system. This way it opens up for third-part developers to develop their applications for the DySCAS platform. Moreover, examples of implementation are the indicators and headlights applications.

The distributed requirement: The system is based on CAN bus which interconnect nodes. However, a CAN-bluetooth gateway is added into the system to communicate with bluetooth devices.

The results visualized that a dynamically self-configuring automotive system was developed and most of the requirements supplied by the vehicle manufacturers are fulfilled. The combination of self-managing, service-based and middleware is an approach that can fulfil future vehicle electronic systems requirements.

5 Conclusions and Future Work

5.1 Conclusions

The implementation of a dynamically self-configuring system is very basic. Because of that many functions are still missing, but functions that were necessary have been developed, examples are the service registry, the link handler and the local resource manager.

As in the results pointed out Little Endian and Big Endian is still not implemented, this is due to that we did not have a board with Big Endian processor. Right now, the link handler detects failed, removed or added node but it does not detects failed process.

The middleware is now OSE operating systems dependence, therefore future work may be to extend the portability of the middleware. Technologies have been studied to solve flexible and scalable distributed systems and not much has been done on the security aspect. Future work will be to add security into the middleware.

The approach with policy-based, service-based and middleware is the right path to develop a dynamically self-configuring automotive system. The immature of dynamically self-configuring automotive systems in the market leads to many limitations in the system. Examples of limitations are networks (self-adaptive bus which supports higher data-rates), middleware (develop for real-time and embedded) and operating systems (develop for distributed communication between processes in different nodes).

In the information era, distributed systems are the future. Dynamically self-configuring automotive systems are here to stay, but more research is required.

5.2 Future Work

Future works for the DySCAS system is to add the security, make the middleware platforms and operating systems independence. Another aspect is to investigate the Quality of Services (QoS) of the system. Due to the limitation of time the system performance has not been analysed completely. In a vehicle electronic system there are many devices requiring responses in a finite time.

There are limitations in the link handler. The nodes are still using the old addressing method (outbox_handler). The limitations of addressable nodes are still 7 only. Adapt the addressing method to the new signals (destination node and destination process are baked in the struct) for full potential of the CAN protocol.

Old method:

extern	dnode	dprocess
1 bit	3 bits	4 bits

dnode = addressee & 0x70 >> 4;
Limited CAN potential

Proposal method:

```
#define DYSCAS_EXAMPLE_SIGNAL (0)
struct dyscas_example_signal
{
   SIGSELECT sig_no;
   U8 dnode;                  // destination node
   U8 snode;                  // source node
   U8 dprocess;               // destination process
   U8 sprocess;               // source process.
   U8 data1;                  // data1
   U8 data2;                  // data2
   U8 data3;                  // data3
   U8 data4;                  // data4
   U8 data5;                  // data5
};
```

dnode = signal->dnode;

Full CAN potential

When a signal is supposed to be send to another node, the destination node, destination process, source node and source process are send in both CAN data field and arbitration. Remove this information from the data field to allow a signal to transfer faster.

Old method:

arbitration	data field
signal addresses	whole signal (addresses + data)

More CAN messages

Proposal method:

Arbitration	data field
signal addresses	signal data

Less CAN messages

Right now the outbox_handler sends the whole long signal (signal larger than 8 bytes) before any short signal (signal smaller than 9 bytes) can be send. Redesign the outbox_handler to send the priority signal between sending a long signal.

Old method:

```
If short:
        Send short signal on CAN
Else:
        Send long signal until finish
```

Proposal method:

```
If short:
        Send on CAN
Else:
        Send part of long signal on CAN
        If receive short signal:
            Send short signal on CAN
        Else:
            Back to send part of long signal on CAN
```

It is possible to upgrade devices (flash) via CAN and due to the time limitation not much has been done on that aspect. There are three pieces of code when flashing devices via CAN:

The CAN initialization and transmission code for transmitter
The CAN initialization and programming code for receiver
The application code

This is an example only, codes and further information can be obtained at Infineon's homepage. Future use-cases impose high demand on the dynamic reconfiguration. Therefore, to redesign the system from fixed policies to dynamic policies are suggested.

6 References

[1] DySCAS, *Proposal Summary Page*, Enea AB, February 2006
[2] DySCAS, *Generic functionalities 2*, Enea AB, 2006
[3] Anthony R, Butler A, *Dynamically Self-Configuring Automotive Systems*, Report, Enea AB, 2006
[4] Kephart J, Chess D, *The Vision of Autonomic Computing*. Computer, IEEE, Volume 36, Issue 1 January 2003, pp. 41-50
[5] White S, Hanson J, Whalley I, Chess D, Kephart J, *An Architectural Approach to Autonomic Computing*, Computer, IEEE, 2004, pp. 2 – 9
[6] Geihs K, *Middleware challenges ahead*. Computer, IEEE, Volume 34, Issue 6 June 2001, pp. 24-31
[7] Fung KH, Low G, Ray PK, *Embracing Dynamic Evolution in Distributed Systems,* Computer, IEEE, Volume 21, 2004, pp. 49 – 55
[8] Nolte T, Hansson H, Bello L.L. *Automotive Communications – Past, Current and Future*, Computer, IEEEE, Volume 1, 19-22 September 2005, pp. 985-992
[9] Parnell K, *Put The Right Bus in Your Car,* http://www.xilinx.com
[10] Cena G, Valenzano A., *On the properties of the flexible time division multiple access technique*, Computer, IEEE, Volume 2, Issue 2, May 2006 pp. 86 – 94
[11] *C16xCx programming an external flash memory via the CAN bus,* www.infineon.com
[12] Navet N, Song Y, Simonot-Lion F, and Wilwert C. *Trends in Automotive Communications Systems*, Computer, IEEE, Volume 93, Issue 6, June 2005, pp. 1204 – 1223
[13] Finocchiaro R, Lankes S, Jabs A, *Design of a real-time CORBA event service customised for the CAN bus*, Computer, IEEE, April 2004 pp. 121
[14] Kaiser j, Mock M, *Implementing the Real-Time publisher/subscriber model on the controller area network (CAN).* Computer, IEEE, May 1999, pp. 172-181
[15] Jabs A, Lankes S, Bemmerl T, *Design and performance of a CAN-based connection-oriented protocol for Real-Time CORBA,* Journal of Systems and Software, ScienceDirect, Volume 77, Issue 1, July 2005, pp. 37-45
[16] *OSE Reference Manual*, Enea AB
[17] Österman J, *Scalability and QoS for embedded distributed control system,* Thesis Report, KTH Stockholm, 2001
[18] Gille M, *Service networks in embedded real-time systems*, Thesis Report, KTH Stockholm, March 200