A Device Message Bus for Sensor Networks

DIMITRI MAZMANOV

Master of Science Thesis
Stockholm, Sweden 2010

TRITA-ICT-EX-2010:316
A Device Message Bus for Sensor Networks

Dimitri Mazmanov

Supervisor: Richard Gold, PhD, Ericsson AB
Examiner: Vladimir Vlassov, Professor, KTH

Master’s thesis
School of Information and Communication Technology (ICT)
KTH Royal Institute of Technology
Stockholm, November 2010
Abstract

Sensor Networks are typically comprised of heterogeneous technologies as they cover a number of different fields, for example, home automation, intelligent transport systems, environmental monitoring and consumer electronics. In order for a systems provider to efficiently communicate with such devices, it is necessary to create a layer of abstraction, called a Device Message Bus (DMB), which allows the heterogeneity of the devices to be hidden from the provider. The goal of this work is to create a system that provides a simple, high-level interface to many different types of sensors deployed by the systems provider and allows, in its most basic form, the sending of messages to the sensors, irrespective of which networking technology the sensors use to communicate. Additionally, this basic functionality needs to be augmented with features such as: support for at least REST, WS and UDP communication protocols, support for future growth of the system, which implies plug-and-play capability, discovery of new sensors, reliability of message transmission and scheduling of communication events. The result of the work is a complete system, that addresses the stated goal. The evaluation phase showed the correctness of the system, its fault tolerance and performance.
# Contents

1 Introduction .......................................................... 1  
   1.1 Problem Description .......................................... 1  
   1.2 Goals ............................................................ 1  
   1.3 Solution Overview ............................................. 2  
   1.4 Report Overview ............................................... 3  

2 Technical Background .................................................. 5  
   2.1 Wireless Sensor Networks .................................... 5  
      2.1.1 Sensor Network Technology ........................ 6  
   2.2 A Message Bus Enterprise Pattern ......................... 6  
   2.3 Service Oriented Architecture ............................ 7  

3 Related Work .................................................................. 11  
   3.1 Telenor Shepherd / COOS Framework ..................... 11  
   3.2 Enterprise Service Buses .................................... 12  
      3.2.1 OpenESB .................................................. 12  
      3.2.2 Apache ServiceMix ................................. 12  
   3.3 Warp ............................................................... 13  

4 The Device Message Bus Design ....................................... 15  
   4.1 Overview .......................................................... 15  
   4.2 System Requirements ........................................ 15  
   4.3 System Design ................................................... 17  
      4.3.1 Component Based Development .................. 17  
      4.3.2 DMB Connection Interface ....................... 18  
      4.3.3 Message Router ....................................... 20  
      4.3.4 Identity Management ................................ 21  
      4.3.5 Resource Database .................................. 23  
      4.3.6 Logger ..................................................... 25  
      4.3.7 Scheduler ................................................ 25  
      4.3.8 Management System ................................. 26  

Chapter 1

Introduction

1.1 Problem Description

Nowadays the integration of the physical world into the digital world is becoming a necessity for future Internet technologies. An increasing number of services are relying on physical world information and interaction capabilities. While many of the current systems require a human interaction we can still observe application specific deployments of wireless sensor networks automating the interaction processes between physical and digital worlds.

Sensor Networks are typically comprised of heterogeneous technologies as they cover a wide range of different fields, for example, home automation, intelligent transport systems, environmental monitoring and consumer electronics. For example, a home automation system integrates all electrical devices in a house with each other. Typically these devices are produced by different vendors, thus there is no guarantee that they will have a common communication and/or integration protocols. An energy sensor can use wired Universal Powerline Bus (UPB) communication, while a motion detector sensor can employ wireless General Packet Radio Service (GPRS) communication. Even if an existing home automation system is configured and ready to use, installing new devices might lead to some major changes to the whole system.

In order for a systems provider to efficiently communicate with such devices, it is necessary to create a layer of abstraction which allows the heterogeneity of the devices to be hidden from the provider.

1.2 Goals

The goal of this thesis work is to create a system that provides a simple, high-level interface to all different types of sensors deployed by a systems provider and
allow, in its most basic form, the bidirectional message transmission between the clients and the sensors, irrespective of which networking technology the sensors use to communicate. In order for the system to increase its productivity, this basic functionality needs to be augmented with features such as: identity management, automated registration of new sensors, reliability of message transmission, security of communication channels, various message transmission strategies, management system, etc.

1.3 Solution Overview

To address the design goals we designed and implemented a message transmission system called a Device Message Bus (DMB). Based on the analysis of the system requirements (see section 4.2) we created a software architecture, which provides a component based environment for building the modules (components) for the system. Typically this is achieved through modularity of programming and encapsulation. Further, we developed a model for representing the components inside the DMB (Figure 1.1). A component can support a functionality that is common for other components.

![Component Based Environment](image)

Figure 1.1: Component Based Environment

In order to support a level of dynamism for communication components we added a plug n play capability, which allows the DMB components to be substitutable at system runtime (blocks P1, ..., PN on the 1.1). Another important feature of the DMB is to provide an API for creating system components that are responsible for communication aspects of the system - the communication handlers. Our implementation distinguishes two types of communication handlers: handlers that are responsible for providing the communication means for sensors, and handlers that are responsible for providing the communication means for applications. The separation of these handlers helps to distinguish between the entry points into the system. A transmitted message has a specific format described in section 4.3.9. A message is transmitted from an application communication handler through the DMB core to a sensor communication handler without any modification within the system. That is, the DMB and all handlers support common format of a message. Apart from single message sending, the DMB also provides support for multicast
message transmission, that is the sending of a message to the multiple destinations. Other components supported by the DMB are: Logger, Scheduler, Management System and Database Manager.

Finally, in order to verify the system a sequence of real world tests were performed, which evaluated system correctness, performance and fault tolerance.

1.4 Report Overview

This document is organized as follows. Chapter 2 describes the technologies that are addressed by this work. Chapter 3 provides the review of existing systems that solve similar problem. Chapter 4 describes the overall methodology followed to design the DMB architecture, and introduced the design goals (stemming from requirements analysis) and resulting design principles. Chapter 5 provides the details about the implementation of designed system. The analysis and evaluation of the system is described in Chapter 6. Finally, Chapter 7 will conclude the overall work and provide some thoughts about future improvements/extensions for the DMB.
CHAPTER 1. INTRODUCTION
Chapter 2

Technical Background

This chapter gives the description of the technologies that are addressed by the Device Message Bus, notably Wireless Sensor Networks, Service Oriented Architectures and loosely coupled enterprise systems.

2.1 Wireless Sensor Networks

A Wireless Sensor Network (WSN) is an infrastructure comprised of low-cost, low-power, measuring, computing, and communicating wireless devices that gives a user the ability to monitor, and react on events in a specified environment. There are four basic components in a WSN: an assembly of distributed or localized wireless sensors, an interconnecting wireless-based network, a central point of information clustering, and a set of computing resources at the central point (or beyond) to handle data correlation, event trending, status querying, and data mining. [11]

Nowadays, interest has focusing on networked biological and chemical sensors for national security applications; furthermore, evolving interest extends to direct consumer applications. Existing and potential applications of sensor networks include, among others, military sensing, physical security, air traffic control, traffic surveillance, industrial and manufacturing automation, inventory management, distributed robotics, weather sensing, environment monitoring, national border monitoring, etc. Near-term commercial applications include, but are not limited to, industrial and building wireless sensor networks, appliance control [lighting, and heating, ventilation, and air conditioning (HVAC)], automotive sensors and actuators, home automation and networking, automatic meter reading/load management, consumer electronics/entertainment, and asset management.
2.1.1 Sensor Network Technology

The technology for sensing and control includes electric and magnetic field sensors, radio-wave frequency sensors, infrared sensors, location/navigation sensors, seismic and pressure-wave sensors, environmental sensors (e.g., wind, humidity, heat)[11]. Today’s sensors can be described as “smart” devices equipped with multiple onboard sensing elements. A possible example of such devices can be a modern smartphone with a set of sensors measuring various parameters, such as temperature, magnetic field, location, and providing this data to the user.

WSN Communication Protocols

Characteristics of WSNs such as limited power supply, massive deployment, limited processing power can lead to design challenges at the communication layer. Message transmission between sensors is especially expensive. Thus an efficient communication layer is a crucial part of every WSN. Although traditional transport protocols (TCP, UDP) allow flexible flow and are still used in deployed WSNs, they are far from being a good choice for WSNs. There are many protocols for WSNs designed with an eye on congestion control, reliability guarantee, and energy conservation: CoAP (Constrained Application Protocol), CODA (Congestion Detection and Avoidance), 6LoWPAN (IPv6 over Low power Wireless Personal Area Networks), RMST (Reliable Multisegment Transport), PSFQ (Pump Slowly, Fetch Quickly), etc.

2.2 A Message Bus Enterprise Pattern

A Message Bus is a combination of a common data model, a common command set, and a messaging infrastructure to allow different systems to communicate through a shared set of interfaces [9]. In order to understand the purpose of a Message Bus consider an integration solution consisting of a set of applications provided by different vendors. These applications use their own messaging techniques in order to send or receive messages. In order for the applications to communicate with each other each of them should have individual connections to the others. While this works well for a few applications, it becomes a burden as the number of applications increases. Adding new application to an integration solution leads to the following complications: communication between applications creates dependencies between the applications; these dependencies translate into coupling between the applications. In a point-to-point connectivity, the coupling has a quadratic growth with the number of applications, i.e. three interconnected applications need three connections, but 10 applications need 45 (Figure 2.1). Different applications have
2.3 Service Oriented Architecture

According to World Wide Web Consortium (W3C) a Service Oriented Architecture (SOA) is “A set of components which can be invoked, and whose interface
Component Based Development and Integration (CBDI) rejects this definition for two reasons: First the components will often not be a set. Second the W3C definition of architecture only considers the implemented and deployed components, rather than the idea of building the architecture. CBDI defines SOA as: “The policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be invoked, published and discovered, and are abstracted away from the implementation using a single, standards-based form of interface”. Not only it is an architecture of services seen from a technology perspective, but the policies, practices, and frameworks by which we ensure the right services are provided and consumed. Any SOA contains three roles: a service requestor, a service provider, a service registry (Figure 2.3):

1. A service provider is responsible for creating a service description and making it available for other entities over the network by publishing that service description to one or more service registries.

2. A service requestor is responsible for finding a published service description, and responsible for using service descriptions to invoke services hosted by service providers.

3. A service registry is responsible for keeping service descriptions published by service providers and allowing service requestors to the collection of service descriptions contained within the service registry.

A Service Oriented Architecture can be implemented using a wide range of technologies, such as: SOAP, RPC, REST, Web Services, WCF, etc. Implementations
2.3. SERVICE ORIENTED ARCHITECTURE

Figure 2.3: Service Oriented Architectures

can use one or more of these protocols. The key is independent services with defined interfaces that can be called to perform their tasks in a standard way, without a service having foreknowledge of the calling application, and without the application having or needing knowledge of how the service actually performs its tasks.

We will consider two widely used technologies that implement SOA.

Web Services

W3C defines Web Services as “a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using Simple Object Access Protocol (SOAP) messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards” [20].

A Web service can be thought of as a layered set of technologies. Figure 2.4 shows the Web services interoperability stack and associated Web services technologies.

A detailed explanation of each layer of the stack is out of the scope of this work and can be found in [18].

REST

Representational State Transfer (REST) is set of architectural principles by which Web services are designed. RESTful architectures consist of Clients and Servers. Clients initiate requests to servers; servers process requests and respond back to the clients. Requests and responses are built around the transfer of “representations” of “resources”. A resource can be essentially any meaningful concept that may be
addressed. A representation of a resource is typically a document that captures the current or intended state of a resource [17]. The foremost REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods (Table 2.1).

<table>
<thead>
<tr>
<th>Compositional</th>
<th>BPEL4WS, WS-Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality of Experience</td>
<td>WS-Security, WS-ReliableMessaging, WS-Transactions, WS-ResourceLifetime</td>
</tr>
<tr>
<td>Description</td>
<td>WSDL, WS-Policy, UDDI, WS-ResourceProperties</td>
</tr>
<tr>
<td>Messaging</td>
<td>XML, SOAP, WS-Addressing</td>
</tr>
<tr>
<td>Transports</td>
<td>HTTP, HTTPS, SMTP, Etc.</td>
</tr>
</tbody>
</table>

Figure 2.4: Web Services Interoperability Stack [18]

For example, to create a resource on the server, POST method must be used; to change the state of a resource PUT must be used, etc.

Another important REST design principle is that all REST interactions are stateless. Stateless services are easy to design, implement and distribute across servers. A stateless collaboration between client application and service improves performance by saving bandwidth and minimizing server-side application state.

<table>
<thead>
<tr>
<th>REST</th>
<th>HTTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>POST</td>
</tr>
<tr>
<td>Read</td>
<td>GET</td>
</tr>
<tr>
<td>Update</td>
<td>PUT</td>
</tr>
<tr>
<td>Delete</td>
<td>DELETE</td>
</tr>
</tbody>
</table>

Table 2.1: REST One-To-One Mapping
Chapter 3

Related Work

In this chapter other approaches that try to solve similar tasks are described. Specifically, Telenor Shepherd / COOS Framework, Open ESB, Warp and Apache ServiceMix.

3.1 Telenor Shepherd / COOS Framework

The Connected Objects Operating System (COOS) [12] is a framework that provides communication between software and hardware objects, enabling data exchange. Any integrated component inside the COOS is called a “Connected Object (CO)” (see Figure 3.1). The notion of a CO describes a “pluggable” application that is deployed into a container in order to exchange data with other modules. The instance of a container itself is called COOS. A COOS instance can contain several COs, which jointly define the characteristics of that specific COOS instance.

An SDK (“Plug-in Framework”) is provided for connecting objects to the platform. The SDK consists of a generic protocol of communication that the object can use to send and receive data and events from around the platform, accompanied by implementation support for communication patterns. An object-numbering scheme is used, to provide a uniform way of addressing connected objects.

The main purpose of COOS is to easily enable communication between objects. Most objects do not have the ability to read or process data coming from another object. COOS solves this problem by providing a messaging framework that reads, transforms, and sends data as messages between those objects. Although the COOS Framework enables data exchange between application and devices, it was unclear how this data exchange is implemented, to what extent the existing framework is extensible, scalable and customizable. Due to the lack of documentation, these questions remained unanswered.
3.2 Enterprise Service Buses

3.2.1 OpenESB

OpenESB[15] is a system that provides for a flexible and extensible platform on which to build a Service Oriented Architecture (SOA) and Application Integration solutions. OpenESB supports open standards for interoperability with other products and solves common problems and patterns, such as fault handling, integration patterns, etc. Open ESB consists of a runtime, a design time, and a management console. The runtime consists of a light weight Java Business Integration (JBI) core, and several components (HTTP, JMS, WS, etc.). Several dozens of different components are available, each with its own design time support. One of the central concepts of OpenESB is a message based integration. Messages are sent to services. What messages a service can receive and will respond with is what constitutes the service interface, and is described using WSDL. Services are typically exposed by components that plug into the core runtime. A service engine (SE) is a component that exposes a service of which it implements the business logic. Examples of service engines are the BPEL SE (orchestration), the IEP SE (complex event processing), and the EE SE (for business logic in Java the form of EJBs). Components are installed into the runtime, and can then be used by many different applications.

3.2.2 Apache ServiceMix

Apache ServiceMix [2] like OpenESB combines the functionality of a SOA and an Event Driven Architecture (EDA) to create an agile, enterprise ESB. There’s not much difference between Apache ServiceMix and OpenESB framework. Both
3.3 Warp

Warp (Web Connectivity Enabler) is an overlay network that provides its own addressing and messaging semantics for bidirectional web-based messaging. The key components of WARP are: addressability, reachability and asynchronous messaging. Addressability is important both on an application and component level; it allows entities on the network to know more precisely who they are talking to other than a given “host”. Reachability and asynchronous messaging provides both clients and servers with the ability to initiate conversation to each other without waiting for one or the other to take an initiative. For example, a server should be able to update a client’s status without the client requesting a status update. Warp identifies two different kinds of entities that need to communicate: Services, always-on network-based entities, and Clients, devices or applications that are potentially temporary and/or sporadically available. A Service is manually configured, does not change its state often, and is accessible without having to maintain an underlying network connection. Examples of Clients are mobile phones (which may lose network connec-
Figure 3.3: Warp

tivity), or computer applications (which need to maintain a TCP connection open in order to be reachable) [6]. The Warp Gateway is the most important component in the Warp architecture, since it represents a connection link between Clients and Services. Warp provides an API enabling JavaScript applications to send and receive messages with any other applications that use the Warp API. The drawback of this overlay network is exclusive reliance on JavaScript language, since the main idea of the DMB is to provide a transparent communication between applications and devices. Also, it is unclear how to extend Warp Deployment by introducing additional component.

The afore mentioned systems to some extent solve stated problem, but they either lack necessary functionality, or are too general by trying to capture as many use cases as possible.
Chapter 4

The Device Message Bus Design

4.1 Overview

In this chapter we will describe the Device Message Bus design. We will start from analysis of system requirements and provide a general description of how the system should work. Then we explain the component based approach selected for designing the system.

Sections 4.3.2 through 4.3.8 will provide detailed explanation of each system component. The structure of the DMB message is described in section 4.3.9. Finally section 4.3.10 will show overall system design and show an example how the component are interconnected.

4.2 System Requirements

The main functional requirement for the system is to provide a high-level communication interface that will allow different types of applications to communicate with different types of sensors, irrespective of the technology they use to communicate. The interface in its most basic form should allow a message bidirectional transmission between these two parties. The system must provide several transmission types: unicast, multicast, and broadcast. The system must provide support for at least WS, REST, and UDP communication protocols. From now on we will refer to an application and a sensor as a client and a resource respectively.

The basic functionality is subsequently augmented with the following requirements:

- **Extensibility.** The system must take in consideration future growth and modification of existing functionality with minimal impact to the rest of the system. Any change in system configuration or functionality should be applied without an interruption of the system. Example: Adding new communication type
(i.e. REST), to the system or modifying existing one should be seamless to the system. The system should apply new communication type during runtime.

- **Identity Management.** The system should support a naming scheme that will uniquely identify a resource in the DMB domain.

- **Reliability.** The system must provide a reliable communication between the clients and the resources. That is, any message sent by the client should be delivered to the resource. In case of message transmission failure the system should inform the client about the fault.

- **Security.** The system in its most basic form must provide a secure communication channel between the clients and the resources.

- **Management System.** The DMB must provide a management system that will give an administrator the ability to install system updates, modify system configuration, manage the resources, and monitor the system operation flow.

- **Resource Registration.** The system must allow manual and/or automatic registration of resources located in the DMB domain. Example: if a new resource is installed in the WSN it should be instantly available to the clients.

These are the fundamental requirements for the Device Message Bus and form a basis for the system design. In order to understand how these requirements will determine the DMB consider the following example.

There is a Wireless Sensor Network connected to the running Device Message Bus, which provides a message transmission between the clients and the resources (sensors, actuators) in this WSN. Consider a case where a number of new resources, supporting new communication protocol (i.e. SMS), needs to be installed in the WSN and made available to the clients. The system administrator should perform following steps to install the resources:

1. Install a new (SMS) communication handler (a system extension) via the DMB Management System. This will allow the DMB to understand how to communicate with new resources.

2. Identify new resources in the DMB via Management System by applying the Identity Management mechanism of the DMB (naming scheme).

The rest should be done by the Device Message Bus. The system should instantly apply a new communication handler, and after that provide an SSL communication for the clients. Upon discovering availability of new resources the clients (by using the newly created names for the resources) can send a message to new resources.
4.3. SYSTEM DESIGN

without any modification on their part. Any failure during message transmission
will be immediately highlighted in the Management System operation flow monitor,
and the client will be notified respectively.

The requirements elicitation phase showed that the system needs to be separated
into distinct features that overlap in functionality as little as possible. Typically this
is achieved through modularity of programming and encapsulation. The DMB will
consist of the features that are united by the system workflow, that is by employing
functionality of each other. This analysis is reflected in the chosen design approach
(see section 4.3.1), which is a **Component-based software engineering (CBSE)**, which
emphasizes the separation of concerns in respect of the wide-ranging functionality
available throughout a given software system.

We will see how the requirements and chosen design approach are reflected in
the system design in the following sections. The section 4.3.10 will describe how
the design is mapped to the system requirements.

4.3 System Design

The first step in designing the DMB is to choose the design approach, which will
define the basis for the architecture of the system. Once we define the approach, next
step is to create a design for the communication mechanisms. This step includes
designing the communication API and all necessary components that will form the
messaging system of the DMB. Further, we will map system requirements to the
chosen design and create corresponding architectural models that will provide a
foundation for the implementation of the system.

4.3.1 Component Based Development

The Device Message Bus is a compound system, since it connects in itself a wide
range of technologies such as communication types (REST, Web Services, SMS,
GPRS, IPv4, 6LoWPAN, etc), integration patterns, reliability mechanisms, security
protocols, etc. Thus, the first and foremost step is to create a model which will
consist of correctly organized interconnected components (modules). Constructed
in such way, the system will have following advantages over other approaches:

- Each component can be created and tested separately from the rest of the
  system. Any changes will affect only the component and keep the system
  intact.

- The components are replacable, that is a component can replace another (at
design time or run-time). Consequently, components can be replaced with
  either an updated version or an alternative one, without breaking the system.
It is evident that these properties satisfy the system’s *Extensibility* requirement. Indeed, component based development implies the future growth of the system, as well as modification of existing functionality.

### 4.3.2 DMB Connection Interface

As it was mentioned previously the DMB should provide a communication interface that will allow the sending of messages between the clients and the resources. We define two types of the interfaces, the *Application Connection Interface (ACI)* and the *Resource Connection Interface (RCI)* that will help to distinguish between the entry points into the system. From one side we will have an entry point for the clients, and from the other side an entry point for the resources (Figure 4.1). We introduced these interfaces in order to provide a uniform representation for the information transmitted via the DMB.

![Figure 4.1: DMB Connection Interface](image)

As shown on Figure 4.1 the separation of interfaces also keeps the multiplicity of connection types separate.

A client that sends messages through the system must prepare the messages so that they comply with the type of messages that the system expects. Upon receiving the message, the system transforms it into a format understood by the resource to whom the message is intended and sends it.

**DMB Communication Handler**

Since the main idea of the DMB is to provide transparent communication it is very important for the system to support as many communication protocols as possible. As the *Extensibility* requirement states, the system must support future growth, hence, in this particular case, the DMB architecture should support the dynamic adding of the communication protocols. Each communication protocol implementation should be considered as a DMB component, which can be added and removed during the run-time of the system.
4.3. SYSTEM DESIGN

A Communication Handler in the context of the system is defined as a pluggable component or a system plugin that provides a communication protocol (REST, WS, IPv4, etc.) implementation so that the clients and the resources could interface with the DMB. A Communication Handler must implement one of the interfaces (RCI, ACI) in order to be connected to the DMB.

Figure 4.2: DMB Plugin

Figure 4.2 shows two system plugins (REST, WS) connected to the DMB. Any system plugin consists of two independent modules: a client, and a server. A plugin server listens for incoming requests, and forwards them to the DMB. On its turn, a plugin client sends messages back to a caller (either DMB client or a resource). For example if the DMB client sends a message to a resource using RESTful communication, it sends the request to the DMB REST plugin server. When the DMB receives response from the resource it uses the REST plugin client to send this response back to the caller.

Both clients and resources can interface with the DMB only by using system plugins. Hence, a system plugin is also responsible for providing a secure communication channel between these parties and the DMB. The implementation of any communication protocol should also take into account the security aspects of information transmission.

Transmission Types

The communication mechanism of the DMB is designed to support following message transmissions types:

- **Unicast** - messages are send from one client to one resource, they are not intended for other resources. (Regular transmission mechanism.)

- **Broadcast** - messages are sent to every resource inside the DMB domain. (Can be used for sending a bootstrap message to the resources installed in the WSN.)
• **Multicast** - a compromise between two previous types. Messages are sent to a set of resources that meet particular criteria. (For example the client wants to retrieve information only from temperature sensors installed in the WSN.)

### 4.3.3 Message Router

While both ACI and RCI provide the communication interfaces for the clients and the resources, there is a need for an interconnection mechanism. More precisely, the following questions must be answered: how the client side of the system will communicate with the resource side; how the DMB will route message to the correct destination and use the right connection type?

![Figure 4.3: Message Router](image)

According to [9] a system that allows to consume a message from an input destination, evaluate some predicate then choose the right output destination is called a **Message Router**. The key benefit of using a Message Router is that all surrounding components in the system are completely unaware of its existence. Hence, changing the router will have no impact on the rest of the system. The DMB Message Router is stateless, in other words it only looks at one message at a time to make the routing decision. The decision criteria for the destination is based on the connection type that is used for the destination. For example, if the destination resource uses SMS based communication, a Message Router will route the message to that resource via SMS system plugin. Apart from deciding on the route, the DMB Message Router selects a message transmission type (unicast, multicast, or broadcast). This selection is done by the **Forwarding Strategy** component, which is represented on Figure 4.3. (Although the figure shows one way communication, that is from the client to the resource, the opposite communication has the same representation.)

So far we have described how the routing operation is handled by the Message Router.

By now we know that the Message Router component is located between the ACI and RCI. The interconnection between these components is provided by **Message**
4.3. SYSTEM DESIGN

Channels [9] - virtual pipes that connect a sender to a receiver. The ACI component writes a message to the channel whereas Message Router reads that message from the channel, decides on the route, and then writes the message to another channel, which is set up with selected connection module. Lastly, the connection module reads the message from the channel.

4.3.4 Identity Management

An important system requirement is that the Device Message Bus should provide an identity management mechanism that will uniquely identify a resource in the DMB domain. The basic idea behind this is to allow the control of access to the resources, create the hierarchies of resources and encapsulate the network details of a resource by providing the clients a simple virtual identifier of a resource.

Universal Resource Name

Each resource must have a unique identifier within the DMB domain it is used. This identifier is defined as a Universal Resource Name (URN) and represents a mapping from URN to a network address of a physical device (sensor). The resource identifier is created using information assigned during the manufacturing or commissioning of a device. In order to simplify the searching for resources the resource identifier URN is created including semantic information. DMB URNs have the following naming scheme for physical resources:

```
urn:dmb:[domain]:[device type]:[resource name]:[unique id or name]
```

where

- `urn:dmb` - the namespace identifier.
- `[domain]` - the domain of the provider.
- `[device type]` - describes the device or entity providing this resource. For example, the manufacturer and the type of a physical device.
- `[resource name]` - since a physical device may have many resources, this field differentiates between those. It might be the type or model number of a sensor.
- `[unique id or name]` - a unique number or a name differentiating devices with the same `[device type]`. For example, the UUID of a device, the HIT of a device, or a serial number.
Following this scheme, a temperature sensor resource could have the following name:

\texttt{urn:dmb:ericsson.com:SoniSenseN53:Temperature:4e3a-4d22-ad4v-e432}

When a resource is issued a unique identifier, the next step is to store this data, along with the low-level network address, in the \textit{Resource Database} (which will be described later). We do this in order to keep the information about a resource available for the clients and the DMB.

\textbf{Resource Groups}

In section 4.3.2 we mentioned that the Device Message Bus supports multiple message transmission types. Evidently, in order for a client to send a message to a particular resource (unicast) it is enough to specify the URN of that resource. Multicast implies the sending of a message from one client simultaneously to several resources. The communication process should be the same, regardless of the type of transmission, that is, both unicast and multicast should be identical for the client. Consequently, the multicast operation inside the DMB should be also identified by a URN.

We introduced a concept of \textit{Resource Groups} - a structure consisting of a collection of resources, that has two basic properties: a name and a identifier (URN). The URN of a Resource Group has following naming scheme:

\texttt{urn:dmb:mc:}[\texttt{group name}]:[\texttt{unique id or name}]

where

- \texttt{urn:dmb:mc} - the namespace identifier.
- \texttt{[group name]} - the name of the group.
- \texttt{[unique id or name]} - a unique number or a name differentiating groups.

A typical group URN has following form:

\texttt{urn:dmb:mc:MultiSet2:3d3q-1a26-2dg1-q4d5}

The main purpose of a Resource Group is to aggregate a number of resources and provide a unique identifier for them. Now the clients can send a multicast message to a set of resources by specifying the URN of the group to which the resources belong.
4.3.5 Resource Database

In section 4.3.4 we explained the identity management mechanism and its role in the DMB domain. When a resource is registered in the system, its network information is mapped to the URN, which is later available to the DMB clients, so that the latter could send a message to the resource. This mapping needs to be stored in a database in order for clients and the DMB to be able to access the same resource using the same URN.

Database Structure

The DMB Resource Database represents a persistent storage for information about the resources connected to the Device Message Bus. The information includes the afore mentioned mapping between a URN to a network address of a physical device. Detailed structure of the Resource Database is shown on Figure 4.4.

![Figure 4.4: Resource Database Table Structure](image)

**Group** table contains the definitions of the groups:

- **GroupURI** - a group identifier.
- **GroupName** - the name of a group.

**Resource** table contains high-level information about a resource:

- **URI** - a resource identifier (Universal Resource Name).
- **Status** - the status of a resource. This field indicates whether the resource is online or offline.
- **Attributes** - additional information about the resource, such as name, tags, etc.
- **GroupId** - a reference to the group to which the resource belongs.

**Connectors** table contains network information about a resource:

- **Typeld**
- **TypeName**
- **TypeClass**
• **ResId** - a reference to the resource.

• **Address** - the network address of the resource. For example, IP address, phone number, MAC address, etc.

• **TypeId** - a reference to the connection type of a resource.

• **Default** - indicates whether this connector is default for the resource.

Since today’s sensors are more than just a sensing elements, they can support multiple connection protocols. For example, a modern smartphone can communicate via SMS, GPRS, TCP, etc. Therefore, each resource inside Resource Database can have multiple **Connectors**, and the **Default** field specifies the default connection.

The **ConnectionTypes** table contains information about the pluggable modules of the system. These modules provide the implementation of communication protocols. It is important to keep this information separate from the **Connectors** in order to preserve loose coupling. As it was mentioned, the components of the system are replacable, hence if a connection type is changed, the modification will be reflected only in **ConnectionTypes** table, keeping the resource entries intact. The fields are:

• **TypeName** - the name of the Connection Type.

• **TypeClass** - the class name holding implementation of the connection.

**Database Interface**

When a client sends a message to a resource it specifies the URN of that resource. The DMB in its turn should route the message to the destination using the actual network address and the communication protocol of that resource. This information, together with the URN of the resource is stored in the Resource Database. Hence there is a need to provide the DMB with an access to the database. For this purpose we created a component called a **Database Interface**.

The **Database Interface** in its simple form allows the DMB to get the low-level information about the resource by specifying its identifier (URN). We represent this component as an interface in order to separate a database abstraction from implementation. For example, the Resource Database can be based on a relational database management system (RDMS), such as MySQL, Oracle, PostgreSQL, etc. each of which requires custom implementation for accessing the data. If the RDMS is changed the DMB will still be able to access new database using the same interface, which in its turn will employ different implementation of data access.
4.3.6 Logger

A *Message Logger* is an essential part of every messaging system. Its main purpose is to provide a utility that keeps track of any event occurred in the system. There are no restrictions on the types of events that can be processed by the message logger. For example, the DMB Message Logger must keep track of every processed message, system reconfiguration, any faulty behavior, such as a system exception, message delivery failure, system component failure, etc. Depending on the importance of an event, a log entry is assigned a level (priority). The DMB Message Logger defines three levels: INFO, WARNING, ERROR.

In order to preserve the loose coupling of the system, the Message Logger is designed as a component, where message logging interface is separated from its implementation. Thereby if the component is replaced, the rest of the system remains unchanged.

4.3.7 Scheduler

The DMB clients might require a systematic information retrieval from the resources, e.g. a client wants to measure a temperature value of a sensor every predefined interval of time. Clearly, the client can set up a periodical message sending via the DMB, but this approach has undesirable implications, such as message flooding in case when the time interval between client requests is short. Since every client request usually assumes a response from the requested resource, the above mentioned approach also implies the message flooding from the side of the resource. Though the system is expected to tolerate huge amount of requests, it is still reasonable to optimize the DMB in order to avoid redundant usage of system resources and a network traffic.

A *Scheduler* is the DMB component that allows clients to schedule their requests in the system at a predefined time. Using this component clients can send a message only once, and expect the system to periodically execute a message sending operation. The period of message sending is specified by clients in the details of the message. In order for a client to schedule a *job*, that is a message sending operation, the following parameters must be specified: job name and execution interval. When the Scheduler receives a scheduling request from a client it creates a *job group* which contains the jobs for that client. Further jobs from the same client will be added to this group. In order to modify existing job the client must specify the same job name and new job interval. If the interval parameter is set to 0, the job will be deleted from the job group. A group containing no jobs will be automatically deleted from the system.

The Scheduler component is designed is the same way as the Logger, that is the scheduler interface is separated from its implementation to preserve *Extensibility*
4.3.8 Management System

One of the requirements of the Device Message Bus is to provide a Management System that will give the system administrator the ability to control and monitor the DMB. The DMB Management System consists of three subsystems, independent of each other:

- **Resource Database Management System (RDBMS)** - provides the system administrator with ability to add / modify / delete resources in the Resource Database, as well as create resource groups and assign resources to a particular group. In order to communicate with the Resource Database this part of the Management System requires the Database Interface of the DMB.

- **Configuration Management System (CMS)** - provides the system administrator with ability to change the configuration parameters of the system. Moreover, this subsystem is responsible for managing the client & resource protocol plugins, that is install new plugin, initialize and start a plugin, remove existing plugin.

- **System Monitor (SM)** - shows the complete system log. During the system runtime the log can be redirected to the various outputs, such as a database, a file, or a console. Main requirement of the System Monitor is to read the log from the output and provide the latest (“live”) information to the system administrator.

Both, the RDBMS and the SM are separated from the DMB as they operate with the information that is not an integral part of the system. The CMS on its turn deal with the system configuration, hence it must be integrated with the DMB. The details of these systems will be described in the next chapter.

4.3.9 DMB Message

The *DMB Message* is information which is sent from a client to a resource, and vice versa. It contains following attributes:

- **Destination** - URN of the resource.

- **DestinationType** (Optional) - The type of connection used for communicating with a resource. If not specified, then default connection is used.

- **ReplyTo** - address of the client which receives requested information.
4.3. SYSTEM DESIGN

- **ReplyToType**: the type of connection used for communication with the client (REST, WS, etc.).

- **Payload** - data sent from a client to a resource.

- **Properties** (Optional) - additional properties, such as scheduling parameters, message delivery confirmation, etc.

4.3.10 Evaluation of Design

In previous sections we have covered all components of the Device Message Bus and provided detailed description for each of it. This section will cover overall design for the Device Message Bus.

Figure 4.5 shows the component diagram of the system. It is worth mentioning that the diagram shows the component composition, not the interconnection between components as such. Note the separation between the interfaces and their implementations of the following components: Resource Database, Logger, Scheduler, Management System.

We will analyze each system requirement to make sure that the designed system is in good agreement with them.

- **Extensibility.** This requirement is satisfied by creating a dynamic plug n play structure for the Communication Handlers of system as well as by using the component based development approach.

- **Identity Management.** This requirement is satisfied by creating a mapping from URN to network address of a resource and by introducing a Resource Database component.

- **Reliability.** The request/reply approach used for a message transmission between clients and resources ensures the fulfillment of this requirement. Indeed, if a message sending failed the system will notify the client with corresponding status (see next chapter for details).

- **Security.** Each connection type represented as a system plug-in must provide a secure communication channel between clients and resources.

- **Management System.** We have developed three subsystems: Resource Database Management System, Configuration Management System and System Monitor. These three subsystems give the systems administrator the ability to configure and manage the Device Message Bus as well as available resources.
• Resource Registration. The registration of new resources in the DMB domain is performed by the Resource Database Management System, which is part of the DMB Management System.

As we can see the proposed design for the Device Message Bus meets the system requirements. Not only it provides a foundation for creating the system, but also makes it easier to adapt the system to new requirements.

Figure 4.5: The DMB Component Diagram
Chapter 5

Object Design and Implementation

5.1 Overview

In previous chapter we described the system in terms of architecture, such as system decomposition onto components, global control flow, persistency and identity management. Next step is to decompose each component (with UML class diagrams) and create an object model for the system, in which we define each component in terms of the operations it provides. The implementation of the DMB is based on Java programming language and employs following technologies: Restlet framework, Apache CXF, Apache Commons, GSMA OneAPI SMS, Ericsson SMS Send & Receive, JBoss Netty, Glassfish application server, Java Servlets, Hibernate framework, Spring framework, JQuery library. The size of the DMB system together with currently supported plugins is around 15000 SLOC (source lines of code).

5.2 Component Types

Each DMB component is represented as an independent module of the system’s architecture, providing a functionality that, together with the functionality of other components defines the DMB as a complete system. Though these components have different designation, there is a need to create a common representation for them. The first and foremost property that unites them is that they belong to the same system - the DMB. We represent this as an interface in our object design called BaseComponent. This interface represents a component inside the Device Message Bus. Another property that unites the components is their operation state, that is a mechanism that indicates whether a component operates properly or is in erroneous state. Therefore BaseComponent has a single operation hasError(), which returns
a boolean value - the state of a component. If the value is true - the component is in erroneous state, otherwise it operates normally.

Some DMB components can be configured by parameters that define their behaviour. For example the Logger component can be configured to redirect a message log either to a database or a file. In order to support the component configuration it is necessary to extend BaseComponent with configuration mechanism. We achieve this by introducing Configurable interface that describes configurable components of the system and has an operation configure(Properties p), which takes a configuration parameters and applies them to a component.

![Figure 5.1: DMB Component Interface Diagram](image-url)
5.3. MESSAGING SYSTEM

Not only the components can be configured, but they can also be loaded or unloaded, e.g. if the a REST communication handler is unloaded, the system cannot provide a communication using REST protocol. Conversely, with loaded handler, the system will provide REST based communication. Loadable interface is an extension of BaseComponent and describes the components with afore mentioned functionality. This interface supports loading and unloading operations: load() and unload() correspondingly.

As it was described in previous chapter a Communication Handler is a dynamic component of the DMB, that is, it can be added or removed during system runtime. Since a Communication Handler consists of two independent modules: a client and a server, it is important to preserve this separation. We introduce two interfaces: Pluggable and Servable that represent handler’s client and server correspondingly. Both interfaces extend Configurable and Loadable, and support additional operations:

- **Pluggable** supports the most important operation of the DMB: sendMessage() which sends a message from one endpoint to another. getConfigurationFileName() operation returns the configuration file of a Communication Handler.

- **Servable** interface contains one operation: getStatus(), which indicates whether the server is started or stopped.

Figure 5.1 shows a relationship between described interfaces. These interfaces cover all types of components that form the DMB system. If new component implements BaseComponent interface, or any extension of it, this component becomes a part of the system.

5.3 Messaging System

5.3.1 Communication Handlers

In previous section we introduced two interfaces that describe communication handlers of the system: Pluggable to be used by the client and Servable to be used by the server. From section 4.3.2 we already know that resource side of the DMB is separated from the client side. Hence, this separation should also be applied to the interfaces that describe a communication handler. This can be achieved by introducing two types of handlers: a client handler, which is represented as ApplicationHandler interface and a resource handler - DeviceHandler interface. Both two interfaces extend Pluggable interface, but do not add additional functionality; they simply represent the separation between handlers. Note that the separation is done only with the client part of a handler (Pluggable interface), the server part
(Servable) remains unchanged. The reason behind this is that the server part of a handler only accepts incoming messages and doesn’t participate in the rest of a message passing process.

Pluggable is a generic interface, where T specifies the type of the PluggableContext. For communication handlers the PluggableContext represents a DMB message that is sent from one endpoint to the other.

Figure 5.2 shows the separation of communication handlers. When it comes to the creation of a communication handler, implementing one of these interfaces will define whether the handler belongs to the client or the resource part of the Device Message Bus. For example if we want to create an SMS handler, which will provide the communication means for the resources, we implement DeviceHandler interface to create an sms client of the handler. This client will send a message to a resource using sms based communication. Not only we want to send a message to a resource, but also expect this handler to receive a message from a resource. For this we need to implement Servable interface. Appendix A shows the listings of two sample classes that implement the afore mentioned interfaces.

---

Communication Handler Loader

So far we have learned how to create a communication handler for the DMB. But it is unclear how a handler is plugged into the system.

Generally, it is a good practice to represent a plugin of any system as a standalone class library that is loaded into the system during system startup or system runtime.
By employing this approach for the DMB we satisfy Extensibility requirement of the system. Each communication handler of the DMB is a plugin, represented as a class library, that has to be loaded in order for the system to provide a certain type of communication.

We introduced ComponentLoader module that uses following approach to support dynamic loading of the communication handlers: Suppose the class libraries containing an implementation of the communication handlers are located in the same directory. ComponentLoader parses this directory, gets all available plugins, extracts all classes from each of it. If a class implements a certain interface, this class is instantiated and the instance is added to the plugin container. This operation is performed for all three key interfaces of a communication handler: DeviceHandler, ApplicationHandler, Servable. Ultimately the ComponentLoader will provide three containers containing the instances of the plugins, where each container is a map with the key representing the name of a plugin, and the value representing its instance. Algorithm 1 shows the pseudocode for our method.

**Algorithm 1 Communication Handler Loader**

```
{Get all available plugins:}
plugins ⇐ ListPlugins(directory)
for all p in plugins do
    {Get all classes within a plugin:}
    classes ⇐ GetClassNames(p)
    for all c in classes do
        {Get the name of an interface that is implemented:}
        interface ⇐ GetInterface(c)
        if interface is a type of DeviceHandler then
            instance ⇐ initialize(c)
            devicesPlugins.add(instance)
        else if interface is a type of ApplicationHandler then
            instance ⇐ initialize(c)
            applicationPlugins.add(instance)
        else if interface is a type of Servable then
            instance ⇐ initialize(c)
            servers.add(instance)
        end if
    end for
end for
```
5.3.2 Router

The `Router` component inside the DMB allows to consume a message from one input, evaluate some predicate (forwarding strategy), then choose the right destination (see section 4.3.3). The `Router` extends `BaseComponent` interface and has two main operations: `routeMessage(Message m)`, which provides routing mechanism, and `pickStrategy(String urn)`, which given the address of an endpoint selects certain forwarding strategy (unicast, multicast, etc.). In order to provide two way message routing we distinguish two types of routers: `ApplicationRouter` and `DeviceRouter` (Figure 5.3). Both routers have different evaluation criteria, and different routing mechanisms, hence the separation. One of the main differences between these two routers is that the `DeviceRouter` also needs to fetch resource details from the database.

![Diagram of Router](image)

Figure 5.3: Message Router

The `Router` represents an entry point to the core of the DMB system. Indeed, a
client connects to a communication handler, which accepts incoming messages, via the server part of the handler. Then the message is passed to the *Router* using the \texttt{routeMessage(Message m)} method. This component can be viewed as the *Facade* design pattern [4], that is, it encapsulates the internals of routing mechanism and provides simple interface to a developer of a communication handler.

### 5.3.3 Forwarding Strategies

When a message is passed to the DMB Router, the latter has to decide on the message forwarding strategy. The *Strategy* design pattern [4] provides a perfect match for describing this decision making mechanism. Indeed, the *Strategy* is intended to define a family of algorithms, encapsulate each one, and make them interchangeable. The factors that define which algorithm to use are not known until run-time. The algorithms in our particular case are the message transmission types supported by the DMB (see section 4.3.2). The selection of a transmission type is done only after analyzing the arrived message.

Figure 5.4: Forwarding Strategy

Figure 5.4 represents a *ForwardingStrategy*, which defines an interface common to all supported message transmission types. *UnicastStrategy, MulticastStrategy* are concrete strategies that implement certain transmission type. The *ForwardingCon-
text uses ForwardingStrategy interface to call the algorithm defined by a concrete strategy.

It is worth to mention that MulticastStrategy can support both sequential and parallel execution message sending. Current implementation of this strategy is based on sequential execution of the strategy.

5.3.4 Message Channels

After selecting the forwarding strategy the system has to open a communication channel to the destination and send a message through the channel. There are two types of channels: one for sending a message to a resource, another for sending a message to a client. Both types have operations for opening a channel, configuring, and closing a channel. Upon initializing a channel the system selects a certain communication handler in order to send a message to an endpoint. Hence the difference between channel types: the resulting communication handler is an ApplicationHandler or a DeviceHandler.

![Message Channels Diagram]

Figure 5.5: Message Channels

Figure 5.5 shows the structure for the message channels. The BusChannel inter-
face represents a general message channel and contains operation for opening and closing a channel. The channel configuration mechanism is achieved by extending Configurable interface. The types of channels are represented by the Application-Channel and DeviceChannel interfaces. Both channel return a reference to a handler which insures a certain type of communication.

5.3.5 Database Interface

When the DMB receives a message from a client, the destination, that is a resource, is represented as a uniform resource identifier. From section 4.3.4 we know that the system needs to route the message to the destination using actual network address of the resource. The DatabaseInterface provides this functionality. The interface extends BaseComponent and has two main operations:

- Address getResourceAddress(String urn, String type) - given the urn of the resource and optionally the name of a connection type, returns the address of that resource. If no type is specified, the method selects default connection type of the resource.

- Address[] getResourceAddressGroup(String groupUri) - returns a collection of addresses of the resources belonging to the specified group.

The URN of the resource inside a message is later overwritten with the actual network address.

5.3.6 DMB Message

The DMB has two separate parties that are communicating via messaging using a communication handlers. The transmitting information is packaged into a DMB Message, which has XML format and contains the fields described in the section 4.3.2. Listing 5.3.6 shows sample XML message sent from a client to the DMB. It is worth to mention that the payload entry of the message contains a byte array of the data sent to either a client or a resource.

Listing 5.1: DMB Request Message

```xml
<request>
  <replyTo>https://localhost:8183/app/rest</replyTo>
  <replyToType>rest</replyToType>
  <destination>urn:dmb:ericsson.com:pf0at:sq:4</destination>
  <destinationType>ipv4</destinationType>
  <payload>PGNvb1hbmQPHJlcGx5VG8aR0cHM6Ly9sb2</payload>
```
Upon receiving a message from a client a DMB communication handler deserializes the XML message and instantiates a `Message` object. Every instance of a `Message` is assigned a unique identifier, which is sent back to the client. Using this identifier a client can track the status of the request.

### 5.3.7 DMB Messaging in Action

In order to understand the operation of the DMB messaging system, consider simple scenario: A client wants to send a message to a resource using the Device Message Bus. The client uses REST based communication and has no knowledge about the type of communication used by the resource. The only information about the resource the client has is the resource URN. In order to send a message to this resource the client connects to the DMB RESTful handler and sends a message. We consider only one way communication, that is from a client to a resource, since the same message sending mechanism applies to both parties.

When the RESTful handler receives a message it initializes the `DeviceRouter`, and hands over the message. The `DeviceRouter` gets information about the resource from the database, replaces the URN with the actual network address of the resource, picks a certain forwarding strategy and executes it. In our case, selected strategy is a `unicast`. When the strategy is executed, the DMB opens a `DeviceChannel` and gets the instance of the SMS communication handler used for this message transmission. The handler then sends the message to the resource. After the message has been sent, the DMB closes the `DeviceChannel`. At this stage the message sending operation is complete. Figure 5.6 shows the sequence diagram of this scenario.
Figure 5.6: Message Sending Sequence Diagram
5.4 Database Manager

In section 4.3.5 we provided detailed explanation for the structure of the Resource Database. We also introduced a DatabaseInterface - a component that provides the DMB with an access to the Resource Database.

A relational database management system used for providing an access to the Resource Database is MySQL [14]. Database Manager on its turn provides an implementation for the DatabaseInterface and is developed as a separate component. The implementation of the Database Manager is based on the Hibernate [3] - an object-relational mapping (ORM) library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. Mapping Java classes to database tables is accomplished through the configuration of an XML file. The Database Manager contains four mapping files each of which represents a respective table in the Resource Database. Using the Hibernate mapping helps to keep object-oriented domain model separate from the type of a relational database. The Hibernate library encapsulates database specific functionality, and provides high level API for operating with a database. The Database Manager component uses this API to fetch information about the resources in a transparent way. Although this information has to be aggregated from all four tables, the Hibernate association mechanism allows to unite the information into a single object by only specifying the type of relationship between the tables (one-to-many, many-to-many, etc.). Ultimately, the result of a database transaction is an object containing all necessary information about a resource.

5.5 Logger

Message logging system of the DMB is divided into two parts: Logger Interface and Logger (see section 4.3.6). We base our implementation of both parts on third party libraries.

For Logger implementation we use Apache log4j logging framework [8]. One of the distinctive features of log4j is the notion of inheritance in loggers. Using a logger hierarchy it is possible to control which log statements are output at arbitrarily fine granularity but also great ease. Hence it is possible to control DMB log levels: INFO, WARNING, ERROR. log4j supports many log output targets: a database, a file, an OutputStream, a java.io.Writer, a remote log4j server, a remote Unix Syslog daemon, and many other. The log output target for the DMB is a database. The reason behind this is that we need to have a structured representation of each log entry, which will help us to create a log analyzer and a log filter for the DMB Management System.

Logger Interface is developed using Simple Logging Facade for Java (SLF4J)
5.6 MANAGEMENT SYSTEM

[16] a simple facade or abstraction for various logging frameworks. SLF4J allows the end-user to plug in the desired logging framework at deployment time. Hence, using this facade the substitution of a Logger is done transparently to the DMB system.

5.6 Management System

The DMB Management System is implemented as a web application consisting of three modules, each representing a subsystem (RDBMS, CMS, SM) described in section 4.3.8.

RDBMS

The Resource Database Management System module inside the Management System allows the system administrator to manage resource groups, the details of each resource and the connection types available in the DMB. It reuses Database Manager component of the system with some several extensions. These extensions include adding/removing a resource, creating/deleting a resource group, assigning a resource to a certain group, and adding/removing a connection type. Since the Database Manager provides a read-only access to the Resource Database, together with these extensions we get a complete management module for the database.

CMS

The Configuration Management System is used to modify the configuration parameters of the DMB, install or uninstall system communication handlers and load or unload the handlers. It is very important that any changes occurred in the system must be instantly applied during system runtime. Thus, the CMS should have direct access to the initialized components of the DMB. To do so, we base the implementation of the CMS on Java Management Extensions (JMX) technology.

The JMX technology provides a simple, standard way of managing resources such as applications, devices, and services. Because the JMX technology is dynamic, it can be used to monitor and manage resources as they are created, installed and implemented. Using the JMX technology, a given resource is instrumented by one or more Java objects known as Managed Beans, or MBeans. These MBeans are registered in a core-managed object server, known as an MBean server. The MBean server acts as a management agent and can run on most devices that have been enabled for the Java programming language [13].

We created a DMBManagerMBean module containing operations for configuring the DMB as well as the system plugins. This module is registered in an MBean server, which allows CMS web application to manage system configuration
by employing the DMBManagerMBean operations. Any modification to the system configuration leads to automatic reloading of configuration parameters.

Another advantage of using the JMX technology is that a published MBean can be employed by any application. For example, we can build a system self-configuration mechanism on top of the DMBManagerMBean, that will modify the configuration parameters based on the conditions of the system.

SM

The DMB System Monitor shows the complete system log, which is stored in a database. Current implementation of the SM supports the filtering mechanism of the log, that is the selection of certain log level.

5.7 Communication Handlers

Currently the DMB supports 5 communication protocols: REST, Web Services for the client side of the DMB, and UDP, GSMA OneAPI SMS, Ericsson SMS Send&Receive for the resource side. Each protocol implementation is represented as a communication handler in the DMB consisting of two parts: a client and a server (see section 4.3.2).

REST

The implementation of the REST communication handler is based on the Restlet [19] - a RESTful Web framework for Java, which allows to build both RESTful services and clients. The biggest advantage of this framework is that it supports building RESTful services that run on embedded web servers by providing multiple HTTP server connectors. Currently the server part of the REST communication handler is built on Simple framework - a lightweight embedded HTTP web server. The server hosts one REST service which accepts POST requests from the clients. The client part of the REST communication handler uses Restlet to connect to a RESTful services.

Web Services (WS)

WS is another communication handler built to accept messages from the clients. The implementation of this handler is based on Apache CXF framework [7], which like the Restlet framework supports embedded HTTP web servers. The structure of the WS handler is very similar to the REST handler.
5.7. COMMUNICATION HANDLERS

UDP

UDP is a communication handler designed to support UDP based communication with the resources. The implementation is based on the *Netty* framework [10] - a NIO client server framework which enables development of network applications such as protocol servers and clients. Again, the communication handler contains the client and the server for serving requests via UDP based connections.

Ericsson SMS Send & Receive

Ericsson SMS Send & Receive [5] is a web API that gives the capability to send and receive SMS messages using a RESTful API. The SMS messages are sent via RESTful service using standard HTTP GET or POST requests. When the Ericsson REST server receives a message from the DMB it forwards it to the resource specified in the message by the MSISDN identifier.

GSMA OneAPI SMS

GSMA OneAPI SMS [1] allows an application to send and receive SMS messages using RESTful and Web Service APIs. Current implementation of this communication handler is based on RESTful API. The procedure of sending and/or receiving a message is the same as for Ericsson SMS Send & Receive API.
Chapter 6

Evaluation

6.1 Overview

This chapter covers the evaluation part of the work. We focused on three major types of tests: Functional Tests - to evaluate correctness of the system; Reliability Tests - to evaluate fault-tolerance, and Performance Tests - to evaluate performance characteristics of the system.

6.2 Case Study Scenarios

To evaluate the correctness of the system we examined some real-world scenarios. Firstly, we need to make sure that the system is able to install a communication handler and apply its configuration, as well as start the handler. Secondly, we need to test the types of a message transmission, that is a unicast and a multicast.

6.2.1 System Bootstrap

We start with an empty distribution of the DMB, which includes only the system management extensions. By starting the Management System we can access configuration management page and specify the directory, which contains available communication handlers. We consider five system plugins: rest (REST), ws (Web Services), oneapisms (GSMA OneAPI SMS), ericssonsms (Ericsson SMS Send & Receive), udp (UDP). After specifying the plugin directory we immediately see the list of available plugins for the system. Next step is to install them and load the server part of each plugin. As expected, the system installed specified communication handlers and started all servers of the plugins. The log output of system showed that all five plugins were applied with no error. As the result, the DMB launched five servers which can handle five aforementioned communication protocols.
6.2.2 Unicast

After the system has successfully installed the communication handlers we can start the message transmission operations. We tested a unicast using the following communication handlers: `rest` plugin for the client side and `ericssonsms` plugin for the resource side. For this purpose we created a sample REST client to send a request to the DMB and a sample REST server to accept the response. To simulate a resource behaviour we developed an Android application, which listens for incoming SMS messages, checks whether an arrived SMS is a command message, and outputs the message to the phone console. The intention of this test is to check whether the system delivers a message to the specified destination. Hence we expect the DMB to accept a message via `rest` communication handler, choose correct forwarding strategy, which is a unicast, and route the message to the `ericssonsms` communication handler. The latter should send the message to the resource and, based on the response from Ericsson SMS server, respond with the delivery status back to the client using the `rest` communication handler.

![Diagram of Unicast](image)

Figure 6.1: Unicast

After running the test we obtained the expected result. The REST client has successfully sent a message to the DMB and got the message identifier as the response. By examining the log output we obtained that the system has selected correct forwarding strategy and has routed the message to the `ericssonsms` plugin, which in its turn has sent the message to the Ericsson SMS server. The latter responded with the OK status, meaning that the operation has completed successfully. The DMB has sent this response back to the REST client by connecting to the client’s RESTful service. Since the whole communication process is asynchronous, we could observe the message delivery to the Android device after small delay. The console output contained the message sent by the REST client. Thus, the test showed that the system is capable of performing a unicast message transmission.
6.2. CASE STUDY SCENARIOS

6.2.3 Multicast

In order to test a multicast message transmission we employed four communication handlers: *rest* and *ws* for the client side and *udp* and *oneapisms* for the resource side. For this scenario we used a sample REST client to send a request to the DMB and a sample WS server to accept the response. To simulate a UDP based communication with a resource we created a sample UDP client and server. The GSMA OneAPI SMS service provides a sandbox for testing purposes, which accepts the requests and returns the delivery status (The reason behind using the sandbox is because currently the service does not support Swedish mobile networks).

![Figure 6.2: Multicast](image)

As the result we created a scenario with one client that uses REST protocol to send a multicast message and WS protocol to receive a delivery status from the resources. The scenario involves two simulated resources, which use UDP and OneAPI SMS communication protocols.

We expect the DMB to accept a message from the client via *rest* communication handler, choose a multicast forwarding strategy and route the message to the *udp* and *oneapisms* communication handlers. We expect these handlers to send the message to the simulated resources. The resources, in turn, should create the replies and send them to the DMB. The DMB should treat these replies as independent processes and deliver them to the client using *ws* communication handler.

After running the test we obtained the expected result. The client successfully sent a message to the DMB and got the message identifier as the response. The log output showed that the system has selected a multicast forwarding strategy and has routed the message to the *udp* and *oneapisms* plugins. The plugins have sent a message to the simulated resources. Upon receiving a message from the DMB each simulated resource has sent a response message to the DMB. The latter delivered two responses to the client using *ws* communication handler. Thus, the test showed that the system is capable of performing a multicast message transmission.
6.3 Fault-Tolerance Tests

The intention of the fault-tolerance testing is to examine the behavior of the DMB when some of the components of the system fail to work properly. We considered five test cases to check whether the DMB is resistant to failures.

Faulty XML Messages

In this test we created two faulty XML messages: not well-formed XML message, and XML message with a missing field. After running these tests the DMB rejected the messages and returned as the response the error code and the description of the error. Our main focus during the tests was on communication handlers, since they are responsible for the deserialization of an XML message and creation of the Message object. During performing the tests we observed that no system component has failed to work properly.

Database failure

It is very important to test the behavior of the system in case of Resource Database failure, since with the offline database the DMB cannot map the URN to the actual network address of a resource. Thus, a message cannot be delivered to the proper destination. To simulate this scenario, we stopped the Resource Database server and initiated a message transmission. As it was expected, the DMB failed to fetch the resource information from the database, and could not proceed with the operation. Although, the DMB failed to deliver a message to a resource, the system responded to the sample client with the corresponding error code and the description, informing the client, that the database is offline. After bringing the Resource Database back online, we were able to send a message to the specified resource. Hence, the database failure did not affect the rest of the components of the system.

Non-existing resource

A client may try initiate communication with the resource that is not registered in the DMB domain, that is, there is no information about the resource in the Resource Database. By performing this test we wanted to make sure that addressing a non-existing resource does not affect the operation of the Device Message Bus. To do so we created a request message containing the URN of a non-existing resource. The result of the test showed, that the system can cope with such cases, and inform the client that the specified resource does not exist.
6.4. PERFORMANCE MEASUREMENTS

Offline resource

In this test we examined the behavior of the system in case when a client wants to address an offline resource. As in previous test, the system could handle this case, and responded informed the client that the resource is offline.

Non-existing communication handler

Lastly, we examined the system when a client wants to send a message to a resource using unsupported communication protocol. We observed that the Message Router component of the DMB threw an exception while trying to initialize the non-existing communication handler. Although an exception was thrown, it did not affect the operation of the system. The exception description was generated and sent back to the client. After specifying the existing communication protocol and sending a message to the DMB, the latter routed the message to correct communication handler, which in turn delivered it to the specified destination. The same behavior was observed after trying to send a message from a simulated resource to a client using non-existing communication handler. The only difference in this case was that the system did not reply with the failure information to the client.

6.4 Performance Measurements

In order to evaluate the performance characteristic of the system we have created two test scenarios.

1. Scenario 1: One client sequentially sends 1000 messages to a particular resource using RESTful communication protocol. The DMB delivers a message to a resource using GSMA OneAPI SMS protocol.

2. Scenario 2: 10 clients sequentially send 100 messages in parallel using the same protocols as in first scenario.

We used Apache JMeter software for simulating these scenarios. The computer used for running these tests has a 2.53GHz Intel Core 2 Duo processor, 4.0GB of RAM and runs MS Windows Vista Enterprise 32-bit operating system. We realize that the hardware performance will greatly affect these measurements, but we still claim that the results can be of interest. The results are represented in Table 6.1.

Figures 6.3 and 6.4 represent graphs that plots all sample times: the current average of all samples (blue), the current standard deviation (red), the current median (violet) and the current throughput rate (green) are displayed in milliseconds.

Our main focus during the simulation of both tests was on the value of throughput, which represents the actual number of requests/minute the server handled.
<table>
<thead>
<tr>
<th>Scenario</th>
<th># Samples</th>
<th>Average (ms)</th>
<th>Std. Dev. (ms)</th>
<th>Error %</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario 1</td>
<td>1000</td>
<td>126</td>
<td>220.10</td>
<td>0.00%</td>
<td>7.9/sec</td>
</tr>
<tr>
<td>Scenario 2</td>
<td>1000</td>
<td>2493</td>
<td>1450.25</td>
<td>0.00%</td>
<td>4.0/sec</td>
</tr>
</tbody>
</table>

Table 6.1: Performance Measurements

In case of first scenario simulation showed that the DMB could handle almost 8 requests per second, while in case of second scenario the value of throughput has dropped to 4 requests per second. During the simulation the source code of the system was profiled and monitored by NetBeans Profiler tool. As the result we observed that almost 66% of the system load was because of the database operations that are used for fetching the resource information from the Resource Database. Hence we concluded that the values of throughput we obtained during the simulation is mainly due to the database operations. Another supposition is that the embedded server used for accepting the requests is not fully capable of tolerating large amount of requests.

Figure 6.3: Single user, 1000 requests
6.5 Discussion

The evaluation phase of the work gave us promising results. The Device Message Bus proved to work correctly under real-world conditions. We were able to bootstrap the system, and after that start a unicast and multicast message transmissions using all five developed communication handlers. Testing the system with a faulty data has showed that the DMB does not come offline, and any failed operation is atomic, that is, it does not affect the subsequent system workflow. The performance tests showed that system is capable of serving not that many simultaneous requests. The observed throughput value was only 8.0/sec in case of one client, and dropped down to 4.0/sec in case of 10 clients. This is a subject for further consideration.

Figure 6.4: 10 users, 100 simultaneous requests
Chapter 7

Conclusions / Future Work

This Master thesis work is a foundation for future research and development, and in this chapter some suggestions on future work will be discussed. This is done to verify that the goals have been accomplished, and hence, that the requirements of the original specification have been met.

7.1 Conclusion

We have designed and developed a system that acts as a bridge between applications and sensor networks. The main goal of the system is to provide a transparent way for the sending of messages to the sensors, irrespective of which networking technology the sensors use to communicate. The DMB fully addresses this goal and meets all the requirements identified during the design phase. Apart from that the current implementation of the DMB supports following communication protocols: REST, WS, UDP, GSMA OneAPI SMS, Ericsson SMS Send & Receive. During the work we have realized that there are many features that need to be added, and many things that could be done differently to increase system usability. We are aware of that the system is not complete, but it is a working solution with great potential.

The evaluation showed that during regular operation flow the system behaves as expected. It also tolerates the faulty cases when some of its components are are not available or are in erroneous state as well as the cases when the requests are not formed correctly. One of the main strengths of the system is that it does not crash in any case. If one operation has not completed successfully, the result has no affect on another operation that runs simultaneously or is about to start. Although the correctness and fault-tolerance has showed very promising results, the system needs large-scale performance, scalability and load testing. Current performance tests showed that the system can tolerate 8 messages per second. This result can be largely improved, which is subject to future work.
Most of our conclusions consist of ideas how to improve and enhance the system, these ideas are presented later in this chapter.

7.2 Performance Improvements

The result obtained during evaluation phase showed that the system can tolerate small amount of messages. The code profiling showed that the most of the system load was because of the database operations. This fact can be due to incorrectly configured parameters of Hibernate framework such as caching mechanism, lazy loading, etc. Apart from that the Resource Database can be optimized as well by creating a database index for Resource table. These are the obvious reasons for causing such a small throughput value. More tests have to be performed in order to reveal the weak points of the system.

7.3 Communication Handlers

In order for the DMB to provide a transparent communication to different types of applications and sensors it must support variety of communication protocols. Currently the system supports five protocols: REST, WS, UDP, Ericsson SMS, OneAPI SMS. The list of supported protocols must be constantly replenished, thus a very important step towards system extension and improvement is the implementation of communication handlers. Next on the list of protocols to support are: Constrained Application Protocol (CoAP), Low power Wireless Personal Area Networks (6LoWPAN), Ericsson Internet Payment Exchange (IPX), Transmission Control Protocol (TCP), etc.

7.4 Authentication, Authorization & Accounting

Current design and implementation of the Device Message Bus supports only security for communication networks such as Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL) allowing applications to communicate with sensors in a way designed to prevent eavesdropping and tampering.

Apart from secure communication, the system needs to support an authentication mechanism - the process of verification of an entity’s identity. This will allow only authenticated clients to send a message to the sensors. Even if a client is authenticated, it may have some restrictions on sending a message, for example a client may be allowed to send a message only to a specific sensor, and has no right to send a multicast message. For this purpose the system must support an authorization mechanism - a function for defining an access policy. With the support
of this mechanism the DMB will provide the system administrator with ability to set access rights to the clients and resources. A service provider might require the tracking of the consumption of sensor data by the clients. This information may be used for management, planning, billing, or other purposes. For this, the DMB must support an \textit{accounting} mechanism - the process of gathering the information about the user of the system, the type of data it requests, when request is started and when it is ended.

These three security mechanisms are united into one definition - an AAA (Authentication, Authorization & Accounting) protocol. The future work related to security aspects of the system will be associated with the selection of an implementation of an AAA protocol, and its incorporation into the DMB.

### 7.5 OSGi Support

The Device Message Bus has custom mechanism for supporting the management of a component, such as a communication handler. The system supports only those components that are designed and implemented specifically for it. Although this mechanism works well and provides a developer with easy and flexible API, it is still important to follow the established standards and simply avoid reinventing the wheel.

The OSGi framework is a service platform designed specifically for the Java programming language that implements a complete dynamic component model. Using OSGI the applications can be remotely installed/uninstalled, started/stopped without requiring a system reboot. By making the DMB OSGi “compatible” we will add the features such as component reuse, versioning, dynamic loading, etc. to the existing functionality:

### 7.6 JBI Support

The Device Message Bus is a standalone system, allowing the applications to send the messages to the sensors. In the future we imagine that the DMB will be integrated into a larger system offering more complex and richer services to the end-user. Thus the DMB needs to support an integration mechanism in order to become a part of more complex system. \textit{Java Business Integration (JBI)} is a specification that provides a pluggable container that hosts service consumer and producer components. By implementing the JBI we will allow the DMB to register itself in another system and participate in its workflow.
7.7 DMB Federation

Typically we expect that a service provider will host one instance of the DMB for providing an access to the deployed sensors. There might be a case when several service providers would unite their networks thus sharing the sensor information. Hence the DMB needs to support such cooperation. This is achieved by employing the concept of an identity federation. Having this the DMB Federation will enable the portability of resource identity information across multiple DMB domains. The ultimate goal of the DMB federation is to enable users of one DMB domain to securely access sensor data of another DMB domain seamlessly, and without the need for redundant user administration.
Listing 1: DeviceHandler implementation

```java
package com.ericsson.dmb.bci;

import java.util.Properties;

/**
 * Sample sms handler device plugin for DMB
 */
public class SMSSHandler implements DeviceHandler {

    public void sendMessage(PluggableContext<Message> pc) throws DMBException {
    }

    public void load() throws DMBException {
    }

    public void unload() throws DMBException {
    }

    public boolean hasError() {
        return false;
    }

    public void configure(Properties p) {
    }

    public String getConfigurationFileName() {
        return null;
    }
}
```
package com.ericsson.dmb.bci;

import java.util.Properties;

/**
 * Sample sms listener device plugin for DMB
 */
public class SMSListener implements Servable {

    public int getStatus() {
        return Status.STARTED;
    }

    public void load() throws DMBException {
    }

    public void unload() throws DMBException {
    }

    public boolean hasError() {
        return false;
    }

    public void configure(Properties prp) {
    }
}
Bibliography


