Applying Web Services as Middleware for Integrating Embedded Systems in Loosely Coupled Environments

Michael Sig Birkmose
birkmose@cs.auc.dk

Lars Haugaard Kristensen
larsk@cs.auc.dk

Jakob Møbjerg Nielsen
vitz@cs.auc.dk

Hans Hvelplund Odborg
odborg@cs.auc.dk

January 7th, 2004
Title:
Applying Web Services as Middleware for Integrating Embedded Systems in Loosely Coupled Environments

Project period:
DAT5 / F9S,
Sep 1st 2003 - Jan 7th 2004

Project group:
B1-215

Group members:
Michael Sig Birkmose
Lars Haugaard Kristensen
Jakob Møbjerg Nielsen
Hans Hvelplund Odborg

Supervisor:
Mikkel Christiansen

Publications: 6
Pages: 87

Abstract:
This report investigates the applicability of Web services as a middleware technology to integrate distributed embedded systems with other computing systems in a loosely coupled environment. The Web service technology is analyzed, along with the technologies it is based upon.

The main contribution of this report is the design of a framework for creating Web services for distributed embedded systems in a transparent manner, using the C programming language. A choice is made to focus on mapping the C representation of data types to the Web service representation, and to focus on the design of the architecture of the middleware code that the framework generates. This architecture is designed to maintain a minimum CPU utilization and memory consumption.

A prototype of the middleware code generated by the framework is implemented for the eCos operating system, and a performance evaluation of this implementation is carried out. The evaluation reveals acceptable results in terms of processing time of request-response message exchanges, and in terms of the memory consumption. The prototype implementation is found to be scalable in relation to the size of the data exchanged between the peers.

Based on the performance evaluations, it is concluded that Web services is applicable as a middleware technology for loosely coupled integration between distributed embedded systems and other computing systems.
The capability to continually add more transistors into smaller devices, while simultaneously decreasing their power consumption and cost, is changing the way computers are used in everyday life. Together with the advances made in low-cost connectivity through wired and wireless communication among computational devices [4], this has led to an increasing demand for connectivity among stand-alone embedded devices, such as Personal Digital Assistants (PDAs) and mobile phones. New application areas arise as these devices become connected to existing networks. Examples include mobile phones connected to online phone books, and GPS equipped PDAs connected to online map repositories. In order to simplify the development of such systems, software that handles challenges such as heterogeneous hardware and operating system platforms is already available. The abstractions and infrastructure that support this development is known as middleware.

As Gustavo Alonso et al. states [1], historically, middleware was introduced because information systems transformed from being single server solutions to being distributed between multiple servers. This happened because the systems were becoming too large and complex to reside on a single server, and because there was a desire to integrate multiple distributed resources into single information systems. Such systems are closely-coupled with complex interactions between the different parts of the system.

Middleware is still applied when creating complex information systems utilizing distributed resources. However, a new type of system is becoming more common. Information systems are now used as building blocks allowing one organization to offer a information system as a service to be used in information systems created by other organizations. This introduces new integration challenges. In order for a client to access multiple heterogeneous services from different service providers, it has to support several middleware platforms. Similarly, service providers face the challenge of choosing the middleware platform that most clients support. When implementing middleware on embedded devices, resources are often very limited. An embedded device acting as a client of one of these services may not have the option of implementing multiple middleware standards due to these constraints.

A solution to this challenge is that service providers agree upon one single middleware technology for offering their services. As Gustavo Alonso et al. states [1], attempts to agreeing upon such a technology has historically failed, because the suggested technologies, were too complex or because they simply never became widespread enough. Inspired by the well-established architecture of the World Wide Web, the World Wide Web Consortium (W3C) has introduced Web services as a middleware technology which is designed to enable communication between loosely-coupled systems in a decentralized distributed environment. Web services is based on well-established technologies such as the Extensible Markup Language (XML) and the Hypertext Transfer Protocol (HTTP). Web services do not attempt to replace existing middleware solutions used to build closely-coupled systems. Rather, they are meant to be built on top of existing applications or middleware as a mechanism to expose the services provided in a lightweight manner.

Web services is already widely used, e.g. the Globus Alliance team has chosen Web services as the basis for the Globus Toolkit [28]—a tool used for building computational
grid systems. The Globus toolkit uses Web services to expose the services offered by the different entities in the grid system, executing on heterogeneous hardware platforms and operating systems.

In this report, it is investigated how Web services can be applied in embedded systems. The aim is to investigate whether Web services is a well-suited and an applicable technology to integrate embedded systems with other systems. The Web services technology is analyzed, and a prototype of a framework for developing Web services for embedded systems is designed and implemented. As a basis for performance evaluations a simple Web service application is developed using this framework.

The report uses square brackets for citations, e.g. [24]. Text in type-writer font signifies code fragments, e.g. `void main(void)`. A description of key terms is provided in the end of the report for reference.
Contents

1 Introduction 3
  1.1 Embedded Devices and Systems ................................. 3
  1.2 Communication and Middleware ................................ 6
  1.3 Web Services .................................................. 12
  1.4 Summary ....................................................... 16
  1.5 Problem Statement ............................................. 16

2 Analysis 19
  2.1 Extensible Markup Language ................................... 19
  2.2 XML Namespaces ............................................... 21
  2.3 XML Schemas .................................................. 23
  2.4 SOAP .......................................................... 24
  2.5 Web Services Description Language .............................. 32
  2.6 XML Parsers ................................................... 36
  2.7 Summary ....................................................... 38

3 Design 39
  3.1 Design of Framework ........................................... 39
  3.2 Creating Web Services .......................................... 46
  3.3 Target Application ............................................. 51
  3.4 Summary ....................................................... 59

4 Implementation 61
  4.1 Common Code .................................................. 61
  4.2 Service Specific Code ......................................... 67
  4.3 Summary ....................................................... 70

5 Performance Evaluation 71
  5.1 Success Criteria ............................................... 71
  5.2 Test Design ................................................... 72
  5.3 Test Environment ............................................... 73
  5.4 Test Results ................................................... 73
  5.5 Summary and Discussion ....................................... 76

6 Conclusion and Future Work 79
  6.1 Results ........................................................ 79
  6.2 Future Work ................................................... 80

Abbreviations 83

Bibliography 85
The aim of this chapter is to introduce and describe the issues related to using Web services as a middleware technology to integrate embedded systems with other computing systems.

First, the concepts of embedded devices and systems are introduced, in order to give a clear understanding of these. Furthermore, definitions of three different classes of embedded systems are given to establish what type of embedded systems are in focus in this report.

Second, a description of the evolution of information systems is given, in order to understand why the demand for middleware arose, followed by a formal definition of middleware. Additionally, an abstraction of middleware, decomposing it into multiple layers, is given, in order to give an understanding of different types of middleware and to establish what type of middleware Web services is. Next, the challenges that heterogeneous middleware technologies are facing in the area of cross organizational application integration are introduced. This is followed by a description of how Web services strive to solve this problem along with an overview of the technical architecture of Web services.

The chapter concludes with a problem statement.

1.1 Embedded Devices and Systems

Increasingly many appliances, such as video recorders, mobile phones, cars, etc., have small computing units embedded within them. All these systems belong to a category of systems called embedded systems. The computing units embedded in these systems are called embedded devices. Embedded systems can be divided into different classes describing the characteristic properties of the environment they operate in. This section gives a definition of embedded devices, embedded systems and three different classes they can be divided into.

1.1.1 Definitions of Embedded Devices and Embedded Systems

Embedded devices are characterized by being hardware devices which

- include a programmable computer, and
have limited resources compared to a traditional personal computer, such as processing speed, memory size, communication bandwidth, power availability, etc.

This leads to the following definition of an embedded device:

\begin{definition}[Embedded device]
An embedded device is a hardware device containing a programmable computer, which has limited resources available when compared to a traditional personal computer.
\end{definition}

Examples of embedded devices include the computing hardware in PDAs, digital cameras, televisions, DVD players, and others.

Many different definitions of embedded systems exist, focusing on different characteristics. The definition of an embedded system used in this report is based on the following characteristics of such a system:

- It includes one or more embedded devices.
- It includes software interacting with an external environment, such as a human user, physical objects, sensors, etc.

From these properties the following general definition of an embedded system is introduced:

\begin{definition}[Embedded system]
An embedded system is any system consisting of one or more embedded devices, and contains software that interacts with an external environment, such as a human user, physical objects and sensors.
\end{definition}

An example of an embedded system is a digital camera. The system consists of a programmable computer with limited resources, i.e. an embedded device. This computer interacts with entities from the external environment, such as the motor controlling the optical unit, the LCD screen, the flash unit, etc.

### 1.1.2 Classes of Embedded Systems

In order to give a understanding of the type of embedded systems that are in focus in this report, three classes of embedded systems are defined according to the environment they operate in:

- Distributed embedded (DE) systems.
- Real-time embedded (RE) systems.
- Distributed real-time embedded (DRE) systems.

When describing these classes, the term “computing system” will be used often. This is used to describe any system containing a programmable computer, without stating anything about the resources available to the system. Such a system could be an embedded device, a personal computer, etc. Each class is described below.
1.1 Embedded Devices and Systems

Distributed Embedded Systems

Distributed embedded systems can be considered as embedded systems consisting of at least one embedded device cooperating with other computing systems in order to solve a common task. Therefore, the definition of such a system is similar to that of a distributed system, as defined by Coulouris et al. [6]:

**Definition 1.3 (Distributed embedded system)**

A distributed embedded system is an embedded system consisting of at least one embedded device communicating and coordinating actions with other computing systems only by passing messages, in order to solve a common well-defined task.

An example of a distributed embedded system is a sensor network [15]. A sensor network is a large ad hoc network of distributed sensors, i.e. embedded devices. These are equipped with transceivers allowing the sensors to coordinate among each other. Each sensor has limited sensing range, but combined in a sensor network the accumulated sensing range becomes larger. Sensors may be added or removed from the sensor network at any time.

Real-time Embedded Systems

Real-time embedded systems are embedded systems where the correctness of the system depends not only on the logical result of the computations, but also on the time at which results are produced [5]. This gives the following definition of an RE system:

**Definition 1.4 (Real-time embedded system)**

A real-time embedded system is an embedded system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment.

A typical role of these systems is monitoring and controlling physical devices [5]. An example of real-time embedded system is a DVD player, which has to maintain certain deadlines in order to produce a proper video and audio playback.

Distributed Real-time Embedded Systems

Distributed real-time embedded systems are systems consisting of at least one embedded device communicating with other computing systems, cooperating to solve a common task. As with distributed systems, communication is possible only by passing messages. Furthermore, the correctness of the system depends not only on the logical result of the computations, but also on the time at which results are produced. The following defines the notion of DRE systems:

**Definition 1.5 (Distributed real-time embedded system)**

A distributed real-time embedded system is an embedded system consisting of at least one embedded device communicating and coordinating actions with other computing systems only by passing messages, in order to solve a common well-defined task. The system is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment in order to provide a correct result.
An example of a distributed real-time embedded system is a modern car consisting of many embedded devices each monitoring inputs from the environment, and controlling different parts of the car. If stimuli arrives from the environment, e.g. if the driver presses the brake pedal, the ABS system has to monitor whether any of the wheels are blocking. It then has to react on this input within a given time interval in order to produce a correct result—in this case to ease on the brakes of the car.

An illustration of the relations between the three subclasses is shown in Figure 1.1. The figure shows the DE systems class and the RE systems class intersecting. This intersection illustrates the set of DE systems that has some sort of real-time requirements. A subset of this intersection is the class of DRE systems, which is the set of DE systems that has real-time requirements to the entire system including the communication. The remaining part of this intersection between DE and RE systems can be seen as the set of DE systems that has real-time requirements that do not relate specifically to being distributed, i.e. there are no real-time requirements to the communication. An example of a system belonging to the DE and the RE classes without being a DRE system is a device monitoring and controlling a physical device that offers a web administration interface. Real-time requirements are present for the monitoring and controlling process. However, there are no real-time requirements for the communication between the monitoring device and the client of the administration interface.

![Figure 1.1: Embedded systems and three subclasses thereof; distributed embedded system (DE), real-time embedded systems (RE), and distributed real-time systems (DRE). The hatched area indicates the set of embedded systems that are the focus of this report.](image)

The scope of this report relates to systems within the DE systems class. The focus area is the communication in such systems and how Web services can be applied as a mechanism for integrating embedded systems with other systems in a loosely-coupled fashion. Applications where real-time requirements to the communication are present are beyond the scope of this report. Therefore, the set of DE systems excluding DRE systems is the one in focus. This set is the hatched area in Figure 1.1, and will from this point on be referred to as DE systems for the sake of clarity.

### 1.2 Communication and Middleware

As distributed systems often operate in heterogeneous environments (such as diverse hardware platforms, operating systems, and programming languages), interoperability is complex. This complexity conflicts with the general desire for fast development time and low development cost. This imposes the need for standardized communication and reusable components in order to:
1.2 Communication and Middleware

- Shield software developers from low-level platform details, such as low-level network programming,
- minimize the software development cost by applying previous development expertise in reusable frameworks, and
- provide a set of high level network-oriented abstractions, that are close to the application requirements in order to simplify the development of distributed applications.

This is captured within the notion of middleware [23]. This section starts off by giving an overview of the structural changes of information systems throughout history that has led to the introduction of middleware. Next, a definition of middleware is given, and an abstraction of middleware, decomposing it into multiple layers, is described. The section ends with a description of the problems arising when integrating applications across companies and organizations using different middleware technologies.

1.2.1 Origin of Middleware

At an abstract level, an information system can be viewed as a three layer model consisting of a presentation layer, an application logic layer, and a resource management layer [1]. Any system that needs to communicate with external entities, such as humans or computers, need a way to represent the information that is communicated. The components in an information system responsible for these tasks form the presentation layer. Likewise, the application layer is the collection of components that in one way or another provide the services requested by the client of the information system, by processing available data. This data is provided by the resource management layer and resides in repositories such as databases, file systems, or some external system.

Originally, information systems consisted of a mainframe and a set of dumb terminals displaying the information as prepared by the mainframe. In this composition, also known as a one-tier architecture, all layers of the information system are located within one tier, as shown in Figure 1.2(a).

When local area networks appeared and PCs became powerful enough, it was possible to move some part of the functionality of information systems from mainframes to the client. Thereby information systems became two-tier client/server architectures instead of simple 1-tier architectures. In the two-tier architecture, the presentation layer is located at the client, while the application logic and the resource management layers are located at the server, as shown in Figure 1.2(b).

As the usage of two-tier systems grew, a need to provide extra scalability of information systems emerged, resulting in a move from two-tier systems to further distributed three-tier systems. With the three-tier architecture an additional middle tier containing the application logic layer was introduced. This layer has the responsibility of communicating with multiple distributed resources in the resource management layer. Due to the heterogeneity of these resources, abstractions and infrastructure that support the development of the application logic was introduced, known as middleware. An overview of the tree-tier architecture is shown in Figure 1.2(c). As information systems became more complex, the resource management tier in addition to simple resources such as databases, also started containing fully developed two-tier and three-tier systems, further increasing the need for middleware to provide the infrastructure needed to handle the complexity of communication. Such architectures are known as N-tier systems. With N-tier systems it becomes difficult to identify where one system ends and the next begin—the application logic layer in one system may be the client in another.
The evolution from one-tier to N-tier architectures was triggered by a need for larger flexibility, functionality, and possibility of distribution. The drawback of this evolution is that each tier introduces a decrease in performance due to the addition of another layer of abstraction. However, the architectural changes survived because the loss of performance was compensated for by the gain in flexibility and the possibility of distribution.[1]

1.2.2 Definition of Middleware

Yellin introduces two perspectives in which middleware can be viewed [42]. One view is that middleware is software that binds distributed components together, such as applications, directories, databases, etc. In this view, middleware has the role of allowing the different components to interact with one another, and is seen as software enabling horizontal composition of distributed components.

Another view is that middleware is an abstraction layer between applications and lower level services, such as operating systems, network protocol stacks, and hardware. In this view, middleware is seen as software enabling vertical composition of applications with lower level services.

These two views illustrates two important properties of middleware—enabling interaction between components, and simplifying programming models for application developers. This leads to the following definition of middleware [23]:
**Definition 1.6 (Middleware)**

Middleware is systems software that resides between the application and the underlying operating systems, network protocol stacks, and hardware. Its primary role is to

- functionally bridge the gap between application programs and the lower-level hardware and software infrastructure, in order to coordinate how parts of applications are connected and how they inter-operate, and to
- enable and simplify the integration of components developed by multiple technology suppliers.

### 1.2.3 Structure of Middleware

As described by Schantz and Schmidt, middleware can be decomposed into multiple layers of abstraction [23]. Figure 1.3 illustrates these different layers of middleware and their relationship to platform and application layers. Each of the middleware layers are described below.

![Figure 1.3: The layers of middleware and their relationship to platform and application layers.](image)

**Host Infrastructure Middleware**

Host infrastructure middleware has the role of encapsulating and enhancing native operating system communication and concurrency mechanisms, in order to create reusable network programming components. These components provide abstractions of lower level components, such as operating systems programming APIs. Some examples of host infrastructure middleware are:

- The Common Language Runtime (CLR) [14], which is the foundation on which Microsoft’s .NET platform is built. The CLR provides an execution environment that simplifies software development by providing elements such as automatic memory management, cross-language integration, and a security system.
- The ADAPTIVE Communication Environment (ACE) [7], which is a portable toolkit that encapsulates OS network programming capabilities, such as connection estab-
lishment, interprocess communication, marshalling/unmarshalling, concurrency, and synchronization.

**Distribution Middleware**

Distribution middleware introduces higher-level distributed programming models built upon the components of host infrastructure middleware. Distribution middleware enables programming of distributed applications equivalent to non-distributed applications programming. This enables invocation of operations on objects (remote method invocation) in object oriented distribution middleware without specific knowledge of their location, programming language, operating system platform, communication protocols, and hardware. Similarly, in non-object oriented distribution middleware, procedures implemented on different hosts can be called (remote procedure calls). Examples of distribution middleware include:

- The Object Management Group’s (OMG) Common Object Request Broker Architecture (CORBA) [19], which is an open standard that allows objects to inter-operate across networks, disregarding the programming language in which they were written and the platform on which they are deployed. Several different CORBA specifications exists each having different focus areas [24]:
  - *Minimum CORBA*, which removes nonessential features from the CORBA specification in order to reduce the memory footprint rendering it usable in resource-constrained systems.
  - *Real-time CORBA* includes features that allow applications to reserve and manage network, CPU, and memory resources in a predictable manner.
  - *Fault-tolerant CORBA* introduces redundancy to support replication, fault detection, and failure recovery.

- Sun Microsystems Java Remote Method Invocation (RMI) [27], which enables the creation of distributed Java applications. It enables invocation of remote Java objects without concern on which host or Java virtual machine they reside. However, Java RMI is specific to the Java programming language, and few implementations that supports other programming languages are available. A special version of Java known as the Java 2 Platform Micro Edition (J2ME) that is optimized for devices with limited resources such as embedded systems is also available.

- SOAP [33, 34], defined by the World Wide Web Consortium (W3C), provides a mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML for encoding of data and typically using HTTP as the underlying communication protocol.

**Common Middleware Services**

Common middleware services extends distribution middleware by introducing high level domain independent services. As the name implies, this type of middleware allows application developers to avoid implementing common elements in applications, such as services managing transactions, security, database connection pooling, and others. Examples of common middleware services are:
• The OMG’s CORBA Common Object Services (CORBA services) [18], which provides numerous domain independent services that can be used by applications, such as logging, persistence, security, fault tolerance, and others.

• Web services [35], which provides a mechanism for exposing services offered by applications using standard Internet technologies. The Web service architecture defines common services such as services to publish the interface of Web services, and discovery services.

### Domain-specific Middleware Services

Domain-specific middleware services are middleware services that address the requirements of a specific domain, such as telecommunications, e-commerce, health care, and others.

An example of a domain-specific middleware service is syngo® [25] developed by the Siemens Medical Engineering Group. The syngo® middleware is both an integrated collection of domain-specific services as well as an open and dynamically extensible application server platform for medical imaging tasks and applications. The syngo® middleware services allow health care facilities to integrate diagnostic and other services via a black-box application template framework.

#### 1.2.4 Application Integration with Middleware

Middleware has many strengths when building distributed applications where communication is complex, due to the heterogeneous environment the application operates in. However, when a need arises for integrating applications developed by different organizations, challenges arise [1].

Figure 1.4 illustrates a case where application integration across different companies is useful. A company is acting as a customer, ordering some goods from a supplier. The supplier may deliver the goods directly if they are in stock, or may request a resupply from the distributor if the goods are not in stock. It would be advantageous for all participating parties, if the whole business process was automated. This is rarely the case when the business process occurs across companies, as shown in Figure 1.4. In most cases the integration is performed manually by employees as illustrated in Figure 1.4.

![Figure 1.4](image)

**Figure 1.4:** There is a desire to automate application integration across companies, but currently this is still often done by manual interaction. At first an employee at the customer places an order at the web server of the supplier. An employee at the supplier then check the order then arrange for the goods to ship to the customer if in stock. Otherwise, a resupply order is issued to the web server at the distributor. An employee at the distributor make arrangements so that the goods will be shipped to supplier. When the goods arrive, the supplier arrange for the goods to be delivered.

If applications developed by different organizations are designed to communicate in
order to form a single greater information system with closely-coupled interactions, the organizations can agree on a middleware technology to provide the needed infrastructure to communicate between the different systems. However, if there is a desire by an organization to offer loosely-coupled services to information systems developed by other organizations, it is not as simple to choose a middleware technology. The organization providing the service faces the challenge of choosing the middleware technology that as many clients as possible will support. Similarly, the provider of a service may also be a client of other services from other information systems—such as the supplier company from the example in Figure 1.4. This company could have a desire to offer a service to order goods, while itself is a client of some services offered by the distributor company.

If each company in the above example internally uses different middleware technologies to develop their applications, challenges arise. As shown in Figure 1.5, a company would then have to support the middleware technology used by each company being communicated with—e.g. the supplier company offers a service using middleware technology “a” while relying on services offered by the distributor company using middleware technology “b”. However, as companies can interact with many different partners and each partner could require the use of another middleware technology, a scenario arises in which a company will have to support a range of different middleware technologies.

![Figure 1.5: Application integration across organizations using different middleware technologies.](image)

Basically, the conventional idea for middleware is to reside in between applications. But in cross-organizational interactions there is no obvious place to put the middleware. In order to allow for such application integration, the W3C has introduced Web services. This can be considered as common middleware services that provide functionality allowing applications to interact in a distributed, decentralized environment. Web services is inspired by the well-established architecture of the World Wide Web, and are layered upon technologies used on the World Wide Web, such as HTTP and XML. This report will concern the application of Web services in distributed embedded systems. This choice is motivated by the fact, that Web services is an emerging technology for application integration in loosely-coupled environments. The increasing connectivity of embedded devices makes it interesting to investigate the applicability of Web service technologies for integrating these devices with other computing systems. The next section gives a description of the Web service middleware technology.

### 1.3 Web Services

The W3C Web Services Architecture Working Group defines a Web service as [35]:
1.3 Web Services

**Definition 1.7 (Web service)**

A Web service is a software system designed to support inter-operable 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 SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

This section gives an overview of the Web service concept and introduces the Web service architecture along with the core technologies applied in this architecture. Furthermore, an example of a Web service is given.

### 1.3.1 The Web Services Concept

In order to understand how Web services strive to meet the challenges presented in Section 1.2.4, three main aspects of Web services are important [1]. These are presented below:

**Service Orientation**

The first aspect of Web services is that they work on the assumption that functionality made available by an organization always will be exposed as a service, having a stable published interface that can be invoked by client programs.

Web services is designed to be invokable across organizations, and as a consequence Web services assume that services are loosely-coupled, since they are expected to be defined, developed and managed by different organizations. Different services are thought of as being autonomous and independent.

**Decentralized Middleware Protocols**

The second aspect of Web services is the redesign of the middleware protocols to allow cross organizational communication. Therefore, Web services has been designed to work in a decentralized manner.

**Standardization**

The final aspect of Web services is standardization in order to allow for invoking services implemented on heterogeneous platforms. The design of Web services is inspired of the design of the Web, which has been characterized by a high degree of standardization, which allows it to function without centralized coordination. Therefore, Web services is based on existing mature technologies used on the Web, such as HTTP and XML.

### 1.3.2 Architecture

A Web service architecture typically consists of the entities illustrated in Figure 1.6 [21]. These interact with each other in the following way:

- **Service providers** provide Web service applications and publish these by registering their interfaces with **service brokers**.
- **Service brokers** maintain a registry of published services and provide an interface for service providers to publish services. Similarly, they provide an interface for **service requesters** to search for services in the registry.

- **Service requesters** initially find the desired service by performing a lookup in the service broker’s registry and then access the service itself directly from the service provider. Alternatively, if the service is already known, it is accessed directly without using service brokers.

![Figure 1.6](image.png)

**Figure 1.6**: A Web service architecture with core technologies illustrated. SOAP and WSDL are standard technologies specified by W3C, while UDDI is an example of a discovery service protocol.

Other components can also be added to the Web service architecture, such as transaction control [3] and security [11]. However, such components have yet to be standardized.

### 1.3.3 Technologies

Web services is based on two essential technologies [1]: SOAP and Web Service Description Language (WSDL) [36, 37, 38]. Additionally, a discovery service, such as Universal Description, Discovery, and Integration (UDDI) [29], is typically added to the Web service architecture. Each of these technologies are described below:

- **SOAP** is a protocol based on XML used to exchange messages between peers. SOAP is used to send requests to Web services, as well as receiving replies to these requests. SOAP is comparable to the Inter-ORB Protocol used by CORBA.

- **WSDL** is a language based on XML used to describe the interface of a given Web service, including elements such as data type definitions, the operations supported by the service, message formats and protocol bindings. This is a central part of the Web service architecture, since it describes the protocol for communicating with a Web service. WSDL is comparable to the Interface Definition Language (IDL) used by CORBA.

- **UDDI** is a specification that provides a mechanism to publish and search for Web services, thereby allowing service providers to register their Web services. This is similar to the Trading service in CORBA. Interaction with an UDDI registry is done via SOAP messages.

The placement of these technologies in the Web service architecture is illustrated in Figure 1.6.
1.3 Web Services

1.3.4 An Example of a Web Service

An example of a simple Web service is the BabelFish Web service, a multilingual translation service, offered by Altavista [41]. A client submits a string that is to be translated, along with information about the language being translated from and the language being translated to. The service replies with the translated text.

In order to build a client the WSDL specification of the service is needed. This specification describes how to generate a request to the service and to understand the replies received from the Web service. This could be retrieved from an UDDI registry or another resource, such as the web site of the Web service provider. A part of the WSDL specification of the BabelFish Web service can be found in Example 1.1.

A key point specified in the BabelFish service WSDL is that the service takes a message BabelFishRequest with the parameters translationmode and sourcedata as input (lines 2-5). A client sending this message indicates that it has some text it wants translated. The parameter translationmode represents the language being translated from and the language being translated to (e.g. the value "en_de" indicates a translation from English to German). The parameter sourcedata represents the text string that the client wants to translate.

Furthermore, the WSDL specifies that the service outputs a message by the name BabelFishResponse with the parameter return (lines 6-8). This message will contain the translation of the input string. The WSDL also contains other information, which is left out in this example for the sake of clarity.

Example 1.1: Part of the WSDL specification of the BabelFish Web service.

Example 1.2: A SOAP request for accessing the BabelFish Web service.
Example 1.2 illustrates an example of a SOAP request to the BabelFish Web service, where the text “SOAP is cool” (line 11) is requested to be translated from English to German (line 10). The request contains additional information required by the SOAP specification. This will be explained in detail in Chapter 2. Example 1.3 illustrates the corresponding SOAP response, where the translated text, “SEIFE ist kühl”, is returned (line 9).

```xml
<soap:Envelope
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:soapenc="http://www.w3.org/2003/05/soap-encoding"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body>
    <namesp1:BabelFishResponse xmlns:namesp1="urn:xmethodsBabelFish">
      <return xsi:type="string">SEIFE ist kühl</return>
    </namesp1:BabelFishResponse>
  </soap:Body>
</soap:Envelope>
```

Example 1.3: A SOAP response from the BabelFish Web service.

1.4 Summary

This chapter has introduced the concept of middleware and some of the challenges associated with integration of different applications implemented by different organizations, and has introduced Web services as a middleware technology that attempts to meet these challenges. As new applications of embedded systems arise, similar challenges are faced. An example is a PDA equipped with GPS, relying on an online service that provides access to geographical map data. The PDA combines these data to build an application that allows the user of the PDA to retrieve a map of the area he/she is currently located in. The PDA could also rely on services offered by other systems, and in order to enable communication with all these system, the PDA would have to support multiple middleware technologies. This is often not practical in embedded systems, due to resource constraints.

The next section gives the problem statement of this report.

1.5 Problem Statement

This report investigates how Web services can be applied in distributed embedded systems. The aim is to investigate whether Web services is a well-suited and applicable technology to integrate embedded systems with other computing systems in a decentralized environment. In order to establish this, a framework for creating Web services on distributed embedded systems is designed. To determine the resources required for implementing Web services on distributed embedded systems, the main focus is put on designing the architecture of the code generated by that framework. A prototype of this code is implemented in order to establish a platform for performance tests.

In Chapter 2 extensive analysis of the key technologies in Web services are performed. In Chapter 3 a design of the framework is carried out, and optimizations are identified which allows for efficient management of the limited resources in embedded systems. Furthermore, a design for the architecture of the Web service middleware code generated by
the framework is developed. In Chapter 4 a prototype implementation of the Web service is described, and in Chapter 5 performance and profiling tests are performed on this implementation in order to establish an overview of the resources requirements of Web services on embedded systems. Finally, Chapter 6 gives a conclusion on the report.
In order to design a framework for developing Web services on distributed embedded systems, an understanding of the technologies and terminology involved in such a design is necessary. This chapter provides a technical analysis of the Web service technologies in order to provide a basis for the design of such a framework.

Initially, this chapter describes the structure and key terms of XML. Furthermore, the XML related technologies XML Namespaces and XML Schemas are described, including their application in the SOAP protocol. Next, a description of the SOAP protocol is given, including message constructs, processing model, and bindings to transport protocols. This is followed by a description of WSDL, the language used to describe the services offered by a Web service. Finally, XML parsing techniques are described in order to give a foundation for choosing a suitable parsing technique for an embedded system.

## 2.1 Extensible Markup Language

XML is a markup language specified by W3C in the W3C XML 1.0 Recommendation [31]. Didier Martin et al. defines a markup language as [2]:

\begin{definition}[Markup language]
“Markup” is a method of conveying meta-data (that is, information about a dataset). Markup languages use string literals, or “tags”, to delimit and describe this data.
\end{definition}

Some of the design goals of XML are [31]:

1. XML shall support a wide variety of applications.
2. It shall be easy to write programs which process XML documents.
3. XML documents should be human-legible and reasonably clear.
4. XML documents shall be easy to create.

XML addresses these design goals by:
1. Separating content from presentation. XML documents can describe any dataset, but it is up to the XML designer to define the semantics of the data. This helps XML to support a wide variety of applications.

2. Using delimiters to distinguish structure and properties of data from the data itself, easing the writing of XML processing programs.

3. Encoding XML documents using text. This aids XML in being readable and clear. Unicode Translation Format (UTF) character encodings UTF-8 and UTF-16 are recommended by the W3C.

4. Dividing the requirements of XML documents into two parts, a mandatory and an optional, and keeping the mandatory part minimal, eases XML document creation.

2.1.1 Markup and Character Data

An XML document consists of markup and character data. An example of an XML document is shown in Example 2.1. As XML provides no means to define semantics, these must be determined by the application. An example of such semantics could be that the document in the example describes a person, including the name and age of that person, and a reference to that person’s picture.

```
<?xml version="1.0" ?>
<person>
  <name>
    <first>Linus</first>
    <last>Torvalds</last>
  </name>
  <picture href="http://catb.org/~esr/faqs/linus/small-linus.jpg" />
  <age>34</age>
</person>
```

Example 2.1: A simple example of an XML document.

Markup specifies structure of data. Markup is text embedded within a pair of “<” and “>” delimiters. A single instance of markup is known as a tag, e.g. `<first>` in Example 2.1 (line 4) is a tag. The text not enclosed within these delimiters is known as character data.

Markup is used to group character data into logical units known as elements. Three types of tags are used to define elements. These tags are called start-tags, end-tags, and empty-element-tags. An example of an XML element delimited by a start-tag and an end-tag is illustrated in line 4 of Example 2.1. The start-tag `<first>` starts the first element, while the end-tag `</first>` ends it. The text in between these tags is character data, and is called the content of the element.

Elements with no content are said to be empty. They can either be represented by a start-tag followed immediately by an end-tag, or by an empty-element-tag. An empty-element tag is a single tag delimited by “<” and “/>”. An example of an empty-element tag is the `picture` tag on line 7 of Example 2.1. A set of attributes can be bound to an element, each having a name and a value. These are specified in either a start-tag or an empty-element-tag. On line 7 of Example 2.1 the attribute with the name `href` and its value (a URL) is specified. The value of an attribute is also considered character data. Figure 2.1 shows an overview of the XML terminology of the parts of an element.
2.2 XML Namespaces

When using different elements with the same name in a XML document, name clashes occur. Example 2.1 shows an XML document containing a representation of a person. One could imagine that a banking application would define a representation of a person in a different way, as illustrated in Example 2.3.

The XML specification does not define a fixed set of elements, but instead allows documents to use custom names. This makes XML extensible in the sense that XML documents can encapsulate data structures not previously defined.

2.1.2 Well-Formedness and Validity

In order for a data object to be an XML document, it must be well-formed. To be well-formed, an XML document needs to be accepted by the grammar defined in the XML specification. Describing this grammar is beyond the scope of this report. The important thing to note is that well-formedness is only concerned with conformance to the syntactic rules of XML, and not the specific elements or their structure.

Furthermore, if an XML document keeps within some further constraints, it is said to be valid. XML provides a mechanism to specify the structure and permissible values of XML documents, namely the Document Type Definition (DTD).

DTDs uses a formal grammar to specify the elements that can be present in an XML document. Example 2.2 shows a DTD specifying the permissible structure (for validity) of a document that contains a person element as the root element. In order to be valid, an XML document needs to keep within the constraints specified in the DTD of the XML document. The XML Document in Example 2.1 would validate against the DTD illustrated before. This DTD specifies that the document should begin with a person element as the root element. Furthermore, the person element has a name element, a picture element, and an age element as sub elements (line 3). The name element consists of the first and last elements which holds character data (indicated by #PCDATA in the DTD in lines 4-6). The picture element is an empty tag (line 7), and the age element holds character data (line 8). Finally, the DTD specifies that it is required that the picture element has an href attribute. DTDs allow specifying whether an attribute is optional or required.

As is described in Section 2.3, DTDs have limitations that limits their use in modern XML applications.

2.2 XML Namespaces

Figure 2.1: An overview of the XML terminology of the different parts of an XML element, using the XML element <person cpr="010245-6789">Linus Torvalds</person> as an example.
Example 2.2: An example of a Document Type Definition for a person.

Example 2.3: An XML data structure describing a person in a banking application.

When using both of these two person data structures in the same document name clashes occur. When encountering a person element, the reader of the XML document is unable to determine whether it is the person element from Example 2.1 or from Example 2.3. This is why the W3C has introduced XML namespaces [32]. The W3C gives the following definition of an XML namespace:

**Definition 2.2 (XML namespace)**

An XML namespace is a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names.

As the definition states, a namespace in XML is defined by a URI. Example 2.4 illustrates the declaration of three different namespaces (lines 2-4 in the example).

Example 2.4: Some example namespace declarations.
Generally, a namespace is defined by adding an `xmlns` (short for XML Name Space) attribute to any element in the document. The value of the `xmlns` attribute has to be a valid URI and is used to uniquely identify the namespace.

In order to be able to refer to a namespace, a prefix can be assigned to it. This is done by adding a `xmlns:prefix` attribute to an element where `prefix` can be any name. In Example 2.4 on line 3 the prefix of the defined namespace is `bankperson`. Furthermore on line 12 in this example a referral is made to this namespace in the `bankperson:person` element.

### 2.2.1 Scope

As in programming languages, XML namespace declarations are limited by a scope. This is an important concept because namespaces can be declared at any point in the document (as illustrated in Example 2.4 line 4). In order to avoid adding prefixes to all elements in a document, the concept of name scope is introduced. Whenever a namespace is declared for an element, this namespace also applies for the attributes of the element and for its child elements and their attributes. However, other namespaces can be introduced in the child elements, overriding the default namespace (as in line 4). Two types of scopes exist:

- **Default scope** is declared by omitting the prefix declaration of a namespace declaration in an element. In Example 2.4 on line 2 a default namespace is declared for the `personlist` element. All the child elements of the `personlist` element will be in that namespace unless overwritten by another namespace.

- **Qualified scope** is declared by prepending the prefix of a already declared namespace to an element, as illustrated in line 12 of Example 2.4. All the children of this element will belong to this namespace (e.g. the `id` element on line 13 is in this scope).

### 2.3 XML Schemas

Problems arise when attempting to use DTDs together with XML namespaces, because DTDs has no knowledge of the concept of namespaces.

Referring to Example 2.4, two different `person` elements are used in the same XML document. In order to validate the document in Example 2.4 two DTDs are needed in order to define the structure of each type of person. This is not possible because DTDs do not permit defining the structure of a single element, and because it is not possible to combine DTDs. Additionally, other issues with DTDs have been pointed out, such as they are written in a different format than XML and there is limited support for data-types [2].

This is why the W3C introduced XML schemas [40]. XML schemas introduces an XML syntax for defining structure of XML elements, have better support for data types, and allow for greater extensibility than DTDs.

Example 2.5 shows the definition of a simple XML schema. This schema defines a `person` element (line 4), consisting of the sub elements `id`, `accountnumber`, and `balance` (lines 5-11). They are all declared as being of the type `int` (integer). Example 2.4 (line 3) shows an XML document referring to the XML schema defined in Example 2.5.

When declaring a namespace, it is identified by a URI pointing to an XML schema, indicating that this schema defines the permissible structure allowed by the elements in this
Example 2.5: Example of XML schema for a person element in a banking application.

namespace. It is also possible to use more than one schema in the same XML Document (as illustrated in Example 2.4).

A further description of XML schemas is beyond the scope of this report. A short introduction has been given to this concept in order to give an understanding of the difference between DTDs and XML-schemas, and in order to be able to understand their use in SOAP.

2.4 SOAP

This section gives an overview of the SOAP protocol version 1.2 specified by the W3C [33, 34]. SOAP is fundamentally a stateless, one-way message exchange paradigm which can be used for exchanging structured and typed information between peers in a decentralized, distributed environment.

Giving a detailed description of SOAP is beyond the scope of this report. Only essential elements and concepts are introduced in order to give an understanding of the protocol. The flow of this description is inspired by the flow of the SOAP 1.2 specification.

Initially, the terminology used in the specification of SOAP is introduced. Next, the structure of SOAP messages is explained. The rules for how SOAP messages must be processed is then described. Finally, it is described how SOAP messages can be conveyed using the HTTP protocol.

2.4.1 Protocol Concepts

The SOAP specification introduces some general terms which are described below:

**SOAP node:** SOAP provides a distributed processing model that assumes a SOAP message originates at an initial SOAP sender and is sent to an ultimate SOAP receiver via zero or more SOAP intermediaries named SOAP nodes. A SOAP node is the embodiment of the necessary processing logic to transmit, receive, process and relay a SOAP message according to the set of conventions defined by the SOAP specification. A SOAP node can be the initial SOAP sender, an ultimate SOAP receiver, or a SOAP intermediary.

**SOAP role:** In processing a SOAP message, a SOAP node is said to act in one or more SOAP roles. The term “SOAP role” is used to refer to a SOAP node’s expected function in processing a message. Some standard roles are defined by the SOAP specification, such as the “ultimateReceiver” role, stating that this node is the final destination of a SOAP message. Other roles can be defined by an application, e.g. a “logger” role. This role could
be taken by an intermediate SOAP node logging all the SOAP messages it receives and then relaying them to another SOAP node.

**SOAP binding:** The term “SOAP binding” is used to describe the formal set of rules for carrying a SOAP message using an underlying protocol, such as HTTP.

**SOAP message exchange pattern:** A message exchange pattern (MEP) is a template for the exchange of SOAP messages between SOAP nodes enabled by one or more underlying SOAP protocol bindings. An example of a MEP enabled by the HTTP binding is the request-response pattern.

**SOAP application:** The term “SOAP application” is used to refer to an application that produces, consumes, or otherwise acts upon SOAP messages in a manner conforming to the SOAP specification.

### 2.4.2 Message Sender and Receiver Concepts

SOAP introduces some terms used when referring to sending and receiving SOAP messages. These are described below.

**SOAP sender:** The term “SOAP sender” is used to refer to a SOAP node that transmits a SOAP message.

**SOAP receiver:** The term “SOAP receiver” is used to refer to a SOAP node that receives a SOAP message.

**SOAP message path:** A SOAP message path refers to the set of SOAP nodes through which a single SOAP message passes. This includes the initial SOAP sender, zero or more SOAP intermediaries, and an ultimate SOAP receiver. These terms are described below. Figure 2.2 illustrates a SOAP message path with two SOAP intermediaries.

![Figure 2.2: A SOAP message path with two intermediary nodes.](image)

**Initial SOAP sender:** This term refers to the SOAP sender that creates a SOAP message at the starting point of a SOAP message path.

**SOAP intermediary:** A SOAP intermediary is both a SOAP receiver and a SOAP sender. It processes the SOAP header blocks of a SOAP message targeted at it, and forwards the message to another SOAP receiver. SOAP header blocks will be explained in Section 2.4.3.
Ultimate SOAP receiver: This term refers to a SOAP receiver that is the final destination of a SOAP message. It is responsible for processing the contents of the SOAP body and any SOAP header blocks targeted at it. SOAP bodies will be explained in Section 2.4.3. In some circumstances, a SOAP message might not reach an ultimate SOAP receiver, for example because of a problem at a SOAP intermediary. An ultimate SOAP receiver cannot be a SOAP intermediary for the same SOAP message.

2.4.3 Message Construct

A SOAP message is application data encoded in XML. This section gives an overview of how SOAP messages are constructed. Figure 2.3 shows the general structure of SOAP messages and the XML elements they consist of. These are described below:

![Figure 2.3: The main XML elements of a SOAP message.](image)

**SOAP envelope:** The SOAP envelope is the outermost element of a SOAP message, which contains the other parts of the SOAP message. Also, the SOAP envelope specifies properties of the message, such as its encoding. As Figure 2.3 shows, an envelope consists of an optional SOAP header and a mandatory SOAP body. These elements are described below. Throughout this section, references to Example 2.6 is made. This example shows how a SOAP message can be constructed, and illustrates some of the elements of a SOAP message that are described. The SOAP message represents a travel plan for a passenger traveling from Copenhagen to Aalborg and back. Line 2 of the example contains the start-tag `soap:Envelope` marking the start of the SOAP envelope element. This element is declared to be part of the namespace with the prefix `soap` identified by the value of the `xmlns:soap` attribute. This value is an URL pointing to an XML schema defining the permissible structure of a SOAP envelope. All SOAP 1.2 envelopes must be part of this namespace.

**SOAP header:** A SOAP header is an mechanism that provides a way to pass information in SOAP messages that is not necessarily meant for the ultimate receiver. An example use of a header is when a SOAP node sends a message with a header that specifies that this message is to be logged in a database. Lines 4-10 of the example illustrates a header.
Example 2.6: An example of a SOAP message.
stating that the message must be logged, indicated by the \texttt{r:logmessage} element. This element is defined to belong to another namespace, in order to avoid name clashes with the \texttt{soap} namespace. A SOAP intermediary in the message path with the role of logging messages will then read this header and find that it is necessary to log it. However, SOAP does not specify how the content of the header element should be interpreted. This is up to the application to decide, thereby allowing a SOAP message to be extended in an application-specific manner. Headers may be inspected, inserted, deleted, or forwarded by any SOAP intermediary encountered along a SOAP message path.

The immediate child elements of the header element are called header blocks, and represent a logical grouping of header data.

**SOAP header block:** A SOAP header block represents a single piece of information in the header of a SOAP message. A SOAP header block may have a \texttt{role} attribute, indicating which SOAP nodes the header block is targeted at, e.g. in the case with a SOAP intermediary logging some messages, the role specified is “loggerNode” (line 6 of the example).

Furthermore, each header block can contain a \texttt{mustUnderstand} attribute. This indicates whether it is a requirement that the SOAP node receiving the message should understand this header block, e.g. in line 7 of the example this attribute is set to “false”, since it is not expected that all SOAP intermediaries have to understand this header field. Only an intermediary that assumes the “loggerNode” role must understand it.

Finally, a header block can contain a \texttt{relay} attribute. This indicates whether a header block should be relayed, if not processed by a SOAP intermediary. Relaying is explained in Section 2.4.4. The \texttt{relay} attribute is often used in conjunction with the \texttt{mustUnderstand} attribute. For instance, in the logger example the \texttt{mustUnderstand} attribute is set to “false” and the \texttt{relay} attribute is set to “true”, indicating that this header block must be relayed along the message path until a SOAP intermediary, understanding this header block, processes it (lines 7-8).

**SOAP body:** The SOAP body is a mandatory element of a SOAP message, and it is targeted at the ultimate SOAP receiver. SOAP only pre-defines one sub-element of the SOAP body, the SOAP fault element, which is described below. Any other sub-elements can be added to the SOAP body as the application desires. Therefore, the SOAP body is used as the main mechanism for conveying data relevant for the ultimate receiver. The SOAP specification does not state anything about how the content of the body should be structured. It is up to the application to define the structure and semantics of the data. An example of the content of a SOAP body is illustrated in lines 13-27. These elements are defined to belong to another namespace than the \texttt{soap} namespace, in order to avoid name clashes. The semantics of this content is that it describes the travel plan of a passenger traveling from Copenhagen to Aalborg.

**SOAP fault:** SOAP provides a mechanism for handling situations when faults arise in the processing of a message, namely the \texttt{fault} element contained within the \texttt{body} element of a SOAP message. Each message can only contain one \texttt{fault} element. The specification describes two mandatory sub-elements of the \texttt{fault} element: The \texttt{Code} and the \texttt{Reason} sub-elements. The \texttt{Code} element is itself made up of a mandatory \texttt{Value} sub-element and any number of optional \texttt{Subcode} sub-elements. The content of the \texttt{Value} sub-element is restricted to a small set of strings specified by SOAP. An example is the string “Sender”, indicating that the message sent by a SOAP node was incorrectly formed.
or did not contain the appropriate information in order to succeed. Example 2.7 shows a SOAP message containing a fault element, with the content of the Value element set to “Sender” (line 6). The application can assign any value of the Subcode sub-elements. The Reason sub-element is not meant for algorithmic processing, but rather to specify the reason for this fault occurring in a human readable form. Example 2.7 illustrates the usage of this element (lines 11-15).

Example 2.7: An example of SOAP message containing a fault element.

```xml
<?xml version="1.0" ?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
    <soap:Fault>
      <soap:Code>
        <soap:Value>Sender</soap:Value>
        <soap:Subcode>
          <soap:Value>Missing argument</soap:Value>
        </soap:Subcode>
      </soap:Code>
      <soap:Reason>
        <soap:Text xml:lang="en-US">
          The argument "departure" was not found while processing the message.</soap:Text>
        </soap:Reason>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

2.4.4 SOAP Processing Model

This section describes the processing model of SOAP. The SOAP processing model describes the actions taken by a SOAP node upon receiving a SOAP message. The processing model applies to a single message only, in isolation from any other SOAP messages. The SOAP processing model itself does not maintain any state or perform any coordination between messages.

The list below describes the steps a SOAP node must take when processing a SOAP message:

1. Determine the set of roles in which the node is to act. It is up to the application to decide which roles a certain node acts in, with the exception of some roles predefined by SOAP, e.g. SOAP defines the “next” role which any SOAP node must act in.

2. Identify all header blocks targeted at the node that has the mustUnderstand attribute set to “true”.

3. If one or more of the SOAP header blocks identified in step 2 are not understood by the node, then a single SOAP fault must be generated, with the value of sub-element Code set to “MustUnderstand”. This indicates that an error occurred due to the node not understanding the header block. If such a fault is generated, any further processing must not be done.

4. Process all SOAP header blocks targeted at the node with the mustUnderstand attribute set to “true” and, if the node acts in the role of the ultimate SOAP receiver,
also process the SOAP body. A SOAP node may also choose to process SOAP header blocks targeted at it which does not have the mustUnderstand attribute set to “true”, however the node is not obliged to do so. It is up to the application to decide how the body and header block is to be processed.

5. If acting as a SOAP intermediary, relay the message to the next node in the message path as described below.

**Acting as a SOAP Intermediary**

As mentioned previously, SOAP assumes that a message originates at an initial SOAP sender and is sent to an ultimate SOAP receiver through zero or more SOAP intermediaries. An intermediary has the responsibility of processing a SOAP message and then forwarding it to the next node in the message path. Depending on the roles an intermediary acts in, the header blocks of a SOAP message can be altered. However, the body of the message is always kept intact. The steps that such an intermediary take are:

1. Determine the set of roles, in which the intermediary is to act in.

2. Process each header block targeted at the intermediary, as described above, and remove these header blocks. The intermediary can choose to re-insert the header blocks, but this is up to the application.

3. Remove each header block that was chosen not to be processed in the previous step (chosen from the set of header blocks with the “mustUnderstand” attribute not set to “true”). However, if such a header block has the relay attribute set to “true”, it must not be removed.

4. Do not alter any header blocks not targeted at the intermediary.

5. Forward the SOAP message to the next node in the message path.

**2.4.5 SOAP Protocol Binding**

This section describes how SOAP can be bound to the HTTP protocol. The SOAP specification is silent upon which underlying protocol should be used to convey SOAP messages. It only defines a framework to specify a binding to a certain protocol. However, the most commonly used protocol is HTTP. A short introduction to the two most recent versions of the HTTP protocol is given, and the binding between SOAP and HTTP is described.

**HTTP 1.0**

HTTP 1.0 [17] defines a protocol for transferring data of varying types in a client-server environment using a request-response MEP. A client initiates the communication by sending an HTTP request, and the server replies with an HTTP response.

Multiple types of requests are possible, such as POST and GET. Only these two types of requests are described, since they are the only ones used by SOAP. POST requests are used when data needs to be transferred from the client to the server. An example of a HTTP POST request is listed in Example 2.8. This request sends a SOAP request to the server identified by the URL “http://example.org/bankservice”. An HTTP POST request consists of three parts in the following order:
1. This part contains the request method (POST), along with the requested resource, and the version of the HTTP protocol being used (line 1 in the example).

2. This part contains a number of HTTP headers. The headers describe properties about the request, such as the content type and content length of the message body (lines 2-3). In this example the content type is "application/soap+xml", indicating that the body contains a SOAP message. The DNS name used when looking up the server address may also be sent using the Host header (line 4).

3. This part contains the HTTP body, which is the data being transmitted along with the request to the resource. The body begins after the first blank line (lines 6-9). In this example the body is a SOAP message.

Example 2.8: Example of an HTTP 1.0 POST request. The SOAP message encapsulated in the body has been shortened for clarity.

An HTTP response is similar to a POST request, except that the first part is replaced by a status code and description indicating the outcome of the request. An example of a HTTP response is listed in Example 2.9.

Example 2.9: Example of an HTTP 1.0 response. The SOAP message encapsulated in the body has been shortened for clarity.

The GET method is an another request method which does not include a body part. Its intended application is document retrieval, meaning that it should not modify the state of the targeted resource. Example 2.10 shows an HTTP GET request, requesting the resource found at “http://example.org/getClockTicks”. Responses to GET requests are similar to those of POST requests.

Example 2.10: Example of an HTTP 1.0 GET request.
HTTP 1.0 uses TCP as the underlying transport protocol. The client initiates the TCP connection and sends the HTTP request. The server then uses the same TCP connection to send the HTTP response. When the server has sent the response the TCP connection is terminated. HTTP benefit from the fact that TCP is a reliable transport protocol, meaning that HTTP does not need to handle lost, duplicate, or out-of-order data.

HTTP 1.1

HTTP 1.1 [16] extends HTTP 1.0 with several features that, amongst other things, are meant to increase performance. One central feature that was added in HTTP 1.1 is persistent connections. It is possible to keep a connection open after a request has been sent and a response has been received. A client is then able to send another request using the same TCP connection. When a client needs to send several requests to a single server, this can increase performance by avoiding the overhead of multiple TCP handshakes.

HTTP 1.1 also supports pipelining, meaning that a client can send several requests on the same TCP connection without waiting for each response to arrive.

The message format of HTTP 1.1 requests and responses are similar to those of HTTP 1.0, except from additional headers used to control the new features of HTTP 1.1.

SOAP over HTTP

The SOAP HTTP binding supports two MEPs: The request-response and the response message exchange patterns. These are described below.

The response MEP is used for requesting a resource at a SOAP node without affecting the state of that resource. This MEP is realized in the SOAP HTTP binding by using the GET method. The resource being requested at the SOAP node is identified by a URL.

The request-response MEP is used for transmitting a SOAP message to a SOAP node and receiving another SOAP message as a response. This MEP is realized in the SOAP HTTP binding by using the POST method. As with the response MEP, the resource at the SOAP node is identified by a URL.

Since SOAP messages must be valid XML documents, they can be encoded in Unicode, as described in Section 2.1. This means that each text character may take up as much as two bytes. Encoding some data, e.g. an integer, takes up more space when compared to a binary representation. For example, an integer taking up 4 bytes in a binary representation may take up as much as 20 bytes using a UTF-16 encoding.

The extra network bandwidth used by SOAP compared to other middleware technologies will have an impact on the response time. It may be important to design for low bandwidth consumption. Otherwise, in some applications it may not be possible to transfer messages as quickly as required. However, the network bandwidth is dictated by the SOAP protocol, and would require a change of the encoding of SOAP messages. This could be done by choosing another way of serializing SOAP messages than XML, however such optimizations are beyond the scope of this report.

2.5 Web Services Description Language

This section describes the Web Services Description Language (WSDL) [36, 37, 38]. WSDL is an XML based language used to describe the services offered by Web services. This
is a central part of the Web service architecture, because it describes the protocol for communicating with a Web service.

![Diagram of service architecture](image)

**Figure 2.4:** An example of a resource offering a service through an interface bound to two different concrete protocols.

WSDL describes on an abstract level the messages that are exchanged between a service provider and a service requester. The exchange of messages between a service provider and a service requester is called an operation, and a collection of operations is called an interface. An interface is bound to one or more concrete protocols and message formats via bindings, which are accessible through one or more entities named “endpoints”, each identified by an URI. A service is a collection of endpoints bound to the same interface. Figure 2.4 shows an example of a resource offering a single service through an interface bound to two different protocols. In the following, the different elements of a WSDL document are described. These elements are illustrated in Example 2.11.

```
<definitions>
  <types>
    definition of types
  </types>
  <message>
    definition of a message
  </message>
  <interface>
    definition of a interface
  </interface>
  <binding>
    definition of a binding
  </binding>
  <service>
    definition of a service
  </service>
</definitions>
```

**Example 2.11:** The structure of a WSDL document.

### 2.5.1 Types

The type section of a WSDL document is used to define types that can be referred to in the rest of the WSDL document. The types are defined using XML Schema. Example 2.12
shows the definition of a type with the name “atype”. This type is a composite type, consisting of two sub-elements.

Example 2.12: An example of a type definition in WSDL.

2.5.2 Messages

Message sections in WSDL documents are used to specify the messages involved in performing operations supported by the Web service. A message can consist of one or more parts, which usually maps to the parameters of a function call in a programming language. Example 2.13 shows the definition of two messages, which each consist of one part. The first message named “testRequest”, consists of a part named “apart” and is of the previously defined type “atype” (lines 2-4). The second message named “testResponse” consists of a part named “anotherpart” which is of the type “integer” (lines 6-8). This type is provided by XML Schema.

2.5.3 Interfaces and Operations

An interface describes a set of messages that a service sends and receives. This is done by grouping related messages into operations. An operation is a set of input and output messages. Example 2.13 illustrates the definition of an interface (lines 10-16). This interface, named “ExampleInterface” (line 10), consists of one operation named “test” (lines 11-15). The definition of this operation states that the message pattern used in the operation is “request-response”, and that it as input takes the message named “testRequest”, and as output returns the message named “testResponse”.

WSDL allows specifying several different message patterns [37]. In WSDL, message patterns define the sequence, direction, and cardinality of messages sent or received by an operation. The direction of a message is either “in” or “out”, representing that the message is being received or sent by the service provider respectively. A message pattern is identified by assigning a URI as value to the “pattern” attribute of the “operation” XML element. Unless explicitly stated otherwise, WSDL message patterns abstract away binding-specific information like timing between messages, whether the pattern is synchronous or asynchronous, and whether the message are sent over a single or multiple channels. Three common message patterns are described below:
In-Only message patterns consist of exactly one message with a direction of “in”. This pattern is identified by the value “http://www.w3.org/2003/06/wsdl/in-only”.

In-Out message patterns consist of exactly two messages. Initially, a message with the direction “in” is sent, and as response a message with the direction “out” is sent. This pattern is identified by the value “http://www.w3.org/2003/06/wsdl/in-out”.

Request-Response message patterns are similar to in-out message patterns with one exception. A request-response message pattern explicitly states that that the message sent in the direction “out” should be sent on the same channel as the message sent in the direction “in”. This pattern is identified by the value “http://www.w3.org/2003/06/wsdl/request-response”.

2.5.4 Bindings

A binding describes a concrete binding of an interface and associated operations to a particular concrete message format and transmission protocol. Example 2.14 shows the definition of a binding of the “ExampleInterface” interface, defined in Example 2.13. This interface is bound to the SOAP 1.2 protocol, using HTTP as an underlying protocol and RPC serialized messages (lines 3-5). WSDL allows specifying two types serialization styles, namely “document” and “rpc”. These types describes the expected structure of SOAP messages. Using “document” serialization, SOAP messages are perceived as containing documents. Using “rpc” serialization, SOAP messages are perceived as containing remote procedure calls, and messages thereby contains parameters and return values from procedure invocations. The invocation semantics of an procedure depends on the type of the binding of the call to a specific protocol. Using the “rpc” serialization, the SOAP specification places some constraints on the permissible structure of SOAP messages.
Example 2.14: An example of the definition of a binding of the interface “ExampleInterface” defined in Example 2.13 to the SOAP 1.2 protocol, using HTTP as the underlying protocol.

2.5.5 Services

A service describes exactly one interface that a Web service provides, and the endpoints it is provided over. Example 2.15 illustrates the definition of a service with the name “ExampleService”, which exposes the operations defined in the “ExampleInterface” interface defined in Example 2.13 (lines 2-3). It is defined that this service can be accessed at the URI “http://example.com/ExampleService” (line 4). Furthermore, an endpoint of this service is defined (line 5-6) exposing the service through the protocol binding “ExampleBinding”, which was defined in Example 2.14.

Example 2.15: An example of the definition of a service providing the operations described in the interface defined in Example 2.13 using the protocol binding defined in Example 2.14.

2.6 XML Parsers

An essential part of a Web service application is creating and reading XML documents. In order to read such documents, a parser is necessary. Several classes of XML parsers exist, and this section introduces three classes of parsers and their properties.

2.6.1 W3C Document Object Model

The Document Object Model (DOM) is specified in the W3C DOM Technical Reports [30]. The purpose of the DOM is to supply a data model for representing the content, structure, and style of HTML and XML documents. An XML DOM parser reads an entire XML
document and creates a DOM representation of it in memory. The application using the parser is then able to access this structure as needed. The advantage of this approach is, that it is relatively simple for an application to access specific content and examine the structure of the document, which makes it suitable for complex documents. A disadvantage is that the entire document representation is resident in memory. A large document will take up a large amount of memory.

![DOM tree for the XML document illustrated in Example 2.1.]

**Figure 2.5**: DOM tree for the XML document illustrated in Example 2.1.

Figure 2.5 shows a graphic representation of a DOM tree, which is constructed from the XML document shown in Example 2.1. The square nodes correspond to XML elements, and the rounded nodes correspond to character data. The `href` attribute of the `picture` element is not seen as a separate node in the tree. Instead, it is seen as an attribute directly associated with the element. This is represented by the dotted line. An application would be able to traverse this tree representation to extract the data needed. Additionally, some DOM implementations allow modification of the data structure, as well as document generation based on a DOM tree. This makes DOM parsers useful for modifying existing documents.

### 2.6.2 Simple API for XML

The Simple API for XML (SAX) was developed in 1998, first as a Java-only API for XML parsers. SAX parsers have since been implemented in many other programming languages. The current version, SAX 2.0.1, is specified at the official SAX home page [22].

SAX parsers are event-based. The parser works by reading an XML document from start to end, tokenizing it, and firing events when encountering specific tokens. These events are handled by the application using call-back. Because SAX parsers push events to the application using it, they are also known as push parsers.

A SAX parser does not keep a record of the state of the XML document being parsed. It is up to the application to handle the structure of documents. This makes it is possible to tailor an application to support the needed functionality, making SAX parsers suitable for parsing documents with simple structure, or documents from which only some of the data needs to be handed to the application.

An important advantage of SAX parsers is that they require relatively small amounts of memory. Contrary to DOM parsers, SAX parsers do not store the entire document in memory. Instead, SAX parsers treat documents as streams, meaning that it is not necessary to wait until the entire document has been parsed before further processing of the parsed data can be initiated. This makes SAX parsers suitable for parsing large documents.
2.6.3 XML Pull Parsing

In 2002 a new API for XML parsing was developed, called XmlPull. The API is described on the official XML Pull Parsing Common API home page [26].

Pull parsers are similar to SAX parsers. They also do not build representations of documents in memory, but read the document token by token. The main difference from SAX parsers is that the application using the parser requests the next token from the document, until no more of the document is needed, or until the end of the document is reached. In other words, the application pulls data from the document. As with SAX parsers, pull parsers treat documents as streams. This gives all the advantages of SAX parsers. Since applications must request each token explicitly, pull parsers are especially suitable for applications that need to read entire documents.

2.7 Summary

This chapter has provided a technical analysis of the technologies that Web services is based upon. The technologies introduced were XML, WSDL, SOAP, and XML parsers.

XML is a human-readable meta language for structured data representation. The XML language can be extended using XML Namespaces to avoid naming ambiguities, and XML Schema to provide a means for describing the valid structure of XML documents. SOAP is an XML-based stateless message exchange protocol, which amongst other things, features special message types for describing Remote Procedure Calls. It makes use of the well-established XML standards, and is usually sent between peers using the widespread HTTP protocol. The services offered by a Web service, and the protocol used to communicate with it, can be described using the XML-based language WSDL. Finally, different types of XML parsers have been covered. Two major classes of parsers are SAX and DOM parsers, which each have their advantages and disadvantages.
This chapter discusses different approaches for creating Web services on distributed embedded systems. A choice is made to concentrate on the service provider aspect of Web services. Consequently, the design focuses on creating Web services, not clients of Web services.

As Web services is middleware layered on top of existing applications, an approach for creating Web services on distributed embedded systems will depend on the programming language in which the application is developed. This is because a mapping between the SOAP representation of messages and an appropriate representation in the programming language is needed. As the application resides on an embedded device, and since the C programming language is often used in this environment, a choice is made to design for the C programming language. Furthermore, the design of such a framework depends on the underlying transmission protocol of SOAP. As the only protocol binding formally described by the SOAP specification is the HTTP binding, this is chosen as the underlying protocol. Finally, the design also depends on the style of SOAP messages that are exchanged between the Web service and the service requester. A choice is made to focus on the SOAP RPC message exchange style.

Initially, two different approaches for designing a framework for creating Web services on an distributed embedded systems are presented. After making a choice of the most suited type of framework, it is described how data types from C can be mapped to a SOAP representation. Next, an algorithm is devised for creating a WSDL specification of the Web service middleware code that the framework generates. This is followed by the design criterias for the middleware code that the framework generates, and a description of the architecture of the generated middleware. Finally, it is discussed how a small memory footprint and a minimum CPU utilization can be achieved in such an architecture.

The result of this chapter is a design of a framework for creating Web service middleware, as well as a design of the architecture of the generated middleware.

### 3.1 Design of Framework

The Web service technology does not state anything about the development process of creating Web services. This is decided upon by the developer of the application which exposes
a resource as a Web service. This section presents two different approaches of frameworks for creating Web services, and clients of Web services, on an distributed embedded system, each supporting a different development process.

### 3.1.1 Abstract Data Model Framework

One approach for creating Web services, and clients of Web services, is to provide an abstraction that allows accessing the content of SOAP messages in a convenient manner. This can be achieved by providing a data model representing the content and structure of a SOAP message, in a fashion similar to that of XML DOM as described in Section 2.6.1. In such a scenario, the framework has the role of transforming SOAP messages into an easy accessible data structure, and provide an interface for creating SOAP messages without concern for the specific XML related details of the message.

Figure 3.1: An illustration of the different steps in a request-response MEP that a Web service client goes through when sending and receiving SOAP messages. The client is developed using a framework providing an abstract data-model of a SOAP message.

Figure 3.1 shows the different steps a Web service client has to go through in a request-response MEP. Initially, when an application desires to send a request to a Web service, a SOAP message data structure is created. This data structure provides an abstraction of a SOAP message, and the framework provides functionality to perform actions on this data-structure, such as setting the URI of the requested resource, setting the name of the procedure being invoked, adding SOAP header elements to the SOAP header, and adding parameters to the procedure call. Example 3.1 shows how such a message could be constructed in the C programming language using a standard interface.

When the application invokes the Web service (line 17 of the example), the framework marshals the data of this request, create a valid SOAP message, and send it to the specified URI. When receiving the response from the Web service, the framework has the role of performing unmarshalling of the received SOAP message and build a data structure representing the message. Example 3.2 illustrates how such a data-structure could potentially be accessed. The example assumes that the response received is valid.

In order to be able to access a Web service, the developer of the client application needs to retrieve the WSDL specification for that service from the service provider or from a service broker. Next, the developer interprets the specification in order to understand
#include "soapapi.h"

(...)

/* Creates a variable to hold the generated SOAP message. */
soap_message_t *request = create_new_soap_message();

/* Creates a variable to hold the response received when invoking the Web Service. */
soap_message_t *response = create_new_soap_message();

/* Build the SOAP message. */
set_target_uri(request, "http://example.org/personservice/");
set_procedure_name(request, "getPerson");
add_string_parameter(request, "cpr", "010245-6789");

/* Send the request to the Web Service and retrieve the response. */
send_soap_message(request, response);

/* Call a function that handles the received response. */
handle_response(response);

/* Frees resources */
free(request);
free(response);

(...)

Example 3.1: An example of an interface in the C programming language sending a SOAP message containing an RPC request to the “getPerson” procedure located at the URL “http://example.org/personservice/”. The interface is created in a framework providing an abstract data model of a SOAP message. The function handle_response is defined in Example 3.2.
```c
#include "soapapi.h"

void handle_response(soap_message_t *response) {
    /* Create variables to contain the XML elements. */
    xml_element_t *body;
    xml_element_t *get_person_response;
    xml_element_t *person;
    xml_element_t *name_element;

    /* Variable to hold the name of the retrieved person. */
    char *name;

    /* Retrieve the name of the person from the response. */
    body = get_soap_body(response);
    get_person_response = get_sub_element(body, "getPersonResponse");
    person = get_sub_element(get_person_response, "person");
    name_element = get_sub_element(person, "name");
    name = get_content(name);

    printf("The person with cpr 010245-6789 has the name \%s", name);

    /* Frees resources */
    free(body);
    free(get_person_response);
    free(person);
    free(name_element);
    free(name);
}
```

**Example 3.2:** An example of handling the response to the request sent in Example 3.1. This example assumes that a valid response is received, and does not handle errors.
the interface of the Web service. The developer then creates the necessary logic to generate requests to the service, and to process the responses received.

Figure 3.2 illustrates the process of creating a Web service provider in the abstract data model design. Initially, the application initializes the framework to start a service listening for SOAP requests. Whenever a new request is received, the framework has the role of unmarshalling the request into a data structure that the application can access. Next, this data structure is handed on to the application, which has the responsibility of interpreting the data received and decide how to handle it. Then, the application generates a data structure to hold the result of processing the request, which is then sent to the framework. The framework then marshals the result data structure, creates a valid SOAP message, and sends it to the requesting client. Example 3.3 shows how a Web service could be generated in C, using this method.

![Figure 3.2: The different steps in a request-response MEP that a Web service provider goes through when retrieving a SOAP message, using an abstract data model framework.](image)

**Evaluation**

The advantage of using a framework as described in this section, is that the application developer gains full control over the messages sent and received. This can for instance be very useful when the application needs to create complex interactions, for example when including SOAP intermediaries. Furthermore, the framework itself becomes simple because most of the processing logic is moved to the application. However, this can also be seen as a disadvantage since the application must include SOAP-aware code. Furthermore a disadvantage is that a representation of the entire SOAP message in memory is needed, which is not a desirable property in embedded systems, due to memory constraints.

### 3.1.2 Language Binding Framework

Another approach to designing a framework for creating Web services, and clients of Web services, is to create a language binding framework. Instead of providing an abstraction of SOAP messages, the SOAP-specific details are made transparent to the developer.

In such a framework, when creating a Web service, a C header file specifying the functions that should be exposed as a service, is given as input to the framework. This is illustrated in Figure 3.3. The figure uses the terminology “host” and “target”. The “host” refers to the computing system on which the application is developed, and on which the framework software resides. The “target” refers to the embedded system, on which the application will be executed. The framework has the responsibility of creating skeleton code that takes care of receiving requests and invoking the requested functions. The user of the framework has the possibility of suppling options to the framework, which will affect the generated code. An example of such an options, is specifying how many requests that
```c
#include "soapapi.h"

/* Functions that handle incoming */
/* requests to the Web Service. */

soap_message_t *requestHandler(soap_message_t *request) {
    soap_message_t *response = create_new_soap_message();
    xml_element_t *person = create_new_xml_element("person");
    xml_element_t *name = create_new_xml_element("name");

    /* Add the name element as a sub-element of the person element. */
    add_sub_element(person, name);

    /* Set the content of the name element. */
    set_content(name, "Linus Torvalds");

    /* Check if a request has been made for the getPerson procedure. */
    if (strcmp(get_requested_method_name(request), "getPerson") == 0) {
        set_response_name(response, "getPersonResponse");
        set_response_element(response, person);
        return response;
    } else { /* Error handling */
        return NULL;
    }
}

/* Set up the server and declare that the function */
/* "requestHandler" will handle incoming requests. */
int setup_web_service_server((void *)request_handler);
start_server();
```

**Example 3.3:** An example of setting up a Web service provider in a framework providing an abstract data model of SOAP messages. Errors in incoming requests are not handled.
can be processed by the Web service concurrently. In such a framework, the application can expose its functionality as a Web service without specific knowledge about SOAP.

Figure 3.3: Overview of the system architecture in a language binding framework (server).

Creating a client of a Web service client is performed in a similar manner. A WSDL file is given as input to the framework, which has the responsibility of generating stub code that can be invoked directly from the client application. Figure 3.4 shows how a Web service client is generated. Initially, the WSDL specification of the requested Web service is then passed to the framework, which has the responsibility of creating stub-code. Along with the WSDL specification, the user of the framework can specify various options which will effect the generated code, e.g. whether the generated code should be optimized with respect to code size, memory footprint, or execution time. Along with the generated stub code, the framework generates C header files that specifies the interface of the Web service in a C specific manner. This is used by the application accessing the Web service to generate requests to the Web service. In order to produce an executable, the application source code is compiled and then linked with the generated stubs.

Figure 3.4: Overview of the system architecture in a language binding framework (client).

Evaluation

Compared to using the abstract data model framework, the development process in the framework described in this section, becomes less complex. This is because the application has no specific SOAP knowledge, due to the framework handling the exchange of SOAP
messages, and invoking the proper functions. However, the framework itself becomes more complex, since it needs to be able generate stub code from WSDL specifications, as well as generating skeleton code and WSDL specifications from C header files.

In a language binding approach, the framework has knowledge about the functionality the Web service exposes. This opens the possibility of making the framework perform optimizations that are not possible with the abstract data model design, because the framework knows the expected exchange of SOAP messages. These optimizations could include having the framework return a SOAP fault and stop processing immediately when detecting a problem with an incoming SOAP message, e.g. a request being made for a non-existing resource. This is not possible in the abstract data model approach, since the framework does not have the possibility to perform these checks. Instead, it must be handled by the application, and so, the entire message must be processed by the framework before all possible errors which are not related to wellformedness of the SOAP message, can be detected.

### 3.1.3 Choice of Framework

A choice is made to use the language binding approach, because it provides a simpler application development model than the abstract data model approach, and because it allows performing optimizations not possible with the abstract data model design.

Furthermore a choice is made to focus on the service provider part of the Web service architecture, meaning the part of the framework responsible for creating Web services, and not the part of the framework responsible for creating clients of Web services. In the following a design of the Web service middleware generator is given, along with a design of the architecture of the generated middleware source code.

### 3.2 Creating Web Services

This section describes the part of the language binding framework having the responsibility of generating Web services from existing C function prototypes. The responsibility of this part of the framework is to generate WSDL specifications of the services being exposed by the Web service, and to generate skeleton code that can interpret requests to the service and invoke the proper functions implementing these services. This section covers how services are specified in the framework, and how a mapping can be created between C and WSDL. For sake of simplicity, a choice is made only to support a subset of C data-types, and only allow a subset of the specifications traditionally possible in C header files. The algorithms and methods developed in this chapter could potentially be extended to support a larger subset, however that is out of the scope of this report.

#### 3.2.1 Mapping C Data Types to XML Schema Data Types

Whenever exposing a C function as a part of a Web service, a mapping between the data types of the parameters that the function takes, and the data types used in the WSDL specification of the service, is necessary. As WSDL uses XML Schema to provide typing information, a mapping between C data types and XML Schema is necessary.

**Basic Data Types**

Basic data types in C maps simply to the data types specified in XML Schema. C types such as `int`, `unsigned int`, `float`, `double`, `char`, and `char[]` has equivalent counterparts
in XML Schema. The names of these in XML Schema are denoted `integer`, `unsignedInt`, `float`, `double`, `byte`, and `string` respectively. Example 3.4 shows an example of how part of a WSDL specification of a C function can be created, taking basic data types as parameters.

```xml
<message name="testFunctionRequest">
  <part name="a" type="integer"/>
  <part name="b" type="float"/>
  <part name="c" type="string"/>
</message>
```

Example 3.4: An example showing a part of the WSDL specification, exposing an interface to a C function taking three parameters with basic data types.

### Structs

Structs similarly maps to the data types specified in XML Schema, which provides the `sequence` construct. This construct allows specifying a complex data type which holds a certain sequence of sub data types. Example 3.5 shows how a C struct can be mapped to an XML Schema representation.

```xml
<types>
  <schema targetNamespace="urn:myschema"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <complexType name="mystruct">
      <sequence>
        <element name="field1" type="integer"/>
        <element name="field2" type="string"/>
      </sequence>
    </complexType>
  </schema>
</types>
```

```c
typedef struct {
  int field1;
  char field2[];
} mystruct;
```

```xml
function test(mystruct a);
```

```xml
<message name="testRequest">
  <part name="a" type="mystruct"/>
</message>
```

Example 3.5: An example part of a WSDL specification of a C function taking a struct as parameter.

### Arrays

In order to map arrays in C to an XML Schema representation, a concept called `restriction` is applied. In C, there is no mechanism for deciding the size of an array when it is passed as a parameter to a function (with the exception of `char` arrays, which are null-terminated), and therefore, when creating functions taking array parameters, additional parameters are passed stating the size of the array. Therefore, this method is adopted in the design of the framework. Whenever a C header file contains a function with an array parameter, the framework checks whether a size parameter is present. The convention is that the type of this parameter is `unsigned int` and that is has the name of the array parameter with the value “\_size” appended to it (e.g. a function with the parameter `float a[]` should be followed by an `unsigned int a\_size` parameter. If the function does not contain such a parameter, the framework must report an error. Example 3.6 shows a block
of WSDL specifying that a function with the name “test” can be called, and that it takes a
parameter with an int[] type and the name “a”. In WSDL this parameter is specified to
be a complex type, which has an unspecified amount of sub-elements named “item”. This is
specified by the minOccurs and maxOccurs attributes in the element element in line 6-7
of the example. Each of these represents one element in the int array.

It should be noted that the generated WSDL does not mention the a_size parameter,
since it is a C specific detail that this parameter is needed. Whenever requests are sent
to the generated Web service, the framework must parse the message and unmarshal the
character data of the item elements while creating a C representation of the array. It must
then deliver this array to the function along with the size of the array, using the a_size
parameter.

```xml
<message name="testRequest">
  <part name="a">
    <complexType>
      <element name="item"
        type="integer"
        minOccurs="0"
        maxOccurs="unbounded"/>
    </complexType>
  </part>
</message>
```

Example 3.6: Part of a WSDL specification for a C function taking an integer array as a parameter.

### 3.2.2 Specifying Parameter Passing Type

As SOAP RPC messages allows for different types of parameter passing, a mechanism to
specify this type in C is necessary. In SOAP, parameters can either be of the in, out,
or inout passing type. Parameters of the type in are passed as input to a procedure,
and out parameters are used for output from a procedure call. Parameters of the type
inout are used for both input and output from a procedure. A convention for naming
of C functions is defined, saying that when exposing a C function as a Web service, the
type of the parameter and the character “_” is prepended to the parameter name (e.g. the
name of parameter a that is of the passing type inout would have the name inout_a). In
order to allow for out and inout parameters, these have to be defined as pointers in C.
Example 3.7 shows an example of the specification of a C function taking an in and out
parameter in WSDL.

### 3.2.3 Specifying Header Parameters

In order to allow the application developer to specify that some certain elements are to
be sent as part of the SOAP header, a convention for specifying this is introduced. When
exposing a C function as a Web service, the name of a parameter can be prepended with
“header_” to specify that this should be passed as a header element in the SOAP header.
Example 3.8 shows how a C function, taking an int parameter that should be send as
part of the SOAP header, can be specified in WSDL.
Example 3.7: Part of a WSDL specification for a C function taking an in and an out parameter.

Example 3.8: Part of a WSDL specification for a C function taking in, out and header parameters.
3.2.4 Algorithm for Creating WSDL Specification

This section describes part of an algorithm for converting a C header file specification of a Web service to a WSDL specification. When specifying the pseudo code for the algorithm, the notation “<” is used to specify that a value is assigned to a variable. The notation “.” is used to specify that two strings are concatenated. A string is delimited by two “’” characters.

Figure 3.5 illustrates the createWSDLTypeDeclaration algorithm. This algorithm is responsible for converting a C type declaration to a type declaration in WSDL. As stated in the introduction, only a subset of the C types are addressed in the design, and therefore this algorithm only deals with mapping type declarations of C structs to WSDL data type declarations. The algorithm takes two parameters: td and r. The parameter td holds the C type declaration being converted, and the parameter r holds the result of the algorithm. Initially, the algorithm checks if the type of the C type declaration passed as input, is a declaration of a C struct (line 1). If this is the case, then an XML element of the type complexType is added to the result (line 2), and a sequence element is added to the result (line 3). Then, for each sub-element of the C struct, the createWSDLTypeDeclarationRecursive algorithm is called with the sub-element as parameter (lines 4-5), and after this call has finished, the proper XML end elements are added to the result (lines 6-7).

createWSDLTypeDeclaration(td, r):
1: if (isStruct(td)):
   r ← r . '<complexType name="' . name(td) . '">'
   r ← r . '<sequence>
   foreach element e ∈ subelementsof(td):
5:     createWSDLTypeDeclarationRecursive(e, r);
   r ← r . '</sequence>'
   r ← r . '</complexType>'
else if (....) // Handle other C type declarations

Figure 3.5: The createWSDLTypeDeclaration algorithm. The parameter td holds the type declaration being converted, and the parameter r holds the current result of the algorithm.

The createWSDLTypeDeclarationRecursive algorithm is responsible for creating the WSDL of the sub elements of the C struct, and to call itself recursively if it encounters elements that are not basic data types. The pseudo code of this algorithm is illustrated in Figure 3.6. Initially, this algorithm checks if the sub element of the struct is a struct itself. If this is the case, an XML element with the name “element” is added to the result, along with a complexType and a sequence XML element. Next, the algorithm calls itself recursively for all the sub elements of the C struct. If this is not the case, it is checked if it is a simple type, and if this is the case, an XML element with the name “element” and the proper XML Schema type is added to the result (lines 10-15). Example 3.9 shows the output of the algorithm, when it is executed on an C struct declaration.

Figure 3.7 shows the pseudo-code for the createWSDLProcedureDeclaration algorithm. This algorithm is responsible for converting declarations of C functions to their equivalent representation in WSDL. The algorithm works by creating three different WSDL message declarations, one for incoming parameters, one for outgoing parameters and one for header parameters (lines 1-18). The declaration of each parameter the function takes
createWSDLTypeDeclarationRecursive(td, r):

1: if (isStruct(td)):
    r ← r . '<element name="" . name(td) . '"/>
    r ← r . '<complexType>
    r ← r . '<sequence>
5: foreach element e ∈ subelementsof(td):
    createWSDLTypeDeclarationRecursive(e, r);
    r ← r . '</sequence>'
    r ← r . '</complexType>'
    r ← r . '</element>'
10: else if (isSimpletype(td)):
    if (isInt(td)):
        r ← r . '<element name="" . name(td) . '" type="integer" />'
    else if (isFloat(td)):
        r ← r . '<element name="" . name(td) . '" type="float" />'
15: else if (....) Handle other basic C types
    else if (....) Handle other types of sub-elements

Figure 3.6: The createWSDLTypeDeclarationRecursive algorithm. The parameter td holds the sub
    element of a C struct declaration being converted, and the parameter r holds the current
    result of the algorithm.

typedef struct {
    int field1;
    float field2;
    struct {
        int subfield1;
    } mysubstruct;
} mystruct;
<complexType name="mystruct">
    <sequence>
        <element name="field1" type="integer" />
        <element name="field2" type="float" />
        <element name="mysubstruct">
            <complexType>
                <sequence>
                    <element name="subfield1" type="integer" />
                </sequence>
            </complexType>
        </element>
    </sequence>
</complexType>

Example 3.9: An example part of how the createWSDLTypeDeclaration algorithm converts a C type
declaration.

is handled by the createWSDLProcedureParameterDeclaration algorithm. When all the
messages has been created, a WSDL “interface” element is created in order to specify that
the function can be called, and to define the messages that can be sent to it (lines 20-31).

The createWSDLProcedureParameterDeclaration algorithm is illustrated in Figure 3.8.
This algorithm creates a WSDL “part” XML element for each parameter, with the type of
the parameter set to the corresponding XML Schema type (line 9). If the parameter is an
array, the restriction concept is applied (lines 1-7) as described in Section 3.2.1.

3.3 Target Application

This section covers the design of the target application. This application is generated by
the framework running on the host machine. The generated application is called the target
createWSDLProcedureDeclaration(fd, r):

1: r ← r . '<message name="' . name(fd) . ' Request">'
   foreach parameter p ∈ inParameters(fd):
      createWSDLProcedureParameterDeclaration(p, r):
   endforeach

2: createWSDLProcedureParameterDeclaration(p, r):
   foreach parameter p ∈ inoutParameters(fd):
      createWSDLProcedureParameterDeclaration(p, r):
   endforeach

3: r ← r . '</message>'

4: r ← r . '<message name="' . name(fd) . ' Response">'
   foreach parameter p ∈ inoutParameters(fd):
      createWSDLProcedureParameterDeclaration(p, r):
   endforeach

5: createWSDLProcedureParameterDeclaration(p, r):
   r ← r . '</message>'

6: r ← r . '</message>'

7: foreach parameter p ∈ headerParameters(fd):
   createWSDLProcedureParameterDeclaration(p, r):
   r ← r . '</message>'

8: r ← r . '<interface name="WebServiceName">
   r ← r . '<operation name="' . name(fd) . '"
       pattern="http://www.w3.org/2003/06/wsdl/request-response">
   r ← r . '<input message="' . name(fd) . ' Request">
   foreach parameter p ∈ headerParameters(fd):
      r ← r . '<soap:header message="' . name(fd) . ' Header" part="' . name(p) . '" use="encoded">
   r ← r . '</input>'
   foreach parameter p ∈ inoutParameters(fd):
      createWSDLProcedureParameterDeclaration(p, r):
      r ← r . '</input>'
   endforeach
   r ← r . '</operation>'
   r ← r . '</interface>'

Figure 3.7: The createWSDLProcedureDeclaration algorithm. The parameter fd holds the C function declaration being converted, and the parameter r holds the result of the algorithm.

application, and designed to run on embedded devices. First, the design criteria for the target application are developed. Then, the overall architecture of the target application is designed, based on these criteria.

3.3.1 Design Criteria

The list below describes how various design criteria are weighted for the target application. These criteria are inspired by Mathiassen et al. [13].

- Efficiency
  The characteristics of embedded devices makes efficiency a central design goal of software for embedded systems. The main focus of this report is on testing whether
createWSDLProcedureParameterDeclaration(p, r):

1: if(isArray(p)):
    r = r + '<part name="' + name(p) + '" type="' + type(p) + '" minOccurs="0" maxOccurs="unbounded" />'
5: else:
    r = r + '<part name="' + name(p) + '" type="' + type(p) + '" />'

Figure 3.8: The `createWSDLProcedureParameterDeclaration` algorithm. The parameter `p` holds the C parameter declaration being converted, and the parameter `r` holds the result of the algorithm.

it is feasible to implement SOAP middleware on embedded devices. For this reason, efficiency is an important criteria for the design of the system. Efficiency can be measured in several ways.

One such measure is the amount of CPU cycles needed to process messages. In DE systems the middleware may not be allowed to disrupt other services. To achieve acceptable response times, the consumption of CPU cycles should be kept to a minimum. This is especially important when real-time requirements must be enforced.

Other measures are memory consumption, both in terms of code size of the generated target application, and memory consumption at runtime. With the limited amount of memory available on embedded systems, the middleware should take up as little memory as possible.

- **Reusability**
  The application generated by the framework does not need to be resusable, since it is generated for a very specific purpose. The application will only be able to solve its very specific task. For that reason it does not need to be designed for reusability.

- **Portability**
  The application may need to take advantage of platform specific features in order to improve performance or reliability. The application itself will not need to be portable. It should be the responsibility of the host application to ensure portability of the applications it generates, by being able to generate applications for various platforms. Therefore, the generated applications need not be designed for portability.

- **Maintainability**
  Since the application is generated for its specific purpose, it makes little sense to design for maintainability. There is no need to generate human readable code. Therefore, maintainability is not an relevant design goal.

- **Reliability**
  Reliability is an important design goal in most computing systems. It can be argued that embedded systems software should be very stable, since user inspection is often not possible. Some embedded systems are able to reset quickly in case of failure, often without it being noticed. These systems may utilize this ability to achieve reliability. In general, reliability should be achieved by designing for stable software,
making reliability an important design goal in production systems. However, as the main focus of this report is to investigate the performance of Web services, it will not be in focus in the remainder of the report.

- **Interoperability**
  The generated SOAP application is acting as distribution middleware. It is the primary goal of this application to improve the interoperability of diverse systems.

- **Security**
  Security is of general concern when dealing with distributed systems. The generated middleware will act as an interface to varying services. Although security is important in general, it is considered to be outside the scope of this report.

- **Testability**
  When developing software for embedded systems it is important to be able to run tests to ensure partial correctness of the system. This may however lead to performance degradation. A production system may leave out the testing code to improve the performance. This report will not consider how to generate testable applications.

### 3.3.2 Processing Model

SOAP requests are received by the target application by a TCP connection. These incoming connections should be handled in such a way that more than one client can be serviced concurrently. Consequently, the application must be able to accept new connections while processing other requests.

![Threaded architecture of embedded application.](image)

This can be achieved by adopting a threaded processing model as shown in Figure 3.9. Each box represents a function in the application. Each vertical bar represents the execution of a thread performing the related function. As time progresses the threads will either be blocked (white) or running (black).

The job of the network listener is to listen for incoming connections. When a connection has been established, it is passed on to a SOAP processing thread. The network listener thread is then ready to accept another connection. Several processing threads should exist, such that the listener can hand over a new request, while a previous request is still being processed.
This solution is a trade-off between efficiency and responsiveness. The multi-threaded model takes up more memory, increases the code size, and increases the number of CPU cycles required compared to a single-threaded model. On the other hand, it improves the responsiveness of the system.

The set of steps that a SOAP processing thread must go through when processing a message is the same for all SOAP requests received. These steps are depicted in Figure 3.10.

![Figure 3.10: Overview of the embedded architecture.](image)

Each box represents a step. The arrows represent data that are input/output at a step. Each step is now described in detail.

- **HTTP parser** - The first task of the SOAP processing step is to read the incoming HTTP request from the connection, which has just been handed over by the network listener. The HTTP parser splits the request into HTTP headers and an HTTP body. The headers contain the name of the service being requested, so they are passed on to the service lookup step. The body contains the SOAP request, which is passed on to the XML parser step. If the HTTP request is invalid, a SOAP fault message should be returned to the sender.

- **Service lookup** - In this step the name of the requested service is mapped to a service handler. If the service is not found, a SOAP fault message should be returned to the sender. Otherwise, the associated service handler is invoked.

- **XML parser** - This step parses the body of the incoming XML document. A lexical analysis is performed on the document, meaning that it splits the document into tokens useful for further analysis. Notice that this step is not SOAP aware. If the XML document is not well-formed, a SOAP fault message should be returned to the sender.

- **Service handler** - Each service has a service handler associated with it. The responsibility of each handler is to check the validity of the SOAP request, as well as to determine which action must be taken as a consequence of the request. A service handler contains service specific code, generated by the host application. Each service handler should only be able to handle SOAP requests for that specific service. When the service handler encounters a message, which it is unable to process, it should return a SOAP fault message to the sender. The result of this step is that a C function to be invoked as identified along with the parameters it should be invoked.

- **Function execution** - This step invokes a C function identified by the service handler. If the invocation fails a SOAP fault message should be returned to the
sender. The result of this step is a description of the output of the function. This result consists of the return value of the function, as well as the values of any out/in-out parameters.

- **SOAP message generator** - This step generates the SOAP response message that should be returned to the sender. The results provided by the function execution step should be marshalled and inserted into a SOAP response message template. This message can then be passed on to the HTTP reply generator. No errors should occur in this step, since the result is based on a validated SOAP request. If an error does occur, a SOAP fault message should be returned to the sender. The result of this step is a valid SOAP RPC response message.

- **HTTP reply generator** - This step packages the SOAP response message into an HTTP response message. The response message consists of a list of HTTP headers and the SOAP response message. When the HTTP message has been constructed, it is transmitted to the receiver.

The advantage of this processing model, is that the generated target application can support exactly those features required for the specific types of SOAP messages it must handle, thereby reducing code size, memory consumption, and CPU usage.

### 3.3.3 Common Code

Some of the processing steps involved in Figure 3.10 are identical for all applications, because they provide some general functionality for the target application. These steps are the HTTP parser, XML parser, service lookup and HTTP Reply generator. Therefore these are designed as standard components that will always be included in the target application, which provide functionality to the service handler, the function execution and the SOAP message generator step.

#### HTTP Parser

The HTTP parser provides functionality to parse the HTTP headers of the request, and to read the body of the request, if the request method contains a body. The parser is designed to use only a fixed amount of memory, and to read only a part of the HTTP request at the time. Figure 3.11(a) illustrates how HTTP headers are read. Initially memory is allocated for a buffer to hold one HTTP header line. Next the first line of the request is read in order to identify the HTTP request method, the requested resource, and the HTTP protocol version being used. Then a loop is entered where one header line is read at a time. If the header is relevant it is saved in a data structure and otherwise it is discarded. Examples of relevant headers, are headers describing the content type of the request, and headers describing whether persistent connections should be used or not. Examples of irrelevant headers, are headers identifying the software sending the request (e.g. a `User-Agent` header), and headers that are not supported by the HTTP server implementation. Figure 3.11(b) shows how the HTTP body is handled by the HTTP parser. Initially a fixed size of memory is allocated. Then a loop is entered, where part of the HTTP body is read from the network, and put into memory. This part is then processed by the XML parser, and when parsing has finished the loop continues, until the entire HTTP body has been processed.
3.3 Target Application

Service Lookup

This step is responsible for identifying which service that has been request in the HTTP header. This step is general for all requests being processed, and therefore this is designed as a standard component that provides functionality to the generated target application to register services in a data structure. Whenever a HTTP request is received this component has the responsibility of examining if a service is registered with the requested name, and if so forward control to the appropriate service handler.

XML Parser

As covered in Section 2.6 there are several approaches to XML parsing. The main distinction between these approaches, that is of interest in this design, is whether they place a representation of an entire XML document into memory and then delivers it to the application, or whether only smaller parts of the document is read and parsed at the time.

The first approach requires that memory is allocated to hold a representation of the entire document. The application must then investigate this representation as needed. Since the size of an incoming SOAP request is not known in advance of a request, it is not possible to limit the memory consumption to some fixed level. This is not a desired property for embedded applications, since the available amount of memory is typically limited.

The second approach opens the possibility of allocating a fixed amount of memory for parsing, since only a fixed size of the XML document is parsed at the time. Therefore such a parsing method is chosen. As shown in Figure 3.11(b), a part of the document is read and parsed at the time.

HTTP Reply Generator

The HTTP reply generator has the responsibility of generating the proper HTTP headers, as a response to a request, and to package the SOAP message into a HTTP response.
As this is functionality necessary for completing each request, this is designed to be a component that is always included in the target application, providing functions that the SOAP message generator can use.

### 3.3.4 Service Specific Code

The remaining steps in the processing model not covered in Section 3.3.3, contains functionality that is specific to the generated target application, and depends on the functionality exposed as a Web service. These steps are the service handling, function execution, and SOAP message generation, which will be discussed below.

#### Service Handler

This step is responsible for unmarshalling SOAP requests, and identifying the C function that has been requested to be invoked. To identify the function name and the parameters to this function, it needs to analyze the tokens retrieved from the XML parser. The design of this step takes advantage of the fact that the expected structure of the incoming SOAP message is known in advance. This makes it possible to optimize the processing of the SOAP message.

```
<soap:Envelope
    xmlns:n="urn:xmethodsBabelFish"
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
    xmlns:soapenc="http://www.w3.org/2003/05/soap-encoding"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
    <n:BabelFishRequest>
      <translationmode xsi:type="string">en_de</translationmode>
      <sourcedata xsi:type="string">SOAP is cool</sourcedata>
    </n:BabelFishRequest>
  </soap:Body>
</soap:Envelope>
```

**Example 3.10:** An illustration of the different states the service handler goes through when processing the SOAP message from Example 1.2. A state is marked with a filled black circle with a number inside it.

Example 3.10 shows how the SOAP message from Example 1.2 can be divided into different states. Initially, when the service handler starts processing this message, it is in state 1. In this state, the service handler expects to meet a SOAP envelope element. If such an element is encountered, the service handler proceeds to state 2. Otherwise, an error is raised, and the service handler skips the rest of the SOAP message, and sends an error message to the SOAP message generator. Similarly, when the service handler is in state 2, it expects to meet a SOAP body element, and after this, it expects to meet a request for one of the C functions that the Web service offers. In Example 3.10 it is assumed that a request is made for a C function named “BabelFish”.

After encountering the XML start tag with the name “BabelFishRequest”, the service handler is in state 4. In this state, the service handler expects to meet a XML element with the name of one of the parameters the function takes (i.e. either `translationmode` or `sourcedata`). Assuming that the first parameter element is `translationmode`, the service
3.4 Summary

This chapter has discussed different approaches for creating Web services for distributed embedded systems, and a choice of a language binding framework has been made, which make the creation of Web services transparent for the programmer. A choice has been made to concentrate on the service provider aspect of Web services. Consequently, the design focuses on creating Web services, not clients of Web services.

A partial design of a framework for creating Web services for distributed embedded systems has been provided. This framework takes a C interface specification as an input, and outputs the middleware code needed to make this interface available as a Web service. Additionally, an algorithm is given for creating a WSDL specification of the generated Web service.
A design of the architecture of the middleware application generated by the framework has been given. The design lists a set of design criteria for this application, and specifies how it must process incoming messages. The processing of messages is done in small parts at the time as they arrive from the network, thereby avoiding to have an entire message represented in memory at the time. It is chosen to use a SAX inspired XML parsing technique, because it features low memory consumption and enables extensive use of static memory allocation. Additionally, a method for building service specific code is introduced. This code is optimized to handle requests for specific services.

The design has established a basis for developing a prototype implementation of the generated middleware application.
This chapter discusses the implementation of a part of the framework discussed in Chapter 3. A choice has been made to focus on the target application generated by the framework. The goal of the implementation is to provide a basis for performing profiling tests, in order to establish the applicability of Web services in distributed embedded systems. Throughout the implementation, the focus has been to minimize CPU utilization and memory consumption. Initially the functionality which is general for all applications generated by the framework is discussed. This is followed by a discussion of the functionality specific to each generated application.

4.1 Common Code

As described in Section 3.3.2, the target application generated by the framework contains some functionalities that is general for any application. The general functionalities are the HTTP parser, service lookup, XML parser, and the HTTP reply generator steps. Furthermore, the implementation identified another element in the generated target application that is general for any generated application, namely the operating system. Initially, this section describes the operating system chosen as a basis for the implementation, followed by descriptions of the implementation of the common processing steps identified in Section 3.3.2.

4.1.1 Choice of Operating System

The operating system chosen as the basis for the generated target application, is eCos, which is short for Embedded Configurable Operating System [8]. eCos is an open source real-time operating system intended for embedded applications and its modular architecture, makes it highly configurable. This makes it possible to tailor the operating system to the specific needs of the application. Furthermore, its networking support makes it highly suitable for distributed embedded systems.

eCos works with the GNU open source development tools, including support for debugging with the GNU Project debugger, gdb. It also includes a special version of the C standard library. This makes it relatively simple to port existing applications, which
depend on the standard library, to eCos.

**Application Development in eCos**

Developing applications for eCos involves creating a configuration for eCos and compiling it using this configuration. This creates a library file containing the operating system that can be linked to an application. The resulting binary contains both the operating system and the application.

The eCos configuration used for the prototype implementation is reduced to include as few modules as possible, to reduce code size and memory footprint. The most central parts included are:

- **The eCos kernel** which includes thread support, a scheduler, and primitives for synchronization and resource locking.
- **Ethernet support** which is required for connecting the device to an ethernet network.
- **TCP/IP stack** which is required for communicating using the TCP protocol. There are two different stack implementations available, namely the FreeBSD and the OpenBSD stacks. The OpenBSD stack was chosen, since some degree of unstable behavior was experienced when using the FreeBSD stack.
- **ISO C library** which provides useful functionality, such as string handling, I/O functions, and general utility functions.
- **Dynamic memory allocation** which is needed by some parts of the code to allocate memory in cases where static allocation is not sufficient.

It is important to note that it is up to the application developer, using the framework to configure eCos correctly. E.g. if the application needs a special math library, to support its functionality, the application developer needs to configure eCos to include the proper module supporting this functionality.

### 4.1.2 Thread Pooling

As mentioned in Section 3.3.2, the target application has been designed to support threading, in order to obtain the needed degree of availability of a Web service. As creating and destroying threads requires resources, a technique called “thread pooling” has been implemented. This technique aims to maintain a set of threads during the execution of the program. When a request arrives, an available thread is assigned to handle the request, and the thread is marked as being busy, meaning that it can not handle any more requests. When the request has been processed, the thread is marked as available, and is ready to process another request. Using this method, the overhead of creating and destroying threads is avoided, thereby saving resources.

Consequently, the initiating step in the target application, is to create a number of threads to handle requests. These threads are denoted “worker threads”. Furthermore, a thread for maintaining information about the state of the worker threads is initialized. This thread is denoted “listener thread”. The functionality of the listener thread is defined by the `listen_program()` function, partly illustrated in Figure 4.1. The `listen_program()` function initializes a server socket (not illustrated in the example), which is used to receive incoming connections. After having initialized this socket, the `listen_program()` function
4.1 Common Code

listens for incoming requests, and assign threads to handle these (lines 11-35). If too many requests are received to be handled concurrently, a response is send to the client, stating that the service is temporary unavailable (lines 37-41).

```c
void listen_program(cyg_addrword_t data) {
    /* Initialization. */
    (...)

    for (;;) {
        /* Flag indicating wether a ready thread has been found. */
        found_thread = 0;

        /* When a request arrives, cs holds the */
        /* socket descriptor of the client. */
        if ((cs = accept(s, (struct sockaddr *)&ca, &ca_len)) != -1) {
            /* Mutex handling. */
            cyg_mutex_lock(&ready_workers_lock);

            /* Searches all threads, in order to find */
            /* an available thread. */
            for (i = 0; i < WORKER_THREADS; i++) {
                /* An available thread was found. */
                if (ready_workers[i] == 0) {
                    /* Flag the worker thread as busy. */
                    ready_workers[i] = 1;

                    /* Set the socket descriptor that the worker thread */
                    /* should read data from. */
                    client_sockets[i] = cs;

                    /* Start worker thread. */
                    cyg_mutex_unlock(worker_thread_mutex[i]);

                    /* Set flag and exit. */
                    found_thread = 1;
                    break;
                }
            }

            /* Available thread was not found. */
            if (found_thread == 0) {
                /* Send 503 header to the client, stating that */
                /* the service is currently unavailable. */
                (...)
            }
        }

        /* Mutex handling. */
        cyg_mutex_unlock(&ready_workers_lock);
    }
}
```

Figure 4.1: The listen_program()’s main loop, assigning threads to incoming requests.

The functionality of a worker thread is implemented by the worker_program() function, which is illustrated in Figure 4.2. Initially the worker_program() sets a flag, stating that it is ready to process a request (lines 5-7). Next, the function waits for a mutex variable to become unlocked. This happens when a request is ready to be processed, and it has been assigned to this thread (this happens in Figure 4.1, line 29). Next a loop is
entered (lines 15-35) where the HTTP request is processed, as described in Section 4.2.1. Finally the connection is closed (line 37), and the thread once again signals that it is ready to process a new request (lines 5-7).

```c
void worker_program(cyg_addrword_t data) {
    for (;;) {
        /* Set flag, to indicate that the thread */
        /* is ready to process requests. */
        cyg_mutex_lock(&ready_workers_lock);
        ready_workers[i] = 0;
        cyg_mutex_unlock(&ready_workers_lock);

        /* Waits for a mutex variable to become locked. */
        /* This will happen when a request is awaiting */
        /* to be processed. */
        cyg_mutex_lock(worker_thread_mutex[i]);

        /* Enter a loop, where HTTP requests are received. */
        keep_alive = 1;
        while (keep_alive == 1) {
            /* Initialize a httpRequest data-structure. */
            initializeRequest(&req, client_sockets[worker_id]);

            /* Handle the headers of the request. */
            httpReadRequest(&req);

            /* Check whether the HTTP request is using */
            /* persistent connections or not. */
            if (req.connectionType == HTTP_CONNTYPE_KEEPALIVE)
                keep_alive = 1;
            else
                keep_alive = 0;

            /* Call a handler function, to handle the request. */
            ressourceHandler = lookup(serviceHandlers,
                req.requestedRessource);
            if (ressourceHandler != NULL)
                ressourceHandler->handlerFunction(&req);
        }
        /* Close connection. */
        close(client_sockets[worker_id]);
    }
```

Figure 4.2: The worker_program() function which is responsible for invoking the appropriate service handler to process a request.

### 4.1.3 HTTP Parser

A HTTP parser has been implemented, that supports a subset of the HTTP 1.1 specification. This subset supports the “POST” and “GET” request methods, and understands the HTTP headers “Content-Type”, “Content-Length”, and “Connection”. Any other headers are ignored by the parser. The interface to the parser contains three functions, illustrated in Figure 4.3.

The first parameter of each function is a pointer to a `http_request_t` struct, which contains information about the HTTP request being parsed. Part of this struct is depicted in Figure 4.4. The `request_method` and `requested_ressource` members of
The struct contains the request method, of the HTTP request, and the URI of the requested resource (e.g. “/ExampleService”). The content_length, connection_type, and content_type members of the struct represents the HTTP headers with similar names. Furthermore, the struct contains some members, containing the state of the HTTP request, such as a socket descriptor to the connection the request is sending its data on. However, these members are omitted in the figure for the sake of clarity.

```c
typedef struct {
    int request_method;
    char* requested_ressource;
    int content_length;
    int connection_type;
    char* content_type;
    ... /* Other members, left out for the sake of clarity */
} http_request_t;
```

The functions described in Figure 4.3 are described below. All these functions returns a status code to indicate whether the operation was successful or not.

- The `initialize_request(http_request_t* request, int socket)` function initializes the members of the request struct. The socket parameter, contains a socket descriptor identifying the connection the request is being transmitted on. After calling this function, all the members of the request struct has been set to null values, except for the socket member, which has been set to the value of the socket parameter supplied to the function.

- The `http_read_request(http_request_t* request)` function reads the HTTP header of a request, by reading data from the socket identified by the socket member of the request parameter. After calling this function, the request_method, content_length, connection_type, requested_ressource and content_type members of the request parameter is set to values supplied in the request.

- The `http_read_chars(http_request_t* request, char* dest_buf, unsigned int length)` function reads an amount of bytes, specified by the length parameter, of the HTTP body part of the request. This is copied to the buffer that the dest_buf parameter points to.

The structure of the HTTP parser allows for processing the HTTP body before it is entirely received. This means that the HTTP body will not be stored in its entire length in memory, but processed in smaller parts.

### 4.1.4 XML Parser

The implementation of the XML parser is based on the open source XML parser “Expat” [12]. Expat is a widely used stream-oriented low-level call-back XML parsing library.
written in C. An existing implementation of an XML parser has been chosen, in order to focus on other parts of the implementation. The Expat library consists of several functions that are used to initialize the parser, assign handler functions, manage memory, and handle errors. The only function that will be described in-depth is the `XML_Parse()` function. This function parses a part of a XML document and invokes the appropriate handler functions. `XML_Parse()` takes four arguments as illustrated in Figure 4.5.

```c
XML_Parse(XML_Parser p, const char *s, int s_length, int is_final);
```

**Figure 4.5:** Expat’s `XML_Parse()` function.

The first argument, `XML_Parser p`, is a data structure representing the instantiation of an Expat XML parser. Expat provides the `XML_ParserCreateNS()` to create a XML parser, which understands XML Namespaces. The second argument, `const char *s`, contains a pointer to a piece of XML that is to be parsed. The third argument, `s_length` is the length of the data specified in the `s` argument. The `int is_final` argument specifies whether the data specified in the `s` parameter is the last part of the XML document being parsed.

Figure 4.6 illustrates how a HTTP request can be parsed in multiple steps. In the example, `xml_buff` is a fixed size buffer, used to hold the current part of the HTTP request being processed. A piece of the request is read using the `http_read_chars()` function described in Section 4.1.3. It should be noted that since the `xml_buff` buffer is statically allocated, the process of reading and parsing parts of the HTTP request, avoids unnecessary dynamic allocation of memory, by using a fixed amount of memory. This eliminates dynamic memory allocation and saves memory as the XML document only is buffered completely if its size is less than the size of `xml_buff`. The `XML_Parse()` function calls handler functions as soon as it detects an element, character data or a namespace declaration in the XML it is parsing. However these handler functions are not illustrated in the example. These are described in Section 4.2.1. If an syntax error occurs, the parsing stops and a flag is set to indicate that an error occurred. The handling of such errors is not illustrated in the example. What would happen, is that the rest of the request is read, without being parsed, and then a error message is generated, which is send back to the client.

```c
1  do {
   /* xml_buff_size and done is assumed to have been calculated. */
   http_read_chars(request, xml_buff, xml_buff_size);

5   /* The data read from the request is passed on to the XML parser. */
   if (! XML_Parse(p, xml_buff, xml_buff_size, done)) {
      /* Something went wrong. Handle error. */
      error = 1;
   }
10  } while (error != 1 && !done);
```

**Figure 4.6:** Streaming data into `XML_Parse()`. 
4.1.5 HTTP Reply Generator

The generation and sending of a response, is the last step in the processing of a HTTP request. General functionality has been implemented, that supports generating HTTP responses. Part of the interface to the HTTP reply generator, is illustrated in Figure 4.7.

The soapResponse200 function generates a HTTP response, with the status set to 200, indicating that the processing of a request was successful. As input, the function takes content_length as a parameter. This indicates the content-length of the body that will be sent as part of the response. Furthermore it takes the parameter header, which holds a pointer to where the generated HTTP header should be written to. Finally it takes the parameter http_header_length, which holds a pointer where to write the length of the generated header. Similar functions has been implemented to handle replies with other status codes.

```c
int soap_response_200(int content_length, char **header, int *http_header_length);
```

Figure 4.7: Part of the HTTP reply generator’s interface.

4.2 Service Specific Code

As described in Section 3.3.4, part of the functionality is specific to the generated target application, namely the service handler and SOAP message generation. The implementation of these are discussed in the following.

4.2.1 Service Handler

The service handler described in the design in Section 3.3.4 is implemented as the handler functions that the Expat XML parser invokes. These functions are responsible for processing the SOAP message. There are three different handler functions which are used in the implementation. Each of these functions takes the parameter ad (short for application data), which is a data structure used to maintain the state of the parsing process, as described in Section 3.3.4. The application data in the implementation is a C struct containing several elements that are used to control the state, hold character data, and to maintain information about potential errors that occurred, among others. The handler functions are described below:

- **start_hndl(void *ad, const char *element_name, const char **attributes):** This function is called when a XML start tag is encountered. The parameter element_name holds the name of the XML element that the start tag delimits. The attributes parameter, is an array containing the names and the values of the attributes of the XML element.

- **end_hndl(void *application_data, const char *element_name):** This function is called when an end tag is encountered. The element_name parameter holds the name of the XML element that the end tag delimits.

- **char_hndl(void *ad, const char *data, int data_len):** This function is called whenever character data is encountered. The char_hndl() function may be called sequentially, if that character data of the current XML element, is larger than the buffer Expat is currently parsing. The function takes the parameter data, which
Implementation

holds the character data itself. Furthermore, it takes the length parameter, which holds the length of the character data.

Example 4.1 illustrates part of a start_hndl function. Initially when this function is invoked by Expat, it checks if an error has previously been encountered during the parsing process (line 5). If this is the case, no further processing is done. Next it is decided which state the parser is currently in (line 7). In this example it is assumed that the parser is in the SOAP_DOC_STATE state (line 8). This state indicates that the parser is at the beginning of the SOAP document. In this case, it is expected that the next start tag encountered is the “Envelope” start tag. If this is the case, the state is changed to SOAP_ENV_STATE, and processing ends (lines 12-18). If this is not the case, an error flag and an error message is set, and processing stops (lines 19-25).

```
void start_hndl(void *ad, const char *element_name, const char **attr)
{
    /* If an error has not been encountered so far in the parsing, */
    /* the processing of this start element can begin. */
    if (ad->errorRaised == 0) {
        /* Based on the current state, decide what to do. */
        switch (ad->currentState) {
            case SOAP_DOC_STATE: /* Document just started state */
                /* The next expected element, is the Envelop element, */
                /* which belongs to the namespace identified by the URI */
                /* "http://www.w3.org/2003/05/soap-envelope" */
                if (strcmp(element_name, 
                        "http://www.w3.org/2003/05/soap-envelope|Envelope") == 0
                    && ad->visitedStates[SOAP_DOC_STATE] == 0) {
                    /* The new state of the parser is set */
                    ad->visitedStates[SOAP_ENV_STATE] = 1;
                    ad->currentState = SOAP_ENV_STATE;
                }
                else {
                    /* Some error occurred, and an error flag is set */
                    ad->errorRaised = 1;
                    /* Set the appropriate error message */
                    (...) break;
                }
            /* Handle other states */
            (...)
        }
    }
}
```

Example 4.1: Part of a start handler function of a Web service

4.2.2 SOAP Message Generator

The implementation of the SOAP message generator described in the design in Section 3.3.4, takes advantage of the fact that much of the structure of a SOAP message, is similar for each generated message. Example 4.2 illustrates how the result of invoking a C function is marshalled.
4.2 Service Specific Code

Example 4.2: Marshalling the output of the call_func function.
The call_func function in line 17 is the C function which has been requested to be invoked, and of which the output is marshalled. It takes two in/out parameters. The first is an array, while the second is an integer specifying the size of that array. Lines 20-33 marshals the response. The for loop starting on line 24 marshals the integer array, by putting <item> and </item> around each element of the array. An example displaying a marshalled array is displayed in Example 4.3.

```
<ret>
  <item>76</item>
  <item>2</item>
  <item>9</item>
  <item>42</item>
  <item>45</item>
</ret>
```

Example 4.3: A marshalled array (formatted for easier reading).

The write_response() function used in the example is responsible for sending the SOAP response to the client. Its behavior depends on the connection type of the request, as HTTP 1.1 requires the “Content-length” header to be present for persistent connections, because the client needs to know where the response ends. Consequently, the whole generated response needs to be kept in memory, and dynamic memory allocation is needed, as there is no way of telling how large the response will be before it is generated. If non-persistent connections are used, write_response sends the data in smaller pieces, and static allocation is used.

4.3 Summary

This chapter has discussed the details of an implementation of a prototype middleware application, which is based on the design presented in Chapter 3. The application has been developed for the embedded operating system eCos, which provides all the necessary functionality to build threaded and networked embedded applications, using the C standard library. Details on the implementation of the HTTP parser, XML parser and HTTP reply generator have been covered. These parts are common for all generated middleware applications. Additionally, details of service specific parts of the applications have been given. This includes handler functions called by the XML parser as well as the method used for generating SOAP response messages.

The prototype implementation is used for experiments revealing its performance characteristics.
Performance Evaluation

This chapter describes the experiments that have been carried out to measure the performance of the prototype implementation of the generated middleware code. Based on the results of these experiments, an evaluation of the performance characteristics of the prototype implementation is given. To evaluate the results of the experiments, a set of success criteria are defined.

5.1 Success Criteria

As experiments are performed solely on the implemented prototype, and not some other implementation or middleware technology, the results of the experiments should be compared to results obtained from other sources. The results are compared to those presented in two different papers.

The first paper, by Elfwing et al. [9], compares the performance of SOAP and CORBA, by performing tests based on an industrial billing application using either SOAP or CORBA as the middleware technology. The paper concludes that SOAP will probably always be slower than a similar CORBA implementation. The paper presents a theoretical lower bound for the round-trip time of SOAP request/response pairs to a test Web service, which is found to be 11 ms. The tested response time for the CORBA implementation is 1.6 ms. Since the industrial billing application is used in both cases, this application can take up no more than 1.6 ms. of processing time. The remaining 9.4 ms used in the SOAP implementation must be caused by processing SOAP messages and transferring data over the network, and not by executing the application. These results are used when evaluating the performance of the prototype implementation.

The second paper, by Govindaraju et al. [10], evaluates various RPC protocols. Performance tests are carried out using several different hardware platforms. These results are also used when evaluating the performance of the prototype implementation.

Other success criteria of the prototype implementation should also be fulfilled. The performance experiments makes use of messages containing an array of 32 bit integers with a varying number of elements. Parsing such SOAP request messages and generating SOAP response messages should have a time complexity of $O(n)$ where $n$ denotes the size of the message, in this case with respect to the number of elements in the array. Whether
this is fulfilled is not formally shown, but argued for by the results of the experiments. Additionally, the memory consumption should be reasonable when compared to the sizes of the incoming requests and the generated replies.

5.2 Test Design

This section describes the tests that are performed on the implementation. The performance of the service application itself should not affect the result of the experiments. For this reason, the application used in the experiments should be very simple, and should not take up a significant amount of time compared to the time used for processing SOAP messages.

The target application provides a function which takes an array of integers of arbitrary size as the only argument. This function is made available as a Web service. Each array element must be an integer. The application then multiplies each element of this array by 4, returning the result to the SOAP skeleton which generates a SOAP response message, and sends it to the client. The complexity of performing this task should not have a major influence on the results of the experiments. For that reason, this application has been chosen, because it requires very simple calculations.

The performance of clients should not affect the results of the experiments either. For that reason, the clients do not perform any kind of processing of requests or responses. The client simply measures the elapsed time from sending a request until the response has been fully received.

The following experiments are carried out:

- **Profiling:** This experiment measures the distribution of the time spent in various parts of the code. This gives an overview of which parts of the processing of a SOAP message take up the most time. This may indicate which parts of the code should be optimized further.

- **Continuous request speed:** In this experiment, a client continuously sends a request to the server using a persistent HTTP/1.1 connection. The average time taken for each request/response pair is measured. This experiment is repeated for different request sizes.

- **Streamed reply speed:** This experiment measures the average time of performing a single request, while streaming the reply or not. Each request involves connection to the service, sending the request, receiving the response, and finally, closing the connection. The size of the request is varied to measure the impact on the request speed.

- **Memory consumption:** The amount of memory allocated by the target application is measured while varying the size of the request. This experiment is carried out in two different settings. The first makes use of streaming of the SOAP response (as made possible by using a non-persistent connection). In the second setting no streaming of the SOAP response is performed (as necessary when using a persistent HTTP/1.1 connection).

- **Code size:** The size of the target application code is measured. This size includes the size of eCos as well as the size of the middleware application. The size of the application before linking is also measured.
5.3 Test Environment

The prototype implementation of the target application is tested in a closed environment. The target hardware platform is an Intel Pentium 133 MHz processor with 128 MB RAM, equipped with an Intel 82559 100 Mbit/s Ethernet controller. This machine is running a development version of eCos from December 2nd 2003. The development version is chosen ahead of the version 2.0 release, because that release includes memory leak problem in the TCP/IP stack implementation. eCos is compiled with the configuration described in Section 4.1.1. The RedBoot bootstrap environment is used to load the eCos operating system and application onto the target machine.

A separate machine is used as a client of the Web service provided by the target machine. This machine is equipped with an Athlon XP1800+ (1533 MHz) processor, 512 MB RAM, and a 3Com 905C-TX 100 Mbit/s Ethernet controller and is running Linux 2.4.22.

The Web service client machine is connected to the target machine by ethernet, using a crossed twisted-pair cable. This avoids any traffic created by other hosts on other networks, thereby minimizing noise on the test network and giving more consistent test results. The connection is configured for 100 Mbit/s full duplex communication.

5.4 Test Results

This section covers the results of the experiments that have been carried out. Each of the experiments correspond to one of those listed in Section 5.2.

The messages used in these experiments contain an array of 32 bit integers. A marshalled message with one array element has a size of 437 bytes. Each additional array element takes up 23 bytes. This is valid under the assumption that the numbers in the array can all be represented by a single digit, which is the case in the experiments. Consequently, the smallest message in this experiment, having 50 array elements, take up 1.565 bytes. The largest message, with 10,000 elements, takes up 230.417 bytes.

5.4.1 Profiling

Figure 5.1 shows the results of the profiling test. This test was performed by collecting timing statistics in the prototype implementation, while it was handling requests. The size of the request was varied. Four parts of the code was measured. The time spent on waiting for and retrieving data from the client (Receive), the time taken to parse the request (Parse), the time taken to generate the response (Generate), and the time taken to send the response to the client (Send).

It is seen that most of the time is spent on parsing the request. This is not surprising, since both XML parsing and unmarshalling is performed in this step, and since these tasks are relatively complex. The time taken on each step rises approximately linearly as the size of the request is increased. The only exception is the receive time. When requests are relatively small, this time fluctuates, which is concluded to be caused by delays in the network communication. It has not been further established why these delays occur.

5.4.2 Continuous Request Speed

Figure 5.2 shows the results of the continuous request speed test for arrays containing between 5 and 8.800 elements. In this experiment, a persistent connection to the server was established, on which 1,000 identical requests/responses were completed. The elapsed
time was measured by the client application. The average time for each request was calculated and used in these results. This test was repeated for varying request sizes.

The test reveals the tendency, that the response time increases linearly with the size of the request. This is consistent with the results in Figure 5.1. There are however some fluctuations in these results. As these fluctuations do not occur in the profiling test, it is concluded to be caused by delays in the network communication.

Figure 5.3 shows the same test results for array sizes between 5 and 150. In this graph it is seen that small requests can be completed within a few milliseconds. The request containing 5 array elements is completed in just 3.6 ms.

5.4.3 Streamed Reply Speed

Figure 5.4 shows the results of the streamed reply speed experiment. The size of the request was varied, and for each request size, the time taken to complete a single request
5.4 Test Results

Figure 5.3: Average response time as a function of the request size, using persistent connections, when the requests are small.

was measured, both using streamed replies or not. Streamed replies are used when the connection is of the type “Close”, while streaming is not used for “Keep-Alive” connections. For each request size, this test was performed 10 times. The fastest and the slowest results were disregarded, and the average of the remaining results was calculated.

Figure 5.4: Response times when streaming replies and when not, as a function of the request size.

Although the results fluctuate somewhat, there is a general tendency that the response times are lower when the responses are streamed to the client. This may be caused by the fact, that when not streaming, a larger amount of memory must be allocated dynamically to hold the entire response in memory. Allocating memory dynamically usually involves system calls to be performed, and a free block of memory must be found and allocated. As a consequence, dynamic memory allocations are considered expensive in terms of CPU usage, and should be avoided when necessary.
5.4.4 Memory Consumption

Figure 5.5 shows the maximum amount of dynamically allocated memory at any time as a function of the size of the request. This amount of memory is allocated for each SOAP processing thread when processing a SOAP request. No method was found to measure the amount of allocated memory under eCos, so this experiment was performed in an equivalent single-threaded application running on Linux. The memory consumption was measured using the memory profiler “memprof” [20]. One experiment was performed for “Keep-Alive” connections as well as “Close” connections.

![Figure 5.5: Dynamically allocated memory when streaming and not streaming, as a function of the request size.](image)

The results clearly show, that less memory is allocated for “Close” connections than for “Keep-Alive” connections. This is due to the fact that “Close” connections can be handled in such a way, that the response is sent in chunks to the client. Only a fixed size buffer needs to be allocated to hold this chunk. When serving a “Keep-Alive” connection, the entire response message needs to be stored in memory before it can be transmitted, in order to calculate its length. This length must be sent with the message in the “Content-Length” HTTP header.

When performing application execution, some amount of memory must be allocated to hold the result of the function call. The size of this result is not always known in advance, so dynamic memory allocation can not be avoided.

5.4.5 Code Size

The size of the binary data representing the program was measured. This size was measured using the RedBoot bootstrap environment. The size of the combined eCos and application binary was measured to 277.344 bytes. The middleware application contributed to this size by 118.516 bytes.

5.5 Summary and Discussion

When comparing the results with those presented by Elfwing et al. [9], the results of the experiments satisfy the success criteria. The average time taken to complete a request on a persistent connection is 3.6 ms. for the smallest array size tested (5 elements). Elfwing
et al. presents a theoretical lower bound of 9.6 ms. when applying all the optimizations they suggest, however the request size used in their experiments is not known. To reach a response time of 9.6 ms. using the prototype implementation, the message must contain an array with approximately 85 elements in our implementation, as seen in Figure 5.3. It should be noticed that the experiments performed by Elfwing et al. are carried out on superior hardware than that used when testing the prototype implementation presented in this report. Elfwing et al. used a Pentium II 400 MHz machine with 192 MB RAM.

The results presented by Govindaraju et al. [10] are interesting as well. The paper compares the efficiency of various middleware protocols. One of the types of messages which are used for performance experiments is SOAP messages containing an array of numbers. The size of this array is varied, and the round-trip time for a single request and response is measured. This test data is similar to that used for the experiments in this report. Govindaraju et al. are able to handle requests with 50 array elements in 28 ms, while our implementation can complete the same task in 11 ms. Govindaraju et al. handle 1,000 array elements in 340 ms, while our implementation completes in just 58 ms. The experiments performed by Govindaraju et al. used two machines containing a Pentium III 533 MHz processor and 256 MB RAM, running the Linux operating system. These two machines were connected by a 100 Mbit/s LAN. Notice that this hardware is significantly faster than that used for the experiments in this report.

The prototype is in general faster than the implementations used in the two papers. Additionally, it runs on slower hardware. This indicates that the prototype implementation has satisfying performance properties.

The memory consumed by the prototype implementation is linearly proportional to the size of the request. A request with 10,000 array elements, having a SOAP representation of the size 225 kB, can be handled using only 50 kB of dynamically allocated memory when streaming the response. This is due to the fact, that only portions of the request is resident at any time, and that only a portion of the response is resident at any time. The extra memory which needs to be allocated dynamically is used for storing application data, i.e. the data returned by the application running on the target machine.

The size of the code was measured to 118,516 bytes. It may be possible to reduce this size in several ways. One possibility is to not use the expat XML parsing library. Expat contains functionality that is not needed, and although it is linked statically with the application, some unneeded functionality may be included in the resulting binary. A specialized XML parser could be implemented, which only includes the needed functionality. Another possibility is to change the architecture of the generated code, such that is makes extensive use of encapsulation of code into functions. This would probably have the side effect of worsening the performance of the implementation.
Conclusion and Future Work

This report is the result of an investigation of the applicability of middleware as a means to integrate distributed embedded systems with other computing systems in a loosely coupled environment.

A study of middleware was performed, and shortcomings of middleware were identified, when performing application integration between decentralized systems. It is the conclusion of these studies that the Web service technology provides a way to address these shortcomings, by providing a stateless message exchange paradigm in which structured and typed information can be exchanged between peers in a decentralized distributed environment. Based on this conclusion, it was decided to investigate the applicability of Web services as a middleware technology for loosely coupled integration between distributed embedded systems and other computing systems.

6.1 Results

A key aspect in distributed embedded systems is performance. In order to demonstrate the applicability of Web services in distributed embedded systems, the primary aim of the work presented in this report, has been to show that Web services can be implemented on embedded systems with a minimum of resource consumption, such as CPU utilization and memory consumption.

The main contribution of this report is a partial design of a framework for developing Web services for distributed embedded systems. The framework enables programmers to make Web services available by automatically generating the necessary middleware in a transparent way for the programmer. In order to do this, an algorithm has been devised for mapping a C representation of data to the message format that Web services is based on. Furthermore, an architecture for the middleware code that the framework generates was designed. This architecture has proven to be scalable in relation to the size of messages that are exchanged. The architecture allows for processing messages in small parts at the time, as they arrive from the network, thereby avoiding to have an entire message represented in memory at the time.

Furthermore, this report contributes with a prototype implementation of an instance of the middleware code generated by the framework. This implementation runs on the
embedded operating system eCos. The implementation illustrates that the concepts introduced by the design are applicable in practice, and that it is possible to implement Web services for distributed embedded systems with a small resource consumption.

Finally, this report contributes with performance evaluations of the implementation, establishing measurements of the resource consumption of the generated middleware. The results were compared to those achieved by Elfving et al. [9] and by Govindaraju et al. [10]. This comparison reveals considerably faster processing than that presented by these authors. Some key measurements from the performance evaluation is the ability to process a request to a Web services containing an array of 10,000 integers, using 50 kB of dynamically allocated memory. The text representation of such a request, dictated by the underlying SOAP protocol, has a size of 225 kB. The code size of the implemented instance of generated middleware takes up 116 kB. This gives a total code size of 270 kB when eCos is included. Finally, the measured round-trip time of processing a request to a Web service containing an array of 5 integers was 3.6 ms on an Intel Pentium 133 MHz processor with 128 MB RAM, transmitted over a 100 Mbit/s LAN.

It is concluded, that the efficiency of the prototype implementation is acceptable in terms of processing time of request-response message exchanges, and in terms of the memory consumption. The performance evaluations revealed that the techniques developed to minimize CPU processing time and memory consumption were successful and that a linear relationship between the resource consumption and the size of the messages exchanged was achieved. Based on these results, it is concluded that Web services is applicable as a middleware technology for loosely coupled integration between distributed embedded systems and other computing systems. However, in order for this technology to be applicable in practice, it is important that it becomes widely adopted among service providers. Web services has already been applied, e.g. the Globus Alliance team has chosen Web services as the basis for the Globus Toolkit [28], a tool used for building computational grid systems. Furthermore, it is important that the technology matures and the specification of the architecture becomes stable. An important component of the Web service architecture, namely the Web Services Description Language, used to describe the protocol for communicating with a Web service, has yet to reach official W3C recommendation status.

## 6.2 Future Work

The focus of the design in this report has been to provide a mapping of C data types to their equivalent Web service representation, and to design the architecture of the generated middleware code. However, algorithms for generating the middleware code has not been devised. In order to show that it is practical to generate Web service middleware, design and implementation of such algorithms is called for. Furthermore, the designed framework only addresses the representation of a subset of C data types. A more detailed study of mapping C data types to a Web service representation may also be of interest. Also, as many embedded systems also are programmed in C++, it could be of interest to study the mapping to such a language. Finally, further optimizations of the implementation itself could be performed. As the performance evaluations showed, significantly smaller amounts of dynamic memory allocation is needed when streaming the response of processing a request to a Web service. In the current implementation, when using persistent HTTP/1.1 connections, the entire response message is generated at one time, in order to calculate the length of the response, as this length must be sent as part of the HTTP/1.1 response. If an algorithm could be devised to calculate this length without having to generate the entire message, the advantages of using persistent connection could be combined with the
advantages of streaming the response.

The main aim of this project was to show that Web services can be implemented on embedded systems with a minimum of resource consumption, such as CPU utilization and memory consumption. However, bandwidth consumption when transmitting SOAP messages has not been addressed. The performance evaluation has shown that encoding an array of 10,000 integers requires a SOAP representation of the size 225 kB. Such a representation may not be desirable in some distributed embedded systems. Formally, a SOAP message is specified as an XML information set [39], and is typically represented as an XML document. However, it is possible to represent a SOAP message in a different manner. Providing a different encoding of a SOAP message could potentially result in smaller messages, thereby reducing bandwidth consumption.

Finally, in order for Web services to become widely accepted, it must be possible to communicate securely. This is an important issue which has not been addressed in this report. Security in Web services has yet to be standardized by the W3C. It could be of interest to investigate how the overhead of adding security to the Web service architecture affects its performance.
**Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture.</td>
</tr>
<tr>
<td>DE</td>
<td>Distributed Embedded system.</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model.</td>
</tr>
<tr>
<td>DRE</td>
<td>Distributed Real-time Embedded system.</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition.</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol.</td>
</tr>
<tr>
<td>MEP</td>
<td>Message Exchange Pattern.</td>
</tr>
<tr>
<td>RE</td>
<td>Real-time Embedded systems.</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation.</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call.</td>
</tr>
<tr>
<td>SAX</td>
<td>Simple API for XML.</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol.</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Description, Discovery, and Integration.</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier.</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator.</td>
</tr>
<tr>
<td>URN</td>
<td>Uniform Resource Name.</td>
</tr>
<tr>
<td>UTF</td>
<td>UCS Transformation Format.</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language.</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language.</td>
</tr>
</tbody>
</table>
Conclusion and Future Work
Bibliography


