Faculty of Engineering and Science
Department of Computer Science

Title:
Producing Efficient Web Services for Distributed Embedded Systems

Project period:
DAT6 / F10S,
Feb 2nd 2004 - Jun 11th 2004

Project group:
B1-215

Group members:
Michael Sig Birkmose
Lars Haugaard Kristensen

Supervisor:
Mikkel Christiansen

Publications: 7
Pages: 104

Abstract:

This thesis studies how Web services can be applied efficiently in distributed embedded systems. The study is approached by developing a tool, which we name WSTOOL, for generating Web service middleware in the C programming language.
An analysis of the differences in the semantics offered by C and Web services is performed, leading to a mapping between C and SOAP. The developed mapping enables a high level of transparency when developing Web services in C. After discussing the overall architecture of WSTOOL and the middleware generated by the tool, we define interfaces enabling the use of alternative network representations of SOAP messages. Additionally, a design is provided which exploits service specific knowledge to generate efficient middleware.
Based on a performance evaluation it is concluded that the techniques used in the design and implementation of WSTOOL does help generate efficient middleware, when compared to alternative tools.
The main contribution of this thesis lies in the design of WSTOOL and a proof of concept implementation of this tool.

Summary in English

This thesis is the result of our study of the applicability of Web services in distributed embedded system. We study how the overhead associated with the abstractions provided by Web services can be minimized. The extra overhead is caused by the extra size of the messages that must be transferred, when compared to the actual size of the data carried within such messages. This includes abstractions on the underlying network protocols and serializations of the data.

In this thesis we approach this problem by developing a prototype implementation of a tool, which we name WSTOOL, for generating Web services. This tool is used to generating Web services (service providers) as well as clients of Web services (service requesters). The generated middleware is tailored to the specific service, for which it will be used. This enables WSTOOL to employ algorithms that specialize the generated source code to include only the necessary functionality for that specific service. The middleware source code generated by WSTOOL also uses special memory allocation strategies to allow it to allocate a minimum amount of memory, using a minimum number of dynamic memory allocation calls.

The design of WSTOOL builds on our prior work presented in DAT5 [2], in which a basic design of a framework for developing efficient Web services in the C programming language for distributed embedded systems was presented. This design is augmented in this thesis. A mapping between the type systems of C and SOAP is necessary to give application developers a transparent interface in service requesters to the remote procedure call facilities of SOAP, and to allow simple development of service providers. A symmetric mapping between C and SOAP is developed in this thesis. The focus is on supporting an almost complete mapping from C to SOAP. Furthermore, a symmetric mapping between SOAP and C is defined, meaning that any types that are mapped from C to SOAP also are mapped from SOAP to C.

A central overhead introduced with Web services is in the size of the SOAP messages that are exchanged. These are represented as XML documents, which introduces a significant overhead. For that reason, an efficient interface is developed, which enables the use of other representations of SOAP messages than XML. This interface is based on the fact that the structure of SOAP messages is formally defined by an XML information set, which allows alternative serializations. The interface we have defined is based on the exchange of a limited set of simple tokens. Again, the service specific knowledge can be used, since this knowledge makes it possible to build a list of all possible tokens that will appear in valid messages. This enables the interface to be efficient, since tokens can simply be represented as constant numbers.

In DAT5 [2], an instance of a Web service service provider was implemented. When parsing incoming messages, the implementation of the service was fixed to accepting only a single type of messages. To formalize the generation of such message parsers, a design for generating context-free grammars which describes the syntax of a specific messages is presented. Using this structure, it is simple to generate a parser specification, which is tailored to supporting a specific service. This parser specification can then be used to

generate an actual parser implementation. In the implementation of WSTOOL, the Bison parser generator is used.

A special semantic stack is developed, to be used in the semantic actions of generated parsers. Using this specialized semantic stack, the amount of copying in memory can be reduced dramatically, allowing for a more efficient parser compared to using the built-in semantic stack supplied by typical parser generators. This reduces both the amount of memory required as well as the execution time of the parser.

An interface for developing applications using the middleware generated by WSTOOL is defined, which enables a developer of a service requester to implement the invocation of a remote procedure as a call to an associated local C function. Parameters to the remote procedure are given as parameters to the local C function. Implementing service providers involves specifying the set of functions in an existing application which must be exposed as part of a Web service, using a C header file which obeys some certain conventions.

Since the interface for remote procedure calls in Web services can not be completely transparent, there are details that the developer must be aware of. The invocation semantics of a remote procedure call is not equal to that of a local function call. For local function calls, exactly-once invocation semantics is achieved. For remote procedure calls however, the invocation semantics are dependant on the semantics of the underlying communication protocols. Given the set of requirements we present for our network abstraction layer, an at-most-once semantics is achieved. The developer must take this into account when performing remote procedure calls.

A prototype implementation of WSTOOL has been completed, using the C++ programming language. Using WSTOOL, an example Web service is developed, which is used as a basis for performance experiments. The same Web service is implemented using two alternative tools, gSOAP and CSOAP. A comparative performance evaluation between the three Web service solutions is completed. From this evaluation it is concluded that the middleware generated by CSOAP is clearly the slowest and most memory consuming. This indicates that the per service generating strategy of WSTOOL and gSOAP is preferable, when compared to the generalized approach of CSOAP.

From the performance evaluation it is also concluded that WSTOOL marshalling performance is superior to that of gSOAP for large requests. The unmarshalling performance of WSTOOL is not quite as good as that of gSOAP, however a number of suggestions for improvements are presented, which should improve the unmarshalling performance.

Finally, the performance evaluation shows that the required amount of dynamically allocated memory is constant for a specific service, also when varying the size of the messages transferred. This is made possible by the streaming architecture of the generated middleware. The amount of stack memory required depends on the size of the data structures that must be handled.

As a final conclusion on our studies of the applicability of Web services in distributed embedded systems, we conclude that Web services is an applicable middleware technology in distributed embedded systems. We consider our implementation of WSTOOL as a proof of concept, which has illustrated that the overhead associated with Web services can be reduced.

Preface

This thesis continues our work presented in the report "Applying Web Services as Middleware for Integrating Embedded Systems in Loosely Coupled Environments" [2]. For the remainder of this thesis, that report will be referred to as DAT5 [2]. In the report, the applicability of Web services as middleware in distributed embedded systems was investigated. The main contribution of the report was the design of the architecture of a framework for developing Web services in the C programming language for the embedded operating system eCos [5], and a prototype implementation of the middleware code generated by this framework. Based on performance evaluations of the prototype implementation, the conclusion of the report was that, in terms of memory and processing resources, it is feasible to apply Web services in distributed embedded systems. However, DAT5 [2] only presented an overall architecture of the framework for developing Web services for distributed embedded systems in the C programming language. Many limitations were put on the functionality of the framework to simplify its design. Such limitations are not desirable in many practical applications. Additionally, only a design for the generated middleware code was provided, which left out many details. Furthermore, no algorithms for generating this code were provided.

In this master's thesis, we further develop the study of the applicability of Web services in distributed embedded system by extending the design of the framework presented in DAT5 [2]. The purpose of this extended design is to remove the limitations of the original design, and to provide algorithms for generating the middleware code. As a proof of concept, the framework is implemented as a tool called WSTOOL. The processing and memory requirements of the middleware generated by WSTOOL is determined.

To study a concrete case in which Web services are interesting to apply, and to establish a concrete case for testing, a cooperation with the company Aeromark has been established. Aeromark specializes in complete mobile voice and data solutions. Aeromark's next generation product platform, named Gefion, is a GSM, GPS, and GPRS enabled embedded device, enabling a range of applications from basic vehicle tracking to comprehensive task and service level management. Communication in Gefion's predecessor is based on the exchange of SMS messages over the GSM network. With the introduction of GPRS to the Gefion platform, a new range of applications are enabled due to the increased bandwidth and lower cost of communication. One example of such an application is a road direction service, where the Gefion device contacts a service provider with a request for a travel description to a given address.

In order to handle the complexity of communication in such distributed applications, and to simplify the development process, Aeromark is interested in achieving a higher level of abstraction in their development process, by applying middleware. Specifically, Aeromark is interested in investigating the applicability of Web services as middleware in their embedded products.

As presented in DAT5 [2], one of the problems of applying Web services in distributed embedded systems is that the network representation of the messages that must be transmitted is relatively large when compared to other middleware standards. This results in

increased bandwidth demands. In order to meet this challenge it should be possible to use other network representations than XML documents. To achieve this, a redesign of the code generated by the framework is necessary. This design must accommodate that the encoding of SOAP messages should not be fixed.

The reader of this thesis is expected to be familiar with the work presented in DAT5 [2], since the terminology and concepts presented in that report will only briefly be summarized in the introduction of this thesis.

A central technology referenced in this thesis is the C programming language. Any reference to the C programming language refers to ANSI C, as described by Kernighan and Ritchie [10].

When words written in `typewriter font` appear in the regular text, it signifies that the words represent source code. When keywords are mentioned for the first time, they appear in *italics*.

The source code of WSTOOL is available on the CD-ROM attached at the back of the thesis. To build and install WSTOOL, follow the instructions found in Appendix C.

Michael Sig Birkmose

Lars Haugaard Kristensen

Contents

1 Introduction

In this chapter we introduce the notion of distributed embedded systems and middleware. Our motivation is to introduce key terminology used in distributed embedded systems and middleware, in order to give an understanding of how distributed embedded systems differ from conventional distributed systems, and to give an understanding of the notion of middleware.

Furthermore, we introduce Web services and discuss how it relates to middleware. We discuss how Web services can be applied in the development of distributed embedded systems in order to achieve a higher level of abstraction. The company Aeromark is introduced, whom are interested in applying Web services in the development of their distributed embedded systems. Our motivation for introducing this company is to give a concrete example of an application of Web services in distributed embedded systems, and to establish a case for evaluating our results.

We end this chapter by a discussion of our prior work in DAT5 [2], and by giving a problem statement.

1.1 Embedded Systems Terminology

Embedded systems belongs to a special class of computing systems. In order to give an understanding of the difference between embedded systems and ordinary computing systems, we here summarize the terminology used in embedded system, and give a definition of this term. Furthermore we introduce a special class of embedded systems, namely distributed embedded systems, which are the class of systems that we are concerned with in this thesis.

1.1.1 Embedded Devices and Embedded Systems

More and more appliances such as video recorders, mobile telephones, 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*.
The important properties of an embedded device is that it:

- Includes a programmable computer.
- It has limited resources compared to a modern personal computer, in terms of processing speed, memory size, communication bandwidth, power availability, etc.

This leads to our definition of an embedded device:

Definition 1.1 (Embedded device)
An embedded device is a hardware device containing a programmable computer, which has limited resources available, in terms of communication bandwidth, power availability, processing speed, and memory capacity, when compared to a modern personal computer.

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

The definition of an embedded system adopted in this thesis is based on the following properties 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.
- It solves a dedicated task.

From these properties we adopt the following definition of an embedded system:

Definition 1.2 (Embedded system)
An embedded system is any system consisting of one or more embedded devices, which contains software that interacts with an external environment, such as a human user, physical objects and sensors, in order to solve a dedicated task.

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 a motor controlling an optical unit, an LCD screen, a flash unit, etc, in order to solve a dedicated task, namely photography.

The definition of an embedded system is slightly different from that introduced in DAT5 [2], since it emphasizes that such systems solves a dedicated task. Thereby, systems such as PDAs are no longer part of the definition as they are classified as general purpose computers.

1.1.2 Distributed Embedded Systems

Having defined embedded devices and systems, we now define distributed embedded systems. Distributed embedded systems can be considered as embedded systems consisting of at least one embedded device cooperating with other computing systems, such as other embedded devices or general purpose computers, in order to solve a common task. Therefore, we define such a system in a similar manner to the definition of a distributed system provided by Coulouris et al. [4]:

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 [12]. 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.

1.1.3 Other Types of Embedded Systems

Other types of embedded systems exists, such as embedded systems where real-time properties are an important aspect, since the correctness of the systems not only depends on the logical result of the computations, but also on the time at which results are produced. However, real-time considerations are beyond the scope of this thesis, and we are only concerned with distributed embedded systems, where the time at which results are produced is not essential.

1.2 Middleware and Web Services

When developing distributed embedded systems, many challenges are faced, due to the heterogeneous environments in which they operate. Applications are executing on different hardware platforms and operating systems, and are written in different programming languages. As today's distributed embedded systems are becoming increasingly complex, the need for introducing a higher level of abstraction arises, in order to handle the complexity of communication in such systems, and to cope with the general desire for fast development time and low development cost of such systems. However, as distributed embedded systems have stringent resource requirements, a conflict arises, as introducing an extra level of abstraction is often associated with increased resource requirements. This conflict is a central issue investigated in this thesis.

In this section we introduce the concepts of middleware and Web services. The relation of Web services to middleware is described, and an introduction to the idea behind Web services is given.

1.2.1 Middleware

The notion of middleware [14] provides a method to handle the complexity of developing distributed systems. The basic idea is to:

- Shield software developers from low-level platform details, such as low-level network programming and operating system specific functionality.
- Minimize the software development cost by applying previous development expertise in reusable frameworks.
- Provide a set of high level network-oriented abstractions, that are useful for fulfilling the application requirements.

1.2.2 Web Services

As described by Schantz and Schmidt [14], middleware can be decomposed into multiple layers of abstraction. Web services belong to a class of middleware called *common middleware services* . Common middleware services introduce high level domain independent services to distributed applications. As the name implies, this type of middleware allows application developers to avoid implementing common elements in distributed applications, such as services managing transactions, security, resource discovery, and others. The W3C Web Services Architecture Working Group defines a Web service as [19]:

Definition 1.4 (Web service)
A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Web services is based on the exchange of SOAP [17, 18] messages, using the SOAP protocol, which belong to a class of middleware called *distribution middleware*. Distribution middleware aims to introduce a higher-level distributed programming model, which 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 and functions implemented on different hosts can be invoked (remote procedure call). This is commonly achieved by providing a stub procedure [4] that provides an interface in the host language to invoke the remote procedure. This stub procedure is responsible for marshalling the parameters of the procedure, sending a request, receiving a reply and unmarshal the reply. Similar a skeleton [4] procedure is provided on the host where there remote procedure resides. This procedure has the responsibility of listening for requests for the remote procedure, unmarshal the parameters, invoke the procedure and marshal the result of invoking the procedure.

SOAP is not tied to any specific object model, and allows for exchange of messages in a RPC fashion. Furthermore, SOAP also allows for the exchange of messages in what the W3C calls *document style*, where messages are exchanged as pure XML documents, thereby leaving it up to the interacting applications to decide upon the semantics of the messages.

Web services is a special type of common middleware services, which aims to work in distributed, decentralized environments, allowing information systems developed by different organizations to interact. Conventionally, middleware resides between applications in closely coupled systems, and in such cases a suitable middleware platform is chosen in order to accommodate for the specific communication needs. However, in cross-organizational interactions, a new challenge is faced, since the different organizations have to agree on a middleware platform that allows their different information systems to interact. As described in DAT5 [2], this is the problem that Web services strives to solve. The idea is to use one middleware technology, namely Web services, to handle interactions across the boundaries of different information systems. Web services allows for this by not being tied to a specific object model and by building on well established technologies such as HTTP

and XML. Internally, each organization can use their preferred middleware platform to build their applications, and for communication with external information systems Web services are used.

Figure 1.1 illustrates the basic architecture of a Web service. A Web service is most commonly a client/server architecture consisting of two actors: A service provider and a service requester. A service provider is an application providing services, and a service requester is a application that requests these services. The exchange of requests and responses are performed using the SOAP protocol. SOAP is a stateless, one-way message exchange protocol which is used for exchanging structured and typed information between peers in a Web service architecture.

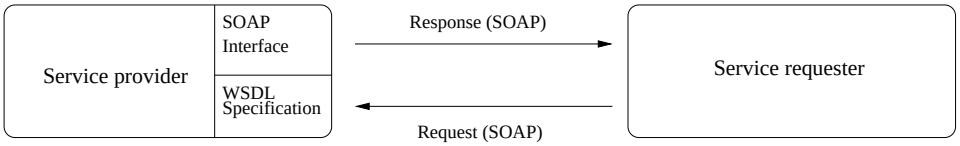


Figure 1.1: The architecture of a Web service.

WSDL [20, 21, 22] is an XML based language for describing the protocol for communicating with a service provider. WSDL describes on an abstract level the messages that are exchanged between a service provider and a service requester.

Other components can be added to the Web service architecture, such as service brokers, transaction managers, etc. However, these are not standardized yet and we do not address them further.

Today, Web services typically find its use in business to business interactions. Many tools such as Sun Microsystems's "Java Web Services Developer Pack" [15] and Microsoft's .NET framework [11] have been developed to support the development of Web services.

1.3 Cooperation with Aeromark

We have established a cooperation with the company Aeromark, which is a company developing distributed embedded systems, specializing in complete mobile voice and data solutions. Our motivation for establishing a cooperation with Aeromark is to establish a concrete case where Web services are interesting to apply, and to establish a concrete case for testing.

1.3.1 The Product

Aeromark develops the product *Orange Fleet Link*, which is a product for companies to manage their vehicle fleets. Figure 1.2 illustrates a typical use-case scenario of the Orange Fleet Link product. Companies install a GSM and GPS enabled embedded device in each vehicle they want to manage, which allows for a range of applications, from basic vehicle tracking to comprehensive task and service level management. Using the GSM network, the different vehicles can be managed. A range of information about each vehicle can be retrieved such as its geographical position, statistics about the vehicle such as average speed, number of incidents of speeding, whether the driver has had his mandatory breaks, etc. All information is routed through the Aeromark data center. By accessing this data center, customers are able to manage their fleet of vehicles. The Internet is typically used when communicating with the data center. Alternative means of access to the data center is also possible, such as using the X.25 protocol over a leased line.

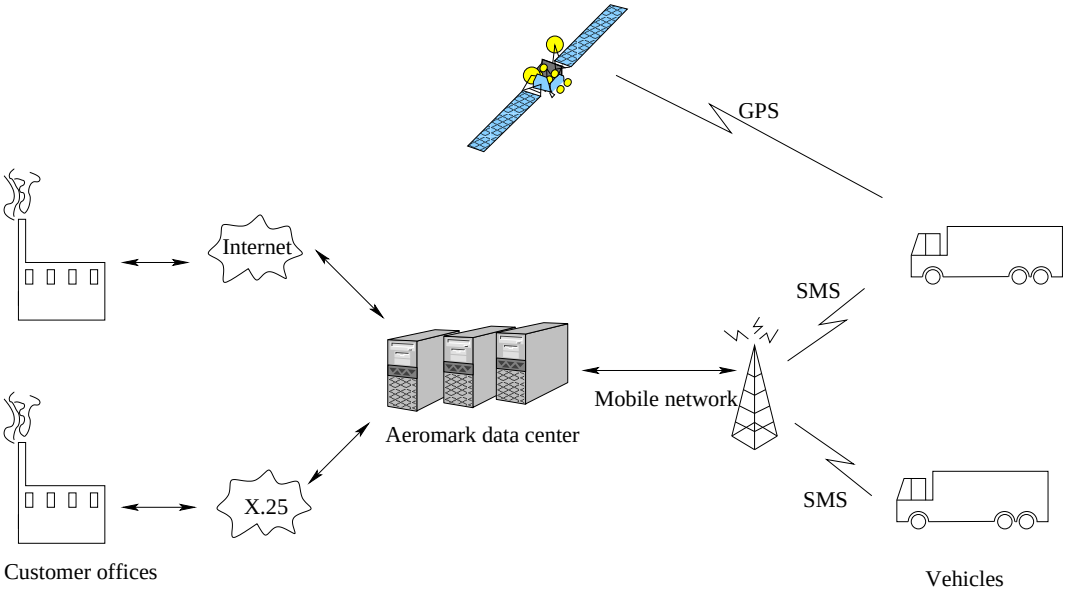


Figure 1.2: Use-case scenario of the *Orange Fleet Link* product.

Aeromark's next generation product named Gefion, is a GSM, GPS, and GPRS enabled embedded device. Communication in Aeromark's current products is based on the exchange of SMS messages over the GSM network. Aeromark has developed and maintains a protocol designed to function on top of the SMS protocol, which handles the communication between the embedded device and the Aeromark data center. With the introduction of GPRS to the Gefion product, a new range of services are enabled, due to the added bandwidth and due to the lower cost of communication. One example of such a service is a road direction service, where the Gefion device requests a travel description to a given address. Another example is specialized services developed on the request of specific customers of Aeromark, such as integration with their existing information systems.

1.3.2 A Need for Abstraction

The introduction of the new services enabled by the new Gefion platform also increases the requirements of continually extending and maintaining the communication protocol, which is costly for Aeromark. Therefore, Aeromark is interested in using a different communication paradigm, in order to simplify the process of developing and maintaining their distributed embedded systems. As their systems operate in a decentralized environment, Web services is an interesting middleware technology for Aeromark to achieve this. For instance, the road direction service described in the previous example could potentially be provided by other organizations than Aeromark. In such a decentralized scenario Web services is a suitable middleware technology. Furthermore, Aeromark produces relatively few units, and their new Gefion platform is powerful in terms of memory and processing resources by today's standards. Therefore, the cost added by the overhead of using middleware is potentially less than what is gained by a simpler process of developing and maintaining the communication protocols. Therefore, Aeromark is interested in applying Web services in their products.

1.4 Prior Work

Our prior work, presented in DAT5 [2], introduced the architecture of a framework for developing Web services in the C programming language, specifically for the embedded operating system eCos. The framework provided two functionalities: Creating service providers in C by generating skeleton code to let service requesters invoke functions of an application written in C, and secondly, creating service requesters by generating stub code in C used to send requests to a service provider. In the following, we briefly describe the overall architecture of the development process described in DAT5 [2].

1.4.1 Creating Service Providers

When creating a service provider, a C header file specifying the functions to be exposed as a Web service is given as input to the framework. The structure of this C header file must respect special conventions and constraints, which aids in describing the functions to be exposed as part of a Web service. The advantage of using a C header file to specify the Web service is that the application developer does not need to understand a separate *interface definition language* such as WSDL in order to be able to specify a Web service.

Figure 1.3 illustrates the process of generating skeleton code for a service provider. 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 resides. The *target* refers to the embedded system, on which the application will be executed. When creating a service provider, the framework has the responsibility of creating skeleton code that takes care of receiving requests, invoking the requested function, and sending responses. This code is linked together with the application. Furthermore, the framework also creates a WSDL document describing the protocol for communicating with the generated Web service.

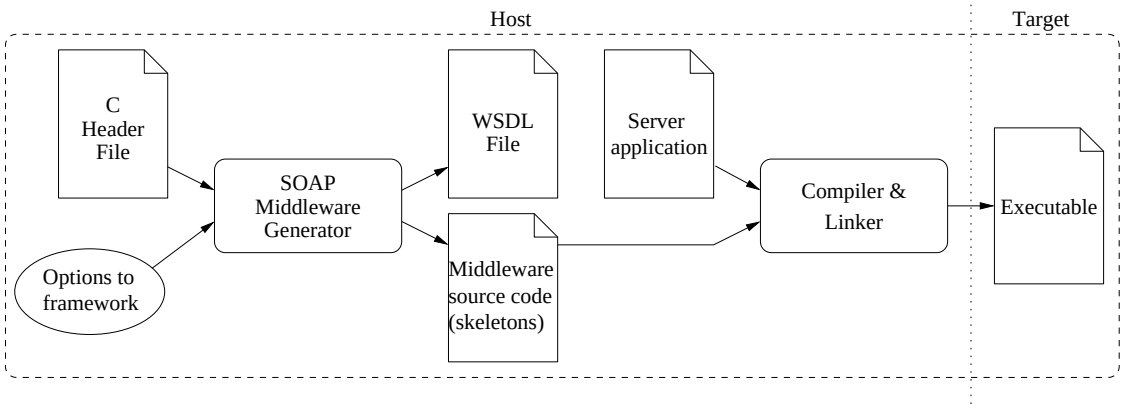


Figure 1.3: Overview of the process of generating service providers using the framework presented in DAT5 [2].

1.4.2 Creating Service Requesters

Creating a service requester is achieved in a similar manner as creating a service provider. A WSDL file is given as input to the framework, which will then generate stub code, enabling the application to request the services offered by a Web service in a transparent manner. Figure 1.4 shows how a service requester is generated. Initially, the WSDL description of the Web service is passed to the framework, which generates stub code.

Along with the generated stub code, the framework generates a C header file specifying the interface of the Web service in C. These header files are used by the application accessing the Web service to initiate requests to the Web service. In order to produce an executable, the application source code is compiled and then linked with the generated stubs.

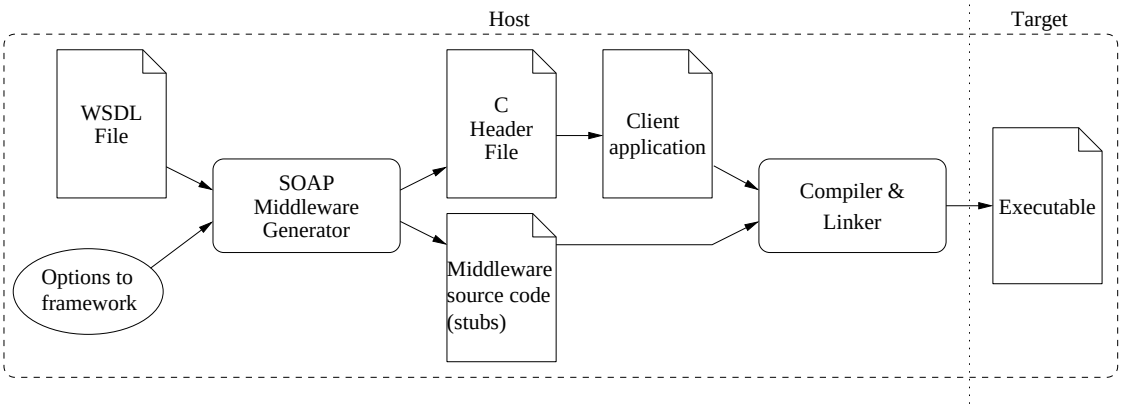


Figure 1.4: Overview of the process of creating service requesters using the framework presented in DAT5 [2].

1.4.3 Limitations

The aim of our work presented in DAT5 [2] was to study the applicability of Web services in distributed embedded systems. Based on an extensive analysis of the Web service technology, a conceptual design of a framework for developing Web services for distributed embedded systems, and a prototype implementation of the middleware code generated by this framework, it was concluded that it is feasible to apply Web services in distributed embedded systems.

However, DAT5 [2] only presented an overall architecture of this framework, and only a limited analysis of the C programming language was performed. This introduced simplifications of the functionality of the framework, which imposed limitations that are not desirable in many practical applications. Furthermore, DAT5 [2] only presented a design of the middleware code that the framework generates. No algorithms for generating this code where provided. Also, our prior work showed that one of the challenges when applying Web services in distributed embedded systems is the relatively large sizes of the messages that must be transmitted, and thereby the network bandwidth required. In order to meet this challenge, a redesign of the framework is needed, which allows for a more efficient network representation of the messages exchanged.

1.5 Problem Statement

In this thesis we extend our prior work presented in DAT5 [2], which studied the applicability of Web services in distributed embedded systems. In our prior work we concluded that Web services is applicable in distributed embedded systems, but applying it introduces an overhead, in terms of CPU, memory, and network usage, as an result of the added layers of abstraction. In this thesis we continue our study of the applicability of Web services in distributed embedded systems. We do this by asking the question: *How can the overhead resulting from the abstractions provided by Web services be minimized?*

We approach this question by developing a prototype implementation of a tool, which we name WSTOOL, for developing Web services in the C programming language. Our motivation for developing this prototype implementation is to gain an understanding of the problem area and its requirements, and to obtain a concrete platform for testing our algorithms, architectures, and designs.

As part of developing WSTOOL, four questions are central:

- How can a symmetrical mapping between C and SOAP be achieved?
- How can an interface be designed, enabling the network representation of SOAP messages to be changed, without changing the implementation of WSTOOL?
- How can algorithms be devised, which generate skeleton and stub layers with transparent interfaces?
- How can the knowledge of the expected behavior of service providers and service requesters be exploited to improve the performance of generated middleware layers?

In Chapter 2 we study the differences in the semantics offered by C and Web services, and investigate how a mapping between C and SOAP can be defined. Furthermore, we discuss what level of transparency can be obtained when developing distributed embedded systems using WSTOOL.

In Chapter 3 we discuss the overall architecture of WSTOOL and the skeleton and stub layers generated by the tool.

In Chapter 4 we study how the tool can support the use of network representations of SOAP messages other than XML, to support representations which require less bandwidth. An efficient interface is designed, which aims to minimize the overhead of introducing this abstraction.

In Chapter 5 we investigate how algorithms for generating skeleton and stub layers can be devised. The stub and skeleton layers are specific to a given instance of a service provider and service requester. We study how this service specific knowledge can be exploited to optimize the generated middleware layers.

In Chapter 6 we present the overall architecture of WSTOOL. A significant part of this thesis involves implementing WSTOOL, and in Chapter 7 we discuss its implementation. In Chapter 8 we use this implementation as a basis for evaluating the algorithms, architectures, and design devised in the thesis. An example embedded distributed application is developed using WSTOOL, on which performance experiments are performed. This service is also implemented using two alternative tools, and their performance is compared to that of WSTOOL.

Finally, in Chapter 9 we summarize the results achieved in this master's thesis, and discuss future work that can be carried out.

2 Analysis

In this chapter we present an analysis of the C programming language and the SOAP protocol. Our motivation is to study the differences in their type systems and the semantics they offer. This study is the basis for constructing a mapping between the type systems of C and SOAP.

Our approach presented in DAT5 [2] for developing service providers and requesters for C, allows application developers to have little knowledge about the Web service technology. However, it is not practical to make the middleware interaction completely transparent to the application developer, as the semantics of non-distributed applications can not be equally achieved in distributed applications. Therefore, we study what level of transparency is desirable when developing Web services, and which elements can not be made transparent for the application developer.

Finally, we discuss the notion of XML information sets which provide a mechanism for serializing SOAP messages in a different format than XML. Our motivation is to study how the stub and skeleton layers generated by WSTOOL can be altered to allow for a more efficient network representation of the SOAP messages they exchange.

2.1 Semantics of Web Services and C

Web services allows for two styles of message exchange in distributed applications: Document style and RPC style. Using message exchange based on the document style, the messages exchanged are XML documents, and it is up to the application to decide upon the semantics of these documents. This message style is flexible and allows for many interesting exchange patterns. Typically, this style finds its use in business-to-business applications.

When using the RPC style, constraints are imposed on the permissible structure of the SOAP messages that can be exchanged. Furthermore, only one message exchange pattern is allowed using this style, namely the request-response message exchange pattern. An interaction in a Web service using the request-response pattern consists of the exchange of exactly two SOAP messages. Initially, a message is sent by the service requester to the service provider (the request). The service provider processes the request, and a message is sent by the service provider to the service requester (the response).

The RPC style defines the semantics of such message exchanges, namely the invocation of a procedure and the response of invoking this procedure. This is the style that we have chosen to support in our tool. In the following we analyze the differences in the semantics of a SOAP RPC message and a C function call.

2.1.1 Invocation Semantics

When invoking a function in the C programming language, the invocation semantics is *exactly-once* [4], meaning that whenever a function is invoked, the function body will be executed, and it will be executed exactly once.

However, when invoking a remote procedure in a Web service, the invocation semantics differ, due to the fact that messages may be transferred between the service requester and the service provider through a network. This underlying network introduces some challenges that are not present when a function is invoked locally. As an example, depending on the network protocol, messages can be lost, and duplicate or corrupted messages can appear, thereby resulting in the request not being carried out or carried out several times. As the SOAP standard does not require any specific underlying network technology to convey SOAP messages, there is no fixed invocation semantics in Web services. The invocation semantics depend on the services offered by the underlying network protocols. The underlying protocol that we use in this thesis is the HTTP protocol, which is operating on top of the TCP protocol. The TCP protocol, from the perspective of the application, provides a reliable connection between two peers. Among the features of TCP is retransmission of lost packages, filtering of duplicate packages, and integrity checks of packages. Consequently, when a procedure is invoked, if the entire request reaches the service provider, it is guaranteed that no duplicate messages will be handed over to the application. Similarly, when the service provider sends the response messages, the same guarantees are present. Therefore, either the invoker receives the result of the invocation, or an error message stating that a network error occurred. In the first case, the invoker knows that the procedure was invoked exactly once. In the latter case the invoker knows that the procedure was invoked either exactly once, or not at all. Therefore, using the HTTP and TCP protocols to convey SOAP messages, an *at-most-once* invocation semantics is achieved.

2.1.2 Types

The C programming language is a typed language, and the SOAP protocol used in Web services allows for exchanging typed information. However, the type systems of the C language and the SOAP protocol differ, as well as their mechanisms for constructing new types to achieve a higher level of abstraction. In the following, we study these two type systems and their differences.

Simple Data Types

The C programming language allows for a limited range of simple data types [10]:

- `char`, a single byte
- `int`, an integer
- `float` a single-precision floating point
- `double` a double-precision floating point

C allows for specifying the value range of some of these data types by using a set of qualifiers, namely `short`, `long`, `unsigned`, and `signed`. The exact value range of a simple data type depends on the specific platform that the C application is executing on, i.e. on some platforms an `unsigned short int` may be represented by 16 bits of memory, thereby allowing values in the range of 0 to $(2^{16} - 1)$. On other platforms it may be represented by 32 bits of memory, thereby allowing values in the range of 0 to $(2^{32} - 1)$. The ANSI C standard specifies the minimum value range which each of the simple data types in C must have on any platform.

Finally, C provides the simple data type of *enumerations*, which are unique types that have integral values. Each enumeration is a set of named constants. Example 2.1 illustrates the declaration of an enumeration type `enum color` that has the value set of `RED`, `GREEN`, and `YELLOW`.

```
1  enum color {
     GREEN,
     RED,
     YELLOW
5  };
```

Example 2.1: The declaration of an enumeration type `enum color` in ANSI C.

Similarly, the SOAP protocol allows for a range of simple built-in data types. The type system of SOAP is based on the XML Schema [24, 25] specification, which provides a way to declare the types and structure of XML documents. XML Schema provides a set of simple data types similar to the simple data types provided by C. Furthermore, it provides several other simple data types, such as types that represents calendar dates, booleans, strings, encoded binary data, and several others which we will not discuss further.

Derived Types in C

Besides the simple data types, C provides a mechanism to construct new types from existing types. The mechanisms provided by C to allow for this are:

- Arrays
- Pointers
- Structures
- Unions
- Typedefs

Arrays, which are known from many programming languages, provides a mechanism for creating an ordered and indexed set of data objects having the same data type.

Pointers are low level abstractions, that provide a way to store the physical memory address of a data object of a specific type. As an example the C statement `enum color *c;`, creates a variable of the type *pointer to enum color*, i.e. the variable `c` can hold the memory address of a data object of the type `enum color`.

Struct types are definitions of a new type, which contains a sequence of data objects of various existing types, as illustrated in Example 2.2. This example illustrates the declaration of the new type `struct student`, which is a derived type consisting of a sequence of data objects of the simple types `unsigned int` and `float`.

```
1  struct student {
     unsigned int id;
     float average_grade;
   };
```

Example 2.2: The declaration of a derived type `struct student` in ANSI C.

Union types contain a specification of a set of different data types, not unlike structs. However, an instance of a union type only holds one value, which can have any of the types specified in the union set. Example 2.3 illustrates the declaration of the type `union product_price`. Two variables, `price1`, and `price2`, are instantiated and assigned values. The variable `price1` is assigned a value of the type `unsigned int`, and the variable `price2` is assigned the value of the type `float`.

```
1  union product_price {
     unsigned int simple_price;
     float complex_price;
   };
5
   union product_price price1;
   union product_price price2;

   price1.simple_price = 5;
10 price2.complex_price = 10.5;
```

Example 2.3: The declaration of a derived type `union product_price`, and the instantiation of two variables of this type in ANSI C.

Finally, C allows for defining new types by declaring a type as a synonym for an existing type, by the use of the `typedef` keyword. This is illustrated in Example 2.4, where a new type `price` is declared as a synonym for the existing type `unsigned int`.

```
1  typedef unsigned int price
```

Example 2.4: The declaration of a new type `price`, as a synonym for the existing type `unsigned int` in ANSI C.

Derived Types in XML Schema

Similarly, the XML Schema specification allows for deriving new types from existing data types. However, it provides more flexibility to do so than C. We here discuss a limited subset of the ways XML Schema makes this possible.

XML Schema allows for defining new types by restricting the value space of an existing type. Example 2.5 illustrates the definition of a new type `less-than-one-hundred-and-one`, which is based on the existing type `short`. It has the same properties as the `short` type, except its value space is limited to a maximum of 100.

```
<simpleType name='less-than-one-hundred-and-one'>
  <restriction base='short'>
    <maxExclusive value='101'/>
  </restriction>
</simpleType>
```

Example 2.5: The declaration of a derived type `less-than-one-hundred-and-one`, restricting the value space of the existing type `integer` in XML Schema.

Furthermore, XML Schema also allows for defining new types by using existing types as building blocks, by defining a *complex type*. This concept is illustrated in Example 2.6, where a new type `employee` is defined. This type is a sequence of two data objects named `firstname` and `lastname`, which are both of the type `string`.

```
<xs:element name="employee">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Example 2.6: The declaration of a derived type `employee` in XML Schema, which is the sequence of two data objects of the type `string`.

2.2 Mapping between C and SOAP

In the following we discuss the possibilities of mapping the C type system to the SOAP type system, and vice versa.

2.2.1 Limitations

As we discussed in Section 2.1, C and SOAP are both have type systems. The SOAP type system is based on the type system offered by XML Schema, which is more flexible than the type system of C, and offers both a larger range of basic data types and opportunities to derive new types.

In this section we discuss the elements of the C type system that are meaningful to map to the XML Schema type system. Due to the comprehensiveness of the XML Schema specification, we choose to focus on mapping a subset of the type system offered in this specification to the C type system. We define a mapping that is symmetric in the sense that the elements we focus on can be mapped from C to XML Schema as well as from XML Schema to C. However, for types that are specific to XML Schema, that do not directly map to C, we do not define a mapping. For instance, we do not define a mapping of the XML Schema type `date`.

It should be noted that in order for WSTOOL to be fully compliant with the Web service standard, a mapping of the entire XML Schema type system to C should be established.

2.2.2 Direction of Parameters

The SOAP protocol handles parameters to a procedure in a special way, in the sense that they can have directions. A parameter can have any of the following directions:

- In
- Out
- In-Out

Parameters with the direction In are input to the procedure, parameters with the direction Out is output from the procedure, and finally, parameters with the direction In-Out is both input and output from the procedure. This concept of direction is different from C, since C does not operate with direction of parameters. Conceptually, in C, a parameter that is not a pointer is known to be a input parameter, since the value of the actual parameter is copied when the function is called. Parameters that are pointers can be considered both In and Out parameters.

In order to allow specifying the direction of the parameters in C, and thereby allowing WSTOOL to know how to marshal/unmarshal the parameter, we introduce a naming scheme of the parameters used in C functions.

```
1  void function(int *in_parm1, int *in_out_parm2, int *out_parm3);
```

Example 2.7: The declaration of a function with three parameters with different directions.

Example 2.7 illustrates this naming scheme. A function is declared taking three parameters `in_parm1`, `in_out_parm2`, and `out_parm3` with the direction In, In-Out, and Out respectively. The parameters that are either Out or In needs to be declared as pointers, in order to be able to return a value, by changing the value of the data object pointed to. However, we introduce a convention that all parameters should be pointers, in order to avoid any unnecessary copying of data. Furthermore, we introduce the convention that return types are not allowed, since not all data types can be returned from a C function (such as arrays).

2.2.3 Specifying Functions to be Exposed as Web Services

When writing a C application to be exposed as a Web service, it may not be desirable to expose all the functions contained within the application as part of the Web service. Some of these may be helper functions that are part of a top down design, and assist some functions in implementing their functionality, but which should not be accessed outside of these functions. Therefore, we introduce a convention which allows the developer of a Web service to specify which functions should be exposed as part of the Web service.

The name of any function which is to be exposed as part of the Web service is prefixed with the string "ws_". We illustrate this in Example 2.8. In the example, the function `ws_convert_to_euros` is exposed as part of the Web service, while the function `helper_function` is not.

```
1  void ws_convert_to_euros(float *in_out_value);

   void helper_function(float *parm);
```

Example 2.8: The declaration of two functions—`ws_convert_to_euros` which should be exposed as part of a Web service, and `helper_function` which is a function that should not be exposed as part of the Web service.

2.2.4 Mappable C Elements

In the following, we discuss the elements of the C type systems that are meaningful to map to the XML Schema type system. Most types are meaningful to map, with a few exceptions:

- Pointers
- Unions

Pointers are low level abstractions that are used in C to hold the addresses of physical memory location of data objects. Since C only supports passing parameters by their value, pointers are among other things used to achieve *call by reference* in function calls.

However, mapping pointers to the XML Schema type system makes little sense. Semantically, pointers represent a memory locations on a certain computer. Web service provides no mechanism to access a specific memory locations of a service provider, and therefore it makes little sense to map pointer types.

Unions are constructions used in C to declare variable types, as illustrated previously in Example 2.3. As it can be seen from the example, the value of an instantiation of a union type is variable, e.g. by accessing the member `price1.simple_price` it has the type `unsigned int`, and by accessing the member `price1.complex_price` it has the type `float`. However, there is no mechanism in C to determine what type a given instantiation of a union has, without having knowledge about the previous history of the assignments to the variable. Since WSTOOL will not have access to this information, it is not possible to marshal/unmarhsal a union type, and therefore it is not meaningful to define such a mapping.

Finally, C provides two type qualifiers that can be used when specifying the parameters of a function:

- Const
- Volatile

Declaring a parameter `const` announces that the value of the parameter must not be changed, or in the case of the parameter being a pointer, it declares that the variable pointed to must not be changed.

Declaring a parameter `volatile` announces that the variable has special properties indicating to the C compiler that no optimizations should be performed on the variable.

Neither of these type qualifiers have meaningful mappings to XML Schema. Declaring a parameter `volatile` does not specify anything about the properties of the value of the parameter, only that the compiler should not optimize it, and therefore there is no meaning in mapping it to XML Schema. Declaring a parameter `const` is only meaningful in the case of the parameter having the direction In, and in this case the semantics of the parameter being constant is already achieved. Therefore, no action is necessary to map these type qualifiers.

2.2.5 Special C Elements

C contains some other types that are mappable, however they require some special attention:

- Strings
- Arrays

C represents strings as arrays of chars. A string of length 5 gets represented as 6 characters in memory. The first 5 characters represent the string, and the last character is the special symbol "\0", which represents the termination of the string. This concept is known as *null-termination*. However, chars can also be used to represent a decimal number in the range 0-255. Therefore, the semantics of chars and char arrays depend on their usage. WSTOOL is unable to decide upon the semantics of a char array, e.g. whether is should be treated as a null-terminated string or as an array of bytes.

Therefore, to make the semantics explicit, we introduce two new types to be used in WSTOOL:

- ws_string_char (defined by `typedef char ws_string_char`)
- ws_string (defined by `typedef ws_string_char *ws_string`)

When using these types, the application makes it explicit that the type is a string/ character and not an array of bytes/a byte.

Furthermore, arrays in C differ from arrays in many other languages due to the fact that it is not possible to retrieve any information about the length of an array, passed as a parameter to a function. This is often handled by adding an extra parameter to the function that indicates the size of the array, as illustrated in Example 2.9

```
1  void function(int (**in_array)[], unsigned long *in_array_size);
```

Example 2.9: The declaration of an array of `int` as a parameter, and an additional parameter indicating the size of the array.

Therefore, we add a convention to WSTOOL that whenever a parameter is an array type, an extra parameter should be present which indicates the size of the array. The name of this parameter should be the name of the preceding parameter (which is an array type), postfixed with the string "_size" as illustrated in Example 2.9.

2.2.6 Defining a Mapping of Basic Types

In the following we define a mapping of the basic C data types to their corresponding XML Schema data types. The XML Schema specification specifies the exact value range of the different types. As the C specification only specifies the minimum value range a certain type must have, we base our mapping on this range.

The mapping of the basic types are simple, as it is a matter of choosing the XML Schema type that can hold the same range of values. Table 2.1 illustrates the different C types and their equivalent XML Schema data types.

C Data Type	XML Schema Data Type
signed char	byte
unsigned char	unsignedByte
signed short	short
unsigned short	unsignedShort
signed int	short
unsigned int	unsignedShort
signed long	int
unsigned long	unsignedInt
signed long long	long
unsigned long long	unsignedLong
float	float
double	double
long double	double

Table 2.1: The mapping of basic C types to basic types in XML Schema.

The XML Schema data types can be mapped to C in an equivalent manner. However, we do not illustrate this here, as it is equally simple.

2.2.7 Defining a Mapping of Structs

Next, we define the mapping of derived types in C to XML Schema, beginning with the mapping of structs. In order to map structs to XML Schema, we use the concept of complex types in XML Schema. As described previously, these allow composite types from existing types in a flexible manner. One of the opportunities offered by complex types is to define a type as a `sequence` of elements of given types. Conceptually, a sequence is similar to a C struct, as structs in C also are derived types containing a sequence of elements (called members in C) of certain types. We illustrate the mapping of a C struct to a representation in XML Schema in Figure 2.1.

C Data Type	XML Schema Data Type

`struct a_struct{`
`  int member1;`
`  float member2;`
 	
`  struct {`
`  int member1;`
`  } member3;`
 	
`};`
	`<complexType name="struct_a_struct">`
	`  <sequence>`
	`  <element name="member1" type="short"/>`
	`  <element name="member2" type="float"/>`
	`  <element name="member3">`
	`  <complexType>`
	`  <sequence>`
	`  <element name="member1" type="short"/>`
	`  </sequence>`
	`  </complexType>`
	`  </element>`
	`  </sequence>`
	`</complexType>`

Figure 2.1: The mapping of a C struct to an XML Schema `complexType`.

A new C type `struct a` is defined, which has three members `member1`, `member2`, and

member3. The first two members are basic C data types, however the member member3 is special, since it is itself a derived type, namely another struct type. When defining this new type in XML Schema, we define a sequence of elements with the same names as the members of the C structs. Each of these element's types are the mapped version of the structs members type. The first two members are trivial, and the third member, member3, is special since a new type is defined to represent this. In this case this new type has no name, and therefore it is defined embedded into the definition of the element member3.

C Data Type	XML Schema Data Type

struct a_struct{
  int member1;
  float member2;
 	
  struct embedded {
  int member1;
  } member3;
 	
};
	<complexType name="struct_a_struct">
	  <sequence>
	  <element name="member1" type="short"/>
	  <element name="member2" type="float"/>
	  <element name="member3" type="embedded""/>
	  </sequence>
	</complexType>
	
	<complexType name="embedded">
	  <sequence>
	  <element name="member1" type="short"/>
	  </sequence>
	</complexType>

Figure 2.2: The mapping of a C struct containing a named struct as its member to two XML Schema ComplexType types.

Figure 2.2 shows a special case of mapping a C struct to a XML Schema representation. Again, a C struct is defined which has the same three members as before. The member member3 is once again another derived type in the form of a struct. However, this time the struct is a named struct, struct embedded. In this case, this member constitutes the definition of a new type, i.e. it is possible to instantiate a variable both of the type struct a_struct and struct embedded. Therefore, the mapping of this type is mapped to the definition of two new complex types in XML Schema, as illustrated in Figure 2.2.

2.2.8 Defining a Mapping of Enumerations

In order to map C enumerations to XML Schema, we use the concept of deriving new types in XML Schema, by restricting the value space of an existing type. For this we use the simpleType construct. We illustrate this in Figure 2.3.

We derive a new type in XML Schema named enum_color, which is a restriction based on the existing type string. We define the new type to be able to be either the string "RED", "GREEN" or "BLUE". Although C internally represents enumerations as integers, we choose to map them to strings in XML Schema. This is due to the fact that when doing the mapping from XML Schema to C, if they where mapped as integers, there would be no reasonable name for the enumerator in C, i.e. they would be mapped to enum color{1, 2, 3}.

C Data Type	XML Schema Data Type

enum color {
RED,
GREEN,
BLUE
};
	<simpleType name="enum_color">
	<restriction base="xs:string">
	<enumeration value="RED"/>
	<enumeration value="GREEN"/>
	<enumeration value="BLUE"/>
	</restriction>
	</simpleType>

Figure 2.3: The mapping of a C enumeration to a XML Schema `SimpleType`.

2.2.9 Defining a Mapping of Typedefs

Mapping a C type defined using `typedef` is straightforward, as such a type is merely an alias for an existing type. Therefore, when mapping a typedef C type to a XML Schema type, it is a matter of copying the definition of the type that is being defined as a new type. Depending on whether this type corresponds to a complex or simple type in XML Schema, this is done in slightly different ways, as illustrated in Figure 2.4.

C Data Type	XML Schema Data Type
typedef unsigned int u_int; typedef struct a_struct { int member; } typedef_struct;	<simpleType name="u_int"> <restriction base="unsignedShort"/> </simpleType> <complexType name="typedef_struct"> <sequence> <element name="member" type="short"/> </sequence> </complexType>

Figure 2.4: The mapping of a two C typedef types to a XML Schema representation.

2.2.10 Defining a Mapping of Strings

Whenever the application desires to expose a C char type as a decimal value, the type `char` is used. When it needs to be exposed as a character value the type `ws_string_char` is used. Similarly, the types `char[]`/`char*` and `ws_string` is used to represent an array of bytes or strings respectively. When mapping these types to XML Schema, the basic XML Schema type `string` is used. When mapping `ws_string_char`, once again the concept of restricting the value range is applied, as illustrated in Figure 2.5.

2.2.11 Defining a Mapping of Arrays

To map an array to XML Schema, it is encoded as a sequence of XML elements named `item`, which all have the type of the array elements. XML schema provides an option to specify that a sequence of elements should occur a certain amount of times, as we illustrate in Figure 2.6. In this example we illustrate how an unbounded array of integers is mapped

C Data Type	XML Schema Data Type

`typedef char ws_string_char;`	`<simpleType name="ws_string_char">`
	`  <restriction base="string"`
	`  minLength="1"`
	`  maxLength="1"/>`
	`</simpleType>`

Figure 2.5: The mapping of a C char to a XML Schema restricted type.

to a sequence of XML elements named "item", which have the type `short`, and should occur between 1 and an unlimited amount of times.

C Data Type	XML Schema Data Type

`int array[];`	`<complexType>`
	`  <sequence>`
	`  <element name="item"`
	`  type="short"`
	`  minOccurs = "1"`
	`  maxOccurs = "unbounded"/>`
	`  </sequence>`
	`</complexType>`

Figure 2.6: The mapping of a C array to XML Schema.

2.2.12 Mapping from XML Schema to C

We have defined a mapping of all the C types to their equivalent XML Schema types. We have limited ourselves only to map, with exceptions, the entire C type system to XML Schema and not the other way around. From XML Schema to C we only define a symmetric mapping, meaning that there is a mapping from C to XML Schema for all types mappable from XML Schema to C. We do not describe the mapping here, since it is simple when having defined the mapping from C to XML Schema. For instance, we defined that a C struct maps to a XML Schema sequence. Therefore, we also map an XML Schema sequence to a C struct, and so forth with the other types.

2.3 Desirable Level of Transparency

As discussed in the previous sections, the type systems and the invocation semantics offered by C and the SOAP protocol are different. Furthermore, we have introduced some constraints to the permissible structure of C programs. These differences imply that developing for Web services can not be completely transparent to the application developer.

In this section we discuss the level of transparency that is desirable using our tool, and which details the application developer should be aware of.

2.3.1 Invocation Transparency

Due to the difference in invocation semantics between C and SOAP, an application acting as a service provider or requester should have an opportunity to notified about the status of an invocation.

Service Requesters

In the situation of developing a service requester, whenever invoking a remote procedure, the application needs a mechanism to receive information about the outcome of the request. Due to the at-most-once invocation semantics offered when using HTTP as the underlying protocol, there are the following general outcomes of a request:

- Request not sent.
- Request sent but no response received.
- Request sent and response with SOAP error message received.
- Request sent and response received.

A request can be attempted but the establishment of a connection to the service provider can be unsuccessful due to many reasons, such as the network being temporarily unavailable or the address of the service provider having changed.

Furthermore, a request can be sent but no response received. This can be due to the service provider having an internal error, that prohibits it from sending a reply, or the network connection becoming unavailable when attempting to send the response. Typically a service requester will detect this due to the timeout of waiting for a response.

Furthermore, a request can be sent, and a response received containing a SOAP error message, indicating that something was wrong with the request.

Finally, the request can be sent, and a response received, indicating the successful execution of the remote procedure.

When developing a service requester, the application needs to be able to receive information about the outcome of the request, so the application can handle potential errors. Therefore we introduce another convention regarding the stub code generated by WSTOOL, namely that the invoker of a stub function has to pass a pointer to a data structure that will contain the status of the outcome of the request as illustrated in Example 2.10.

```
1  typedef struct {
     enum { WS_NO_NETWORK_CONNECTION,
            WS_NO_SUCH_URL,
            WS_RESPONSE_TIMED_OUT,
5           WS_SOAP_FAULT_RECEIVED,
            WS_OK
          } outcome;
     ws_string soap_fault_msg;
   } ws_request_outcome;
10
   void ws_function(ws_request_outcome *outcome, int *in_parm);
```

Example 2.10: The declaration of a stub function to invoke a Web service remote procedure, and the definition of a data-structure to represent the outcome of invoking this procedure.

Whenever a stub function is generated, the first parameter is always a pointer to the data structure `ws_request_outcome`, which is defined as a C struct with two members: `outcome` and `soap_fault_msg`. After invoking the function, the application can check the value of the member `outcome` in order to establish the status of the request. The `outcome` can have five values as described below:

- `WS_NO_NETWORK_CONNECTION` - This indicates that no network connection could be established.
- `WS_NO_SUCH_URL` - This indicates that a network connection could be established, but the resource identified by the URL that the generated stub code attempted to access is not available.
- `WS_RESPONSE_TIMED_OUT` - This indicates that the request was sent, but the waiting for the response timed out. Thereby the status of the execution of the remote procedure is unknown.
- `WS_SOAP_FAULT_RECEIVED` - The request was sent, and a response was received. However the response contained an error message. The member `soap_fault_msg` will contain the message.
- `WS_OK` - The request was sent and a response was received, indicating that the procedure was executed successfully.

One of these status messages are special, namely the `WS_SOAP_FAULT_RECEIVED`, which indicates that the request was sent and that a response was received which contained a SOAP fault message. This message is then contained in the `soap_fault_msg` member of the `ws_request_outcome` struct.

Service Providers

In the case of creating service providers, the achievable level of invocation transparency is higher than when building service requesters. This is due to the fact that the service provider does not need the same level of information about the status of the request/response. Either the request arrives to the service provider or it does not. If it arrives, the generated skeleton code has the responsibility of unmarshalling the request, invoking the proper function and sending the reply. If the request is invalid, for instance a non-existent service is requested, the skeleton code can handle sending a SOAP fault message back to the service requester without the Web service application having any knowledge about this. The errors that the middleware can handle transparently are:

- Unknown data encoding of SOAP message.
- Request for non existing procedure.
- Bad arguments. An incorrect number of parameters were specified, or the specified parameters contained invalid data types.
- Version mismatch. The SOAP message was sent in a different message format than version 1.2.
- Must understand error. The SOAP message contained a header that WSTOOL did not understand, which had the `mustUnderstand` attribute set to `true`.

The SOAP protocol specifies the error messages that should be sent when encountering such errors. However, there are other types of errors that can not be handled by the middleware which are application specific. In such error situations, the application may need to send error messages back to the service requester in a manner described by the SOAP protocol. The SOAP protocol provides a flexible way to generate fault messages, specifically by sending SOAP fault messages. The SOAP protocol describes several standard fault codes used to indicate the type of error that occurred. Two important examples of these are:

- Sender
- Receiver

The code `sender` indicates that there was a problem with the message sent by the service requester, e.g. the lack of authentication. The code `receiver` indicates that the service provider had a temporary problem processing the message, e.g. some of the back-end functionality was temporarily unavailable. SOAP fault messages of this type indicate that the request may succeed if it is resent at a later point of time.

In order to allow the application that offers a Web service to send these messages, we introduce another convention used when writing C functions that should be exposed as Web services, as illustrated in Example 2.11.

```
1   typedef struct {
      enum { WS_SENDER,
             WS_RECEIVER,
             WS_OK
5          } outcome;
      ws_string reason;
    } ws_response_outcome;

    void ws_function(ws_response_outcome *outcome, int *in_parm);
```

Example 2.11: The declaration of a C function to be exposed as part of a Web service. This includes an definition of a data structure to represent the outcome of invoking the function.

This example illustrates the declaration of a C function that should be exposed as part of a Web service, and the declaration of the data structure `ws_response_outcome` as a struct. This struct contains a member `outcome` that can have three values: `WS_SENDER`, `WS_RECEIVER`, and `WS_OK`. The values `WS_SENDER` and `WS_RECEIVER` represents the type of error as described above. When specifying a SOAP fault, the SOAP protocol allows supplying a `reason` value, which is a string meant to be read by a human. This value can be used to describe that reason for the fault using natural language, e.g. "No person with ID 42 was found in the database". If the `outcome` data structure is set to any of these two values, the member `reason` can be set to indicate the reason for the error in this human readable form. Finally, the `outcome` member can have the value `WS_OK` indicating that the request was processed successful.

2.3.2 Memory Transparency

The C programming language does not provide garbage collection. Therefore an important aspect of the stub and skeleton code generated by WSTOOL is the way the allocation and

releasing of memory is handled. We here discuss how to place responsibility for allocating and de-allocating memory. In Chapter 5 we will discuss algorithms for handling memory in an efficient manner.

Stub Code

The stub code generated by WSTOOL declares all its parameters as being pointers to the data-type they represent.

As is often the policy in the C standard library, when passing pointers as arguments to a function, it is the responsibility of the caller to free the allocated memory. In the case of the parameter having either the direction In-Out or the direction Out, the value of the memory location pointed to is overwritten by the stub code. However, a special case exists. Consider the case illustrated in Example 2.12.

```
1  void ws_function(ws_request_outcome *outcome, int (**in_out_parm)[], int *parm_size);
```

Example 2.12: The declaration of a stub function that take the parameter `in_out_parm` that is a one-dimensional array of unlimited size.

This example illustrates a stub function that takes the parameter `in_out_parm`, which is a one-dimensional array of unlimited size. In this case, the memory pointed to by the parameter can not in all cases simply be overwritten with new values, due to the fact that the output can be an array of a different size. Therefore, in this special case, the stub code needs to allocate space to hold the response. Therefore, it is the responsibility of the application invoking such a function to free the memory that was pointed to by `in_out_parm` before passing it to the function, as well as the memory pointed to after the function has returned. In Example 2.13 we show an example of how this can be handled. An array and a variable which points to this array is created (the variables `array` and `array_ptr` in the example). Furthermore a copy of `array_ptr` is created (`array_ptr_before`). Next the function is executed, and afterwards it is checked if `array_ptr` does not point to the same location anymore (`array_ptr_before != array_ptr`). If this is the case, it means that the function has internally allocated extra space (e.g. because it needs to return a larger sized array), and in this case the memory is de-allocated.

Skeleton Code

In the skeleton code generated by WSTOOL, a higher level of transparency is achievable, since all function calls to the application code is performed by the skeleton code. Thereby it is the responsibility of the skeleton code to free any allocated memory.

Therefore, the skeleton code has the responsibility of allocating memory for all the parameters passed to the function, and free the memory allocated after the function has returned. Furthermore in the case that the parameter has the direction Out or In-Out, and is an array, the function can potentially also allocate memory to hold the response. In this case it is also the responsibility of the skeleton code to free this memory in a similar manner as illustrated in Example 2.13.

```
/* Creation of variables */
int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8 ,9};
int (*array_ptr)[]  = &array;
int (*array_ptr_before)[]  = array_ptr;
int parm_size = 10;
ws_request_outcome outcome;

/* Invocation of the function */
ws_function(&outcome, &array_ptr, &parm_size);

/* De-allocation of memory allocated
 * internally in the function.
 */
if (array_ptr_before != array_ptr)
  free(array_ptr);
```

Example 2.13: An example of an application invoking a stub function which internally allocates memory. The application has the responsibility of de-allocating this memory.

2.3.3 Concurrency Transparency

In the following chapters when we discuss the design of the skeleton code, we will illustrate that we use a multi-threaded architecture to allow service providers to handle multiple requests at the same time. Therefore, the functions exposed as part of the Web service, and all the functions called within these, can be executed in parallel. Consequently, the application has the responsibility of ensuring that proper mutual exclusion mechanisms are applied if this is needed, since the skeleton code can not do this.

When building service requesters, there stub code does not introduce concurrency, and therefore no extra measures has to be taken by the application in this situation.

2.4 Alternative Representations of SOAP Messages

One of the conclusions of our previous work presented in DAT5 [2] was that one of the limitations of applying Web services in distributed embedded systems is the size of the SOAP message format. In this section we analyze how a SOAP message can be serialized in a different manner in order to allow for a more efficient network representation.

The W3C has introduced an abstract data set called the XML information set [23] for use in other specifications, such as SOAP, that need to refer to the information encapsulated in XML documents. An XML document has an information set if and only if it is well-formed and satisfies certain XML namespace constraints. An XML information set may be created by parsing an XML document or by other means, such as parsing a different representation of an XML document.

An XML information set consists of a number of information items. An information item is an abstract description of some part of an XML document. The information set for a well-formed XML document will always contain exactly one “document” information item, which represents the document itself, and has a set of named properties from which the different entities of the document are accessible. One example of such a property is the “children” property, which is an ordered list of child information items. Other information items introduced by the XML information set specification are “element” information items, “attribute” information items, “namespace” information items, and others. These all correspond to the entities found in an XML document, i.e. for each element in a well-formed XML document, an element information item will be present in the information

set representing this XML element.

```
1  <?xml version='1.0' ?>
   <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
     ...
   </env:Envelope>
```

Example 2.14: A simple example of part of a SOAP message.

Example 2.14 illustrates the envelope part of a SOAP message. When leaving out the elements that are children of the envelope element, the XML information set for this XML document contains the following information items:

- A document information item.
- An element information item with namespace name "http://www.w3.org/2003/05/soap-envelope", local part "Envelope" and prefix "env".
- Two namespace information items for the "http://www.w3.org/XML/1998/ namespace" and the "http://www.w3.org/2003/05/soap-envelope" namespaces.

It should be noted that the namespace information items for the "http://www.w3.org/XML/1998/namespace" namespace is part of the XML information set because the document uses namespaces, and therefore the information item is implicitly defined.

It is important to note that the W3C specification of XML information sets does not favor a specific interface to access information sets, neither does it specify how the information set is obtained. It merely describes a set of definitions used to represent structure and data. This is an important issue, as a SOAP message is defined as an XML information set. Therefore, SOAP messages are not limited to being serialized as XML documents, but can be serialized in any representation that has an XML information set. In order to allow for more efficient representations of SOAP messages in terms of message size, a compact serialization could be chosen. The only requirement is that it has the same XML information set as its corresponding XML document serialization would have. In Chapter 4, we will explore how the concept of XML information sets can be incorporated in the design of WSTOOL.

2.5 Summary

In this chapter we have presented an analysis of the C programming language and the SOAP protocol, and discussed the differences in their type systems and the semantics they offer.

We have defined a mapping of the C type system to the XML Schema type system, which is the type system used by SOAP. Furthermore, we have discussed how a mapping from XML Schema to C can be constructed. We have limited ourselves to mapping only a subset of XML Schema to C, due to the comprehensiveness of the type system of XML Schema.

We have introduced the notion of XML information sets, and discussed how it can be used to achieve a more efficient network representation of SOAP messages.

We end this chapter by summarizing the conventions and constraints we have introduced to the permissible structure of Web services developed using WSTOOL. These have

been introduced in order to handle the differences in C and SOAP, and to cope with the elements of the development process that can not be made transparent due to these differences.

Naming of functions: When developing a Web service, it is not always desirable that all functions get exposed as part of the Web service. Therefore, we introduce the convention that a function name is prepended with the string "ws_" if it is to be exposed as part of the Web service.

Direction of parameters: The SOAP protocol offers the concept of direction of parameters, and a parameter can have either the direction In, In-Out, or Out. Therefore, we introduce the following conventions:

- All names of the formal parameters of a function must be prefixed with their direction, e.g. a parameter `a` which has the direction In is named `in_a`.
- All parameters should be declared as pointers, both to allow for better efficiency, and to achieve the semantics of a function having parameters with the direction Out or In-Out.

Special data types: The C programming language contains two types that needs special attention, namely strings and arrays. C strings are arrays of the data type char, which introduces an ambiguity in the semantics of arrays of chars. They can either represent a string or an array of decimal numbers. Therefore, we have introduced the following two new data types to be used when declaring strings, in order to avoid this ambiguity:

- ws_string_char
- ws_string

In C, no mechanism is provided to decide the length of array types. Therefore, we introduce the convention that any parameter which is an array type should be immediately followed by a parameter declaring the number of elements in the array.

Invocation semantics: The C programming language offers exactly-once invocation semantics, while the invocation semantics offered by the SOAP protocol depends on the underlying network. Using HTTP over TCP as the underlying network, SOAP offers at-most-once invocation semantics. Due to the difference in invocation semantics, we have introduced a set of conventions and constraints on the structure of the C functions that should be exposed as Web services:

- The first parameter of all stub functions is a pointer to the data structure `ws_request_outcome` which represents the outcome of executing the remote procedure.
- The first parameter of all skeletons functions is a pointer to the data structure `ws_response_outcome`. The value of this parameter should be set to represent the outcome of executing the function.

Memory transparency: The C programming language does not provide garbage collection. This makes the handling of allocation and deallocation of memory an important issue. Therefore, we introduce the following conventions regarding the parameters passed to functions that are exposed as part of a Web service:

- It is the responsibility of the generated skeleton code to deallocate the memory allocated to the parameters passed to the functions.
- It is the responsibility of the application developer to allocate and deallocate memory passed to a stub function.
- In the case that a function exposed as part of a Web service allocates memory internally, in order to pass an array as an output parameter, it is the responsibility of the skeleton code to deallocate this. At the client side it is the responsibility of the application developer.

Concurrency transparency: When exposing a set of C functions as a Web service, the generated skeleton code introduces concurrency, since a multi-threaded architecture is used to allow multiple requests to be processed at the same time. Therefore, the application exposed as a Web service needs to enforce mutual exclusion if needed.

3
Overall Architecture

In this chapter we provide an overall architecture of WSTOOL, and the middleware layers generated by WSTOOL. Our motivation is to provide an overview of the overall design of the tool and the middleware generated by the tool. In the following chapters we will discuss the design of specific parts of the tool and the middleware generated by the tool in greater detail.

3.1 Architecture of wstool

WSTOOL allows for generating service providers by generating a skeleton layer that allows exposing the functionality of an existing application transparently for the application. Similar, the tool allows for generating service requesters by generating a stub layer, that allows requesting the services offered by a Web service in a transparent manner.

Figure 3.1 illustrates the placement of the stub and the skeleton layers in relation to the application in the service provider and the service requester.

The stub layer allows for calling a remote function in a Web service, using the SOAP protocol. The stub layer provides a C interface to the remote function, allowing the servicer requester to invoke a remote function, as if it was a local function. The stub layer is responsible for marshalling the arguments to the function and sending a request to the Web service in the SOAP message format. After the response have been received, the stub layer unmarshals the result and returns these values to the application.

The skeleton layer has the responsibility of listening for requests from the network for invoking a local C function in the servicer provider application. Whenever a SOAP message arrives from the network, the skeleton layer has the responsibility of identifying which function has been requested, followed by unmarshalling the arguments to the function. Next, the function is invoked, and after the execution of the function, the skeleton layer marshals the result, and sends it across the network to the service requester.

The tool that we design in this thesis has the responsibility of generating these two middleware layers. On an overall level, the functionality of WSTOOL can be divided into two main modules as illustrated in Figure 3.2. These modules are:

- The service provider generator.

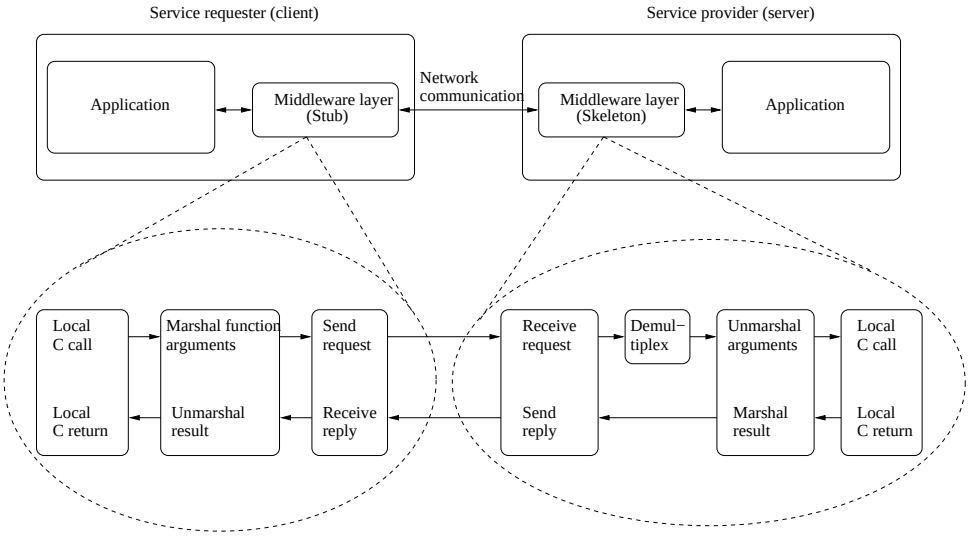


Figure 3.1: The two middleware layers (skeleton and stub) generated by WSTOOL, and the overall flow of the tasks carried out by these layers.

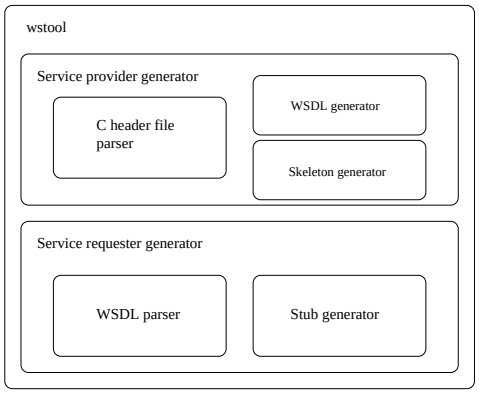


Figure 3.2: Overview of the two main modules of WSTOOL, and their sub-modules.

- The service requester generator.

The responsibility of the service provider generator is to generate the skeleton layer, that allows exposing an existing application as a Web service. To do so, the module must parse a C header file in the format we defined in Chapter 2, which specifies the data types and functions that should be part of the Web service. The information obtained by parsing this document is used to generate the appropriate skeleton layer, and to generate a WSDL specification, which specifies the protocol for communicating with the service.

The service requester generator has the responsibility of generating the stub layer. In order to do so, the module must parse a WSDL specification of the Web service. The information obtained by parsing this document is used to generate the appropriate stub layer.

As the skeleton and the stub layers are C code that will be compiled and linked with the application and running on the embedded platform, we will be referring to these layers in the following as the *target code*. The target code is the output of WSTOOL, and we choose to discuss its architecture, before discussing the design of the service provider generator module and the service requester generator module. This choice is made, since the architecture of the target code will greatly affect the design of these two modules.

3.2 Design Philosophy of Target Code

As discussed previously, the target code is the general term for the skeleton and stub layers generated by WSTOOL. These two layers will essentially consist of generated source code, that can be compiled and linked with the existing application, which is executing on the embedded platform. As previously described in Chapter 1, embedded systems are characterized by having constrained resources such as network, memory, and processing power. The introduction of skeleton and stub middleware layers introduces an overhead that is in conflict with these constraints. However, as discussed previously, these middleware layers aim to introduce a higher level of abstraction in the development of distributed systems which makes the overhead affordable.

The aim with our design of the target code is to reduce the overhead associated with introducing a middleware layer. We choose the two most important focus areas of the design of the target code as being:

- Minimizing CPU utilization.
- Minimizing memory consumption.

With respect to memory consumption, special attention is given to minimizing the use of dynamic memory allocation, which is not desired because of the performance penalties it implies, as we will discuss in Section 5.2.3. The two focus areas are the basis for the architectures and algorithms we present in this and the following chapters.

As described previously, the SOAP message format used to exchange messages in a Web service traditionally is encoded using XML, which introduces a large overhead in the network resources consumed. In Section 2.4 we introduced the notion of XML information sets, which are abstract descriptions of the information contained within XML documents. This notion allows for representing XML document in a different encoding. We do not design such a representation. However, we consider how an efficient interface to accessing the information in an XML information set can be created.

3.3 Architecture of the Skeleton Layer

In the following, we describe the architecture of the skeleton layer generated by WSTOOL. The architecture extends the ideas presented in DAT5 [2]. Figure 3.3 illustrates the architecture of the skeleton layer.

The purpose of the skeleton layer is to receive a request for invoking a function, unmarshal it, and invoke the requested C function. After the C function has returned, the skeleton layer marshals the values returned from the function, i.e. it marshals all the parameters with either the direction In-Out or the direction Out. The main difference between this architecture and the architecture presented in DAT5 [2] is the XML information set encoding parser/serializer. In DAT5 [2], the serialization of a SOAP message was constrained to being an XML document. In the new architecture, the representation can be any XML information set representation, as we discuss later.

A skeleton layer needs to be generated for each C function that is exposed as a Web service. Any skeleton layer generated by WSTOOL has some common tasks that always needs to be performed, namely sending and receiving data from a network, and parsing a XML information set. Therefore, we introduce a general layer that the skeleton layer relies on. This layer provides functionality to listen for incoming network connections, read requests, and parse the XML information set of the data contained in the request.

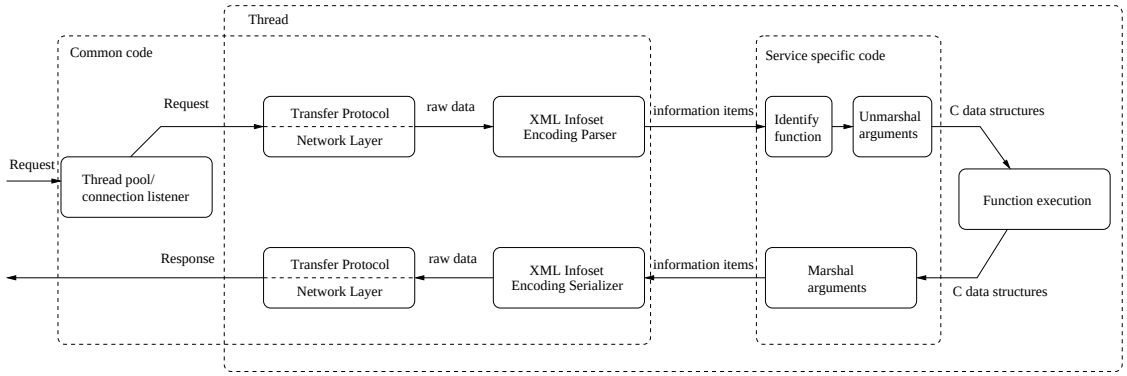


Figure 3.3: Architecture of the generated skeleton layer. Requests are received by a connection listener, and a thread is assigned to process the request. The request is unmarshalled followed by the execution of the proper C function. The outcome of the function is marshalled and send as a response.

Furthermore, this layer provides functionality for creating a serialization of a XML information set, and sending the response of handling the request back to the service requester. We refer to this general layer, that is the same for any skeleton layer generated by the tool, as *common code*.

Furthermore, any generated skeleton layer consists of functionality that marshals and unmarshals arguments to a given function. This functionality is specific for each function that a skeleton layer is generated for. Therefore, we refer to this part of the skeleton layer as the *service specific code*.

3.3.1 Common Code

The common code layer consists of three sub-layers:

- Thread pool layer.
- Network layer.
- XML information set layer.

The responsibility of the thread pool layer is to listen for incoming requests for the Web service and assign a thread to process the request. We choose a threaded architecture, in order to allow the skeleton code to process multiple requests concurrently. The responsibility of the network layer is to read a request from the network using some underlying protocol (such as HTTP). The raw data read from the network is passed to the XML information set layer. This layer parses the XML information set contained within the data (e.g. parsing XML documents if this is the chosen format for serializing the SOAP message). This parser produces XML information set information items which are passed to the service specific code for further handling. After the service specific code has finished its task, it outputs information items which the XML information set layer is responsible for serializing to raw data (e.g. serialize it to a XML document). This raw data is then sent as a response to the request, using the services offered by the network/transfer protocol layer. We describe the design of the layers in the common code further in Chapter 4.

3.3.2 Service Specific Code

The service specific code is responsible for unmarshalling an incoming request, executing requested functions, and marshalling responses. This is done by using the services offered by the common code layer. The service specific code receives XML information set information items from the common code layer, and initially identifies which function has been requested. Next, the arguments to the function are unmarshalled and the requested function is executed using these parameters. When the function has returned, the service specific code marshals the data returned by the C function. The result of the marshalling process is a stream of XML information set information items representing the SOAP message which must be sent as the response. These information items are passed to the common code layer, which is responsible for serializing the information items and sending them across the network. We describe the design of the service specific code further in Chapter 5.

3.4 Architecture of Stub Layer

Figure 3.4 illustrates the architecture of the stub layer generated by the tool. This architecture is similar to that of the skeleton layer. The purpose of the stub layer is to provide an interface to the service requester to invoke a function in a Web service as it was a local function. This is done by providing a stub function, which has the responsibility of marshalling the function call and the arguments supplied to it into a SOAP message.

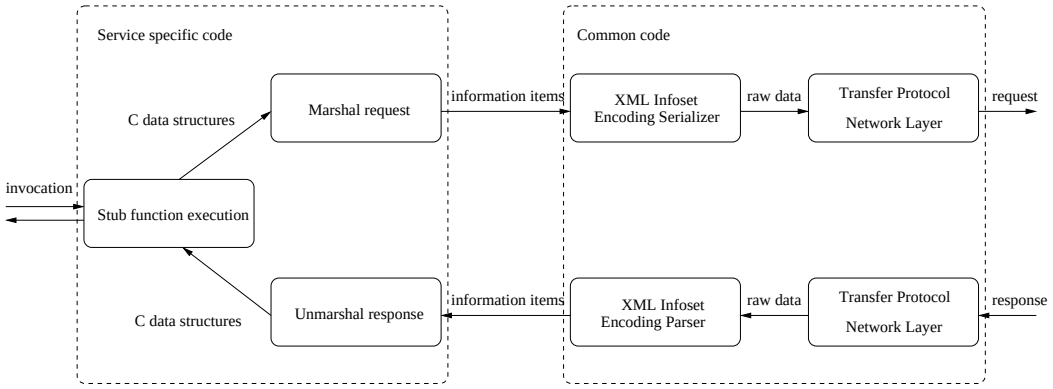


Figure 3.4: Architecture of the generated stub layer. The application representing a service requester invokes a stub function which marshals the parameters of the stub function and sends a request to the Web service. When the response from the Web service is received, it is unmarshalled and the stub function returns the result of the invocation.

A stub function is generated for each function in the Web service. As with the skeleton layer, the stub layer relies on a common code layer that provides similar services as the skeleton common code layer. The only difference is that the stub common code layer does not contain a thread pool layer. This is not needed since it is up to the application to implement threading if needed.

When invoked, the stub layer generates a SOAP message as a request to the Web service. This is done by generating a stream of information items representing the XML information set of this SOAP message. These information items are then passed to the XML information set layer, which serializes the XML information set, and a request is sent to the Web service using the services offered by the network layer. When the Web service responds, the XML information set layer produces a stream of information items

representing the SOAP message received as a response to the request. The stub function now unmarshals the data contained within the response and returns these as output to the function.

3.5 Summary

In this chapter, we have provided an overview of the key functionality of WSTOOL. We have identified two main modules of this tool: The service provider generator and the service requester generator. The service provider generator is responsible for parsing a C header file in the format discussed in Chapter 2 and generating a skeleton layer that allows exposing a set of C functions as a Web service. Furthermore, the module is responsible for generating a WSDL specification that describes the protocol for communicating with this Web service. The service requester module is responsible for parsing a WSDL specification of a Web service and generating a stub layer that allows requesting the services offered by this Web service in a transparent manner.

Furthermore, we have provided an overall design philosophy for the design of the middleware generated by WSTOOL, and described an overall design of the architecture of this middleware. We have introduced the notion of the service specific code and common code layers. The service specific code layer is the part of the skeleton and stub layers generated by WSTOOL, which is responsible for marshalling and unmarshalling requests/responses. As the name of this layer indicates, this functionality is specific to a given service provider/service requester. The common code layer is a general layer of functionality that the service specific code relies on, and which is always the same regardless of any service provider/service requester.

In the following chapters we will discuss the design of the different modules and layers introduced in this chapter in detail. In Chapter 4 we discuss the design of the common code layer. In Chapter 5 we discuss the design of the service specific code layer, and finally in Chapter 6 we describe the design of WSTOOL.

4

Design of Common Code Layer

In this chapter, we describe in detail the design of the common code layer. The common code layer is a static part of the middleware generated by WSTOOL, that always has to be included in an application, whether it is a service requester or service provider. The common code layer provides a set of services that the skeleton and stub layers rely on.

In the previous chapter, we identified three sub layers of the common code: The thread pool layer (only included in the skeleton layer), the network layer, and the XML information set layer. Our motivation in this chapter is to study which services should be offered by each of these layers, and to study how interfaces can be designed, which allow accessing these services efficiently.

4.1 Thread Pool Layer

In order to allow the skeleton layer to process multiple requests concurrently, the thread pooling strategy presented in DAT5 [2] is adopted. A short description of this strategy is included for completeness.

Requests are received by the skeleton layer using the services offered by the network layer. The 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.

This can be achieved by adopting a threaded processing model as shown in Figure 4.1. 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 runnable (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.

Different applications may have different needs for concurrent processing. For that reason it should be possible to tune the common code layer by adjusting the number of processing threads that should be started. This solution is a trade-off between efficiency

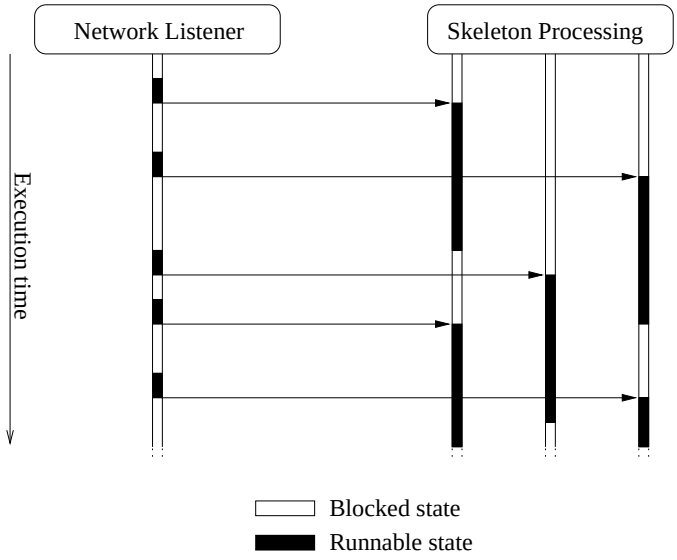


Figure 4.1: Threaded architecture of generated skeleton layer.

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 server.

4.2 Network Layer

The SOAP protocol does not specify which network protocol should be used to convey SOAP messages. We aim to provide a design of the common code which is not constrained to only using one network protocol.

The task of the network layer is to provide a consistent set of services to the XML information set layer and the service specific code to receive and send data to and from a network, without the upper layers having knowledge of the details of the network communication.

By providing a general design of the services that has to be offered by this layer, constraints are imposed on which network protocols can be used to supply these services. The focus of this thesis is not to provide a design of an interface general enough to accommodate as many network protocols as possible, but rather to provide an efficient interface and identify the needed properties of the network protocol, that allows for efficient exchange of SOAP messages in terms of CPU utilization and memory consumption.

We identify the set of services that are needed by the upper layers as:

- Connection management.
- Sending and receiving data.
- Handling retransmission of lost messages and filtering out duplicate messages.
- Provide error checking of messages.

It should be noticed that we assume that an implementation of this layer is connection-oriented and reliable. By introducing these constraints, a simple interface can be provided to the upper layers, since they will not need to handle network specific details. They

should only be concerned with reading and sending data. Furthermore an at-most-once invocation semantics will always be achieved by introducing these constraints.

4.2.1 Connection Management

We provide three basic functions to perform the task of accepting incoming connections, establishing outgoing connections and closing connections. Throughout the remainder of this chapter, we will describe the interfaces of the functions supported by the common code layer, by stating the C prototype for the function.

C return type	C function name
`nw_connection_t`	`nw_accept_connection();`

Figure 4.2: The `nw_accept_connection` function, providing a general interface to accepting an incoming connection.

C return type	C function name

`typedef enum {`
`  WS_NO_NETWORK_CONNECTION,`
`  WS_NO_SUCH_URL,`
`  WS_TIME_OUT,`
`  WS_SUCCESS`
`} status_t;`
	`nw_connect(nw_connection_t *c_ptr,`
	`  char URL[]);`

Figure 4.3: The `nw_connect` function, providing a general interface to establish an outgoing connection.

The functions `nw_accept_connection`, and `nw_connect` are illustrated in Figure 4.2 and Figure 4.3 respectively. The function `nw_accept_connection` is responsible for listening for a request from the network, and to negotiate the establishment of such a connection. When the connection has been established, the function returns a `nw_connection_t` data type that represents this connection. The structure of this data type is undefined, as it depends on the specific implementation of the network layer. The purpose of the data structure is to provide an identification of a connection, and to provide status information about the connection if necessary.

The function `nw_connect` establishes a connection to a location on the network that is addressable by a URL. It is a requirement by the SOAP specification that any underlying network protocol used to convey SOAP messages should support addressing network locations by URLs. E.g. if the underlying network is HTTP, FTP or SMTP, examples of URLs are "http://ciss.dk/webservice", "ftp://ciss.dk/webservice" and "mailto:webservice@ciss.dk" respectively. This function returns a status indicating the outcome of attempting to establish the connection. The possible outcomes are:

- `WS_NO_NETWORK_CONNECTION` - no connection to the network was available.
- `WS_NO_SUCH_URL` - no connection could be established to the network location addressed by the requested URL.
- `WS_TIME_OUT` - the request for establishing the connection timed out.

- `WS_SUCCESS` - the connection was successfully established.

C return type	C function name
`void`	`nw_close_connection(nw_connection_t *c_ptr);`

Figure 4.4: The `nw_close_connection` function, providing a mechanism for the upper layers to announce that they have finished using a connection.

Furthermore, the function `nw_close_connection` is provided to allow the upper layers to indicate that they are finished using the connection and that the network layer can close the connection. The interface of this function is illustrated in Figure 4.4.

C return type	C function name

`typedef enum`
`{`
`  WS_SUCCESS,`
`  WS_CONNECTION_LOST`
`} nw_write_status_t;`
	`nw_write_buffer(nw_connection_t *c_ptr,`
	`  char buffer[],`
	`  unsigned int size);`

Figure 4.5: The `nw_write_buffer` function, providing a general interface to writing a specific amount of data to a network connection.

C return type	C function name

`typedef struct {`
`  enum {`
`  WS_SUCCESS,`
`  WS_CONNECTION_LOST`
`  } status;`
`  unsigned bytes_read;`
`} nw_read_status_t;`
	`nw_read_buffer(nw_connection_t *c_ptr,`
	`  char buffer[],`
	`  unsigned int size);`

Figure 4.6: The `nw_read_buffer` function, providing a general interface to reading a specific amount of data from a network connection.

4.2.2 Sending and Receiving Data

We aim to design an interface that allows for a small CPU utilization and memory consumption when sending and receiving data. As discussed in DAT5 [2], in order to allow for this, it is desirable that the network layer provides the functionality to send and receive a stream of data without prior knowledge about the total size of the SOAP messages being transmitted. This means that the network layer should provide a mechanism for receiving a request in small parts at the time, and that it should support sending the response in small parts of the time. Provided that the upper layers allows for streaming as well, which we will discuss later, this allows for the service specific code to receive part of a request/response into a buffer, and then unmarshal this part. When processing of this part has completed and converted into a efficient C representation, the buffer can be

emptied and a new part of the request can be received and unmarshalled. Thereby, the design goal of minimizing memory consumption is achieved. Similarly, when marshalling the response/request the service specific code performs marshalling of small parts of the time into a buffer, followed by flushing the buffer and sending its contents across the network and then continuing to marshal the next part.

In Figure 4.5 and Figure 4.6 the two functions `nw_write_buffer` and `nw_read_buffer` are illustrated.

These two functions provides a general abstraction of sending and receiving data from a network connection. The function `nw_write_buffer` provides a interface of sending a specific amount of data from a buffer. The function takes three parameters:

- `c_ptr` - the connection that should be used to send the data.
- `buffer` - the buffer that contains the data to be sent.
- `size` - the size of the buffer containing the data to be sent.

The function can return two values

- `WS_SUCCESS` - the data was successfully written to the connection.
- `WS_CONNECTION_LOST` - an error occurred while writing to the connection.

The function of writing to the connection was either successful or not. If the function was unsuccessful, all the upper layers should stop processing the current task. The upper layers have no further responsibilities when the network fails, besides signaling this to the application code as described in the previous chapter.

The function `nw_read_buffer` has a similar interface. However, the parameter `buffer` now indicates the buffer that should be filled with the data arriving from the network. Furthermore, the function returns two values: The outcome of reading (`WS_SUCCESS` or `WS_CONNECTION_LOST`), and the value `bytes_read`. This value indicates how many bytes were read from the connection. If this value is smaller than the requested amount of bytes to be read, it indicates that no more data is available from the network.

4.3 An Instance of the Network Layer

Having designed a general interface of the network layer, we now provide the design of an instance of this layer. Currently the W3C has only formally specified the well-established HTTP network protocol to be used to convey SOAP messages. Therefore, we choose to use this as the network layer. This protocol is based on the TCP protocol, which provides a range of desirable features. The HTTP 1.1 protocol has the following properties that are required by our design of the network layer:

- Supports the concept of connections (handled by TCP).
- Can handle retransmission and filtering of duplicate messages (handled by TCP).
- Allows for streaming of data.

In the following, we investigate how an instance of the network layer can be designed using the HTTP protocol.

4.3.1 Chunked Transfer Encoding

Our previous work presented in DAT5 [2], suggested the use of chunked transfer encoding in the HTTP 1.1 protocol as an optimization, to achieve the capability of streaming data without prior knowledge of its total size.

```
POST /WebService HTTP/1.1
Host: ciss.dk
Content-Type: application/soap+xml
Transfer-Encoding: chunked

50
<!-- Chunk 1 -->
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
 <env:Body>
87
<!-- Chunk 2 -->
  </ciss:ws_wild2>
   <ciss:uname xmlns:ciss="http://ciss.dk/embeddedService/">
    <!--more data -->
  </ciss:uname>
2E
<!-- Chunk 3 -->
 </env:Body>
</env:Envelope>
0
```

Example 4.1: An example of sending a SOAP message serialized as an XML document, using the chunked encoding offered by the HTTP 1.1 protocol.

Example 4.1 illustrates how a SOAP message serialized as an XML document can be sent using the chunked encoding offered by the HTTP 1.1 protocol.

The document is sent in three parts (called chunks). The beginning of each of these chunks are illustrated in the example as XML comments (e.g. "`<!- Chunk 1 ->`"). Before each of these chunks, the HTTP 1.1 protocol specifies that the length of the upcoming chunk should be specified in the hexadecimal system. The first chunk is 120 bytes long (represented as 50 in hexadecimal). The next chunk is 135 bytes long (87 in hexadecimal), and the final chunk is 42 bytes long (2E in hexadecimal). When the document ends, a "0" is sent to mark this.

4.3.2 Reading and Writing Data

In order to handle CPU and memory resources efficiently when writing and reading network data, we provide a special buffering strategy. Remembering the interface of the `nw_read_buffer` and `nw_write_buffer` functions, we saw that these could be used to send/read any number of bytes of data over the network. As we will illustrate later, these functions will often be invoked by the service specific code with requests for sending/reading small amounts of data. If not handled carefully, this can be a source of inefficiency. In many operating systems, sending/and reading from the network is associated with a system call, which can introduce a large overhead if too many system calls are needed.

To avoid this overhead, we provide an internal buffer for each established HTTP connection. Figure 4.7 illustrates the buffering strategy. When a request is made to read a given amount of data from the network, a large amount is read from the network and placed in the internal buffer. Then the requested amount of data is copied to the external buffer specified by the `buffer` argument to the read function. Next time a read function is requested, the data is copied from the internal buffer. This process continues until the buffer is empty, and then a new amount of data is read from the network.

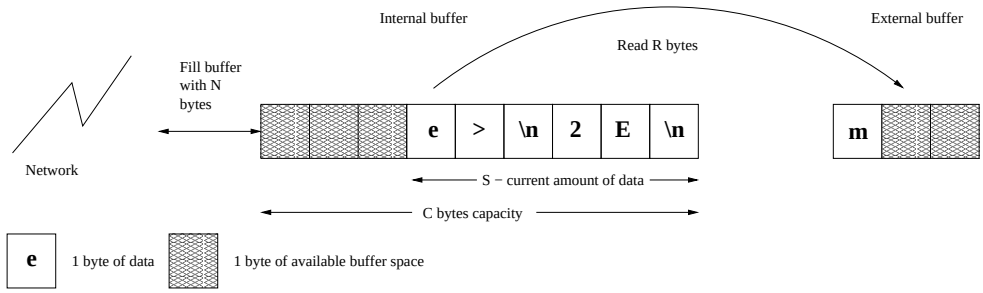


Figure 4.7: An illustration of the buffering strategy applied when data is read from the network using the `nw_read_buffer` function. A certain amount of data is read from the network and buffered. Whenever calls are made to the `nw_read_buffer` function, data is served from this buffer instead of reading it directly from the network

Figure 4.8 illustrates how the `nw_read_buffer` function functions, and how the internal buffer is handled. The figure is simplified, since it does not illustrate how the chunked encoding of the network data is handled. This is left out, since the relevance of the figure is to illustrate how the buffers are handled.

The figure introduces several notions, of which `N` is special. Whenever the internal buffer is empty, a given amount of data, (`C`), is read from the network. Often, the actual amount of data read will be smaller than requested, since the data read from the network also includes the formatting used in the chunked transfer encoding. Therefore, the actual amount of data read from the network is a little smaller. The value `N` defines this value.

Figure 4.9 illustrates the handling of the internal buffer when the `nw_write_buffer` function is invoked. The same method is applied. Whenever data is requested to be written to the network, it is saved to an internal buffer. When the buffer is full, its content is HTTP chunk encoded, and sent over the network, and the buffer is emptied.

It is important to note that the algorithms for handling the internal buffer have been optimized to handle request of reading/writing small amount of the data. Therefore the size of the internal buffer (`C`) is set to a value that is assumed to be much larger than the amount of data that typically will be requested to be sent/received (`R`). When we describe the design of the service specific code, it will be illustrated that this will be the case.

4.4 XML Information Set Layer

The SOAP specification does not specify how a SOAP message should be serialized. As discussed in Section 2.4, SOAP messages are formally specified as XML information sets, which can be serialized in any manner, as long as the serialization keeps the properties of the XML information set. In the following we describe the design of the set of services offered by the XML information set layer in the common code. This layer has the following responsibilities:

- Read raw data from the network, using the underlying network layer.

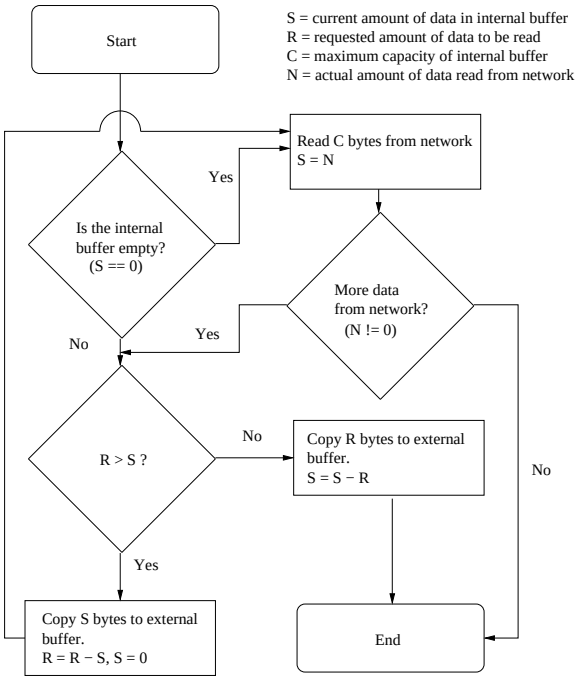


Figure 4.8: Flowchart illustrating the implementation of the `nw_read_buffer` function, using a buffering strategy.

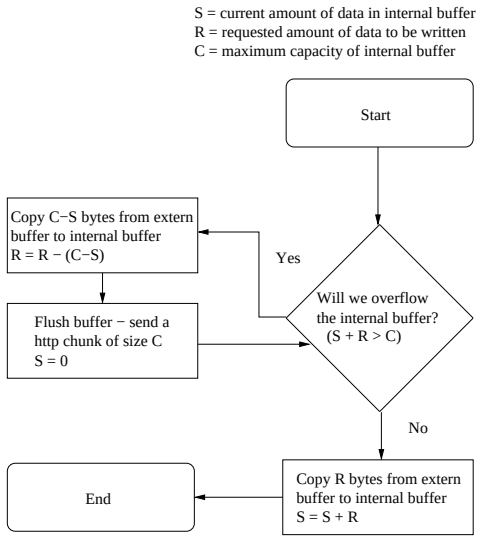


Figure 4.9: Flowchart illustrating the implementation of the `nw_write_buffer` function, using a buffering strategy.

- Parse the received data and produce the XML information set contained within the data. Then hand this over to the service specific code layer.

- Serialize XML information sets received from the service specific code layer and send it over the network.

4.4.1 Designing an Efficient Interface

In this section, we are concerned with the design of an efficient interface that allows for passing the information items contained within the parsed data, to the service specific code. The XML information set specification does not state anything about how a XML information set should be represented or how the information items within a XML information set should be accessed through some standard interface. However, in the XML information set specification, a XML information set is described as a tree structure.

In Figure 4.10 we illustrate a tree representation of the XML information set of the SOAP message illustrated in Example 2.14.

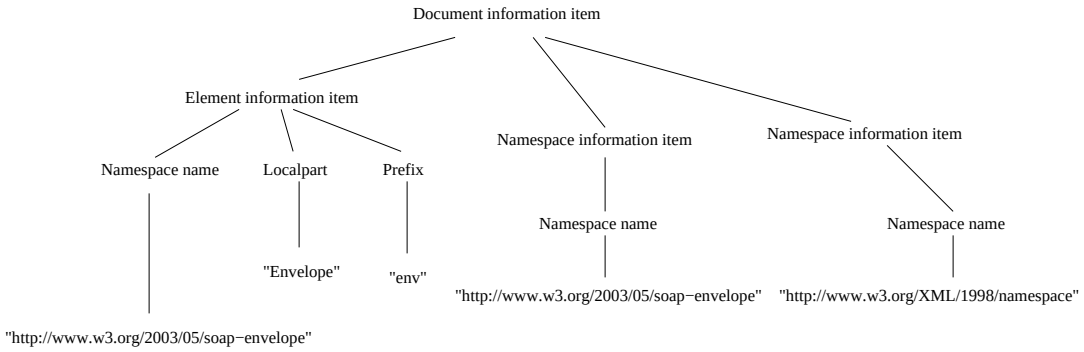


Figure 4.10: A tree representation of the XML information set contained within the SOAP message illustrated in Example 2.14.

Such a representation is not desirable, because passing an entire tree to the service specific code requires a large amount of memory. Furthermore, this approach is not in line with our desire to process small parts of the XML information set at the time.

In order to allow for an efficient interface it is desirable that small parts of the information items of the XML information set can be passed to the service specific code at the time. Therefore, we introduce the idea of providing an interface where tokens describing the information items contained within a XML information set is streamed to the service specific code. This is illustrated in Figure 4.11.

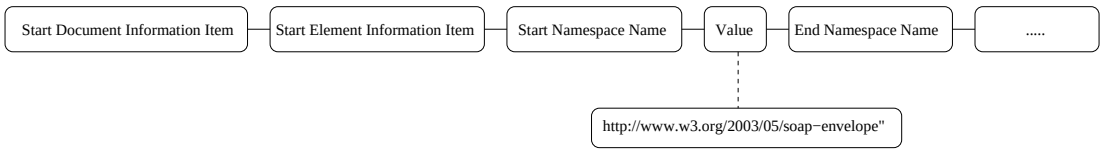


Figure 4.11: A streamed representation of the XML information set contained within the SOAP message presented in Example 2.14.

The idea is that tokens are passed to the service specific code stating that an information item has been started, and if this information item contains other information items, tokens are passed describing the content of these.

Using this idea, we need to define how these tokens should be represented. Representing the tokens as strings is an inefficient method. In order to provide a more efficient interface to access the data contained within the XML information set of a SOAP message, we do the following observations about the needed information:

- Only the information contained within element information items and attribute information items is needed.

- For each element information item, only the local part, character information item, and the namespace name is needed.
- For each attribute information item, only the local part, value information item, and the namespace name is needed.
- Given that the service specific skeleton and stub layers can only handle a limited set of remote procedure calls, the set of information items that the service specific code will understand is limited.

The XML information set specification provides a long list of information items that a XML information set can contain. However, not all these information items can appear in a valid SOAP message—e.g. a document type definition information item can not appear in a SOAP message, although it is part of the XML information set specification. We observe that the information items that are of interest in a SOAP message is the element information items and the attribute information items.

Furthermore, for each element information item, the only information of interest is the local part, namespace name and the character information item. Similarly for an attribute information item the only information of interest is the local part, namespace name and the value information item. Therefore, the interface to access the tokens representing the information items only needs to support passing these informations.

Finally, an important observation is that the skeleton and stub layers will only need to recognize a limited set of information items. We illustrate this by a simple example.

Web service C Specification	Permissible Structure of SOAP Message

`void ws_simple(int * in_out_parm);`	`<?xml version='1.0' ?>`
	`<env:Envelope xmlns:env=`
	`"http://www.w3.org/2003/05/soap-envelope" >`
	`<env:Body>`
	`<ciss:ws_simple xmlns:ciss=`
	`"http://ciss.dk/embeddedService/">`
	`<ciss:in_parm>60</ciss:in_parm>`
	`</ciss:ws_simple>`
	`</env:Body>`
	`</env:Envelope>`

Figure 4.12: A simple C function exposed as a Web service, and the permissible structure of the SOAP message requesting to invoke this function.

Figure 4.12 illustrates a simple C function exposed as a Web service, and the permissible structure of the SOAP message representing a request for invoking this function. This message only contains element information items and only contains four element information items which are illustrated in Figure 4.13.

4.4.2 Announcing Information Items

The fact that the stub and skeleton layers only supports marshalling/unmarshalling a limited set of SOAP messages can be exploited to create a efficient interface to access the information contained within the XML information set of a SOAP message.

Local part	Namespace Name	Character Inf. Item
Envelope	http://www.w3.org/2003/05/soap-envelope	n/a
Body	http://www.w3.org/2003/05/soap-envelope	n/a
ws_simple	http://ciss.dk/embeddedService/	n/a
in_parm	http://ciss.dk/embeddedService/	60

Figure 4.13: The four element information items and their namespace name and character information items contained in the XML information set of the SOAP message illustrated in Figure 4.12.

We introduce an interface, allowing the service specific code to assigning unique IDs to each valid information item. This list of possible information items can be generated while generating the stub and skeleton layers.

We illustrate two functions that enable the service specific code to announce valid information items. This is illustrated in Figure 4.14 and Figure 4.15, showing the functions `xis_announce_attribute_ii` and `xis_announce_element_ii` respectively.

C return type	C function name
`void`	`xis_announce_attribute_ii(char *local_name, char *namespace_name, unsigned id);`

Figure 4.14: The `xis_announce_attribute_ii` function, which allow for announcing the knowledge about an attribute information item to the XML information set layer.

C return type	C function name
`void`	`xis_announce_element_ii(char *local_name, char *namespace_name, unsigned id);`

Figure 4.15: The `xis_announce_element_ii` function, which allow for announcing the knowledge about an element information item to the XML information set layer.

These functions enable the service specific code to announce all the information items to the XML information set layer that it will understand. When the XML information set layer parses a stream of raw data, and extracts the XML information set from this, only IDs are needed to be passed to the service specific code. This achieves the design goal of minimum CPU utilization and memory consumption, since no advanced tree structures needs to passed between the layers. The `xis_announce_attribute_ii` and `xis_announce_element_ii` functions are expected to be carried out during the initialization of the stub and skeleton layer, and afterwards the XML information set layer uses this information to serialize/de-serialize encodings of XML information sets.

4.4.3 Retrieving Information Items

Figure 4.16 illustrates the function `xis_get_next_ii`. This function allows retrieving the next information item from the XML information set layer. The function returns the id of the next token in the XML information set being parsed, and a flag `type` that indicates whether this token represents the start of an information item or the end of an information

item. As input, the function takes the arguments `c_ptr` and `status_ptr`. `c_ptr` identifies the network connection from where the XML information set layer can read the data being parsed, and `status_ptr` is the structure returned from the network layer signaling the outcome of reading from the network connection.

C return type	C function name

`typedef struct {`
`  unsigned id;`
`  enum {START,`
`  END} type;`
`} information_ii_t;`
	`xis_get_next_ii(nw_connection_t *c_ptr,`
	`  nw_read_status_t *status_ptr);`

Figure 4.16: The `xis_get_next_ii` function, that allows retrieving the next information item from the XML information set layer.

Since the service specific code only announces a limited set of information items, the XML information set layer may encounter information items that have not been announced. The XML information set layer needs to be able to signal this to the service specific code. Therefore, a set of special tokens are reserved to indicate this. These tokens are listed in Table 4.1.

Constant name	Description
`UNKNOWN_ELEMENT_II`	A element information item that had not been announced was encountered
`UNKNOWN_ATTRIBUTE_II`	A attribute information item that had not been announced was encountered
`VALUE_CHARACTER_II`	Either a value or character information item have been retrieved
`NO_MORE_II`	No more information items are contained within the XML information set

Table 4.1: A reserved set of tokens that can be produced by the XML information set layer to indicate that a special information item has been encountered.

The two tokens `UNKNOWN_ATTRIBUTE_II` and `UNKNOWN_ELEMENT_II` are reserved to indicate to the service specific code that unknown information items have been encountered. This can occur if the request contains a SOAP header which the skeleton layer does not have any knowledge about. SOAP headers may have an attribute `mustUnderstand` set, which determines whether a SOAP node must understand the header or not. If a header is is not marked `mustUnderstand`, the skeleton layer can simply ignore it. Otherwise, it is considered a fatal error, and a SOAP fault must be sent, according to the SOAP protocol.

If unknown information items are encountered in the SOAP body, it is an error and the skeleton layer must stop processing the body, and generate a SOAP fault.

Furthermore, two other tokens are reserved for special purposes. The token named `VALUE_CHARACTER_II` indicates that either a value information item or a character information item has been encountered (as part of an attribute information item and an element information item respectively). Whether the token is a value or a character information item can be established by the preceding encountered token, e.g. if the preceding token was an element information item start token, the current token must be a character information item. In the case of encountering such a token, an additional function should

C return type	C function name
char *	xis_get_next_character_value_ii(nw_connection_t *c_ptr, nw_read_status_t *status_ptr);

Figure 4.17: The `xis_get_next_character_value_ii` function, which allows retrieving the next character or value information item from the XML information set layer.

be provided to retrieve the value of this information item. The interface of this function `xis_get_next_character_value_ii` is illustrated in Figure 4.17. This function also takes the arguments `c_ptr` and `status_ptr`.

Finally the token `NO_MORE_II` is reserved for indicating that no more information items are available from the parsed data, i.e. the XML information set layer has finished parsing the entire XML information set.

4.4.4 Marshalling an XML Information Set

The final part of the interface that needs to be specified is how data is marshalled using the XML information set layer.

In order to marshal an information item, only the id of an announced token must be passed to the XML information set layer. Therefore, only two additional functions are needed, namely the functions `xis_write_ii`, and `xis_write_character_value_ii`. The interfaces of these functions are illustrated in Figure 4.18 and Figure 4.19 respectively.

C return type	C function name
void	xis_write_ii(unsigned id, nw_connection_t *c_ptr, nw_write_status_t *status_ptr);

Figure 4.18: The `xis_write_ii` function, which tells the XML information set layer to serialize a certain information item.

C return type	C function name
void	xis_write_character_value_ii(char *value, nw_connection_t *c_ptr, nw_write_status_t *status_ptr);

Figure 4.19: The `xis_write_character_value_ii` function, which tells the XML information set layer to serialize either a character or value information item.

The `xis_write_ii` function requests the XML information set layer to serialize a certain information item. The `xis_write_character_value_ii` function requests serialization of either a value information item or a character value information item.

4.4.5 An Instance of the XML Information Set Layer

The details of our implementation of the XML information set layer will be discussed further in Chapter 7. We do not discuss the design of an implementation of the XML information set layer in this chapter, as we do not design our own XML parser. Rather, we rely on an existing XML parser library, called *Expat* [7].

4.5 Summary

In this chapter, we have provided a design of the common code layer, which is a layer providing general functionality for the stub and skeleton layers generated by WSTOOL.

The common code layer has been divided into three sub layers, and the services offered by these different layers have been discussed. A thread pool layer has been briefly discussed, which follows the design provided in our previous work presented in DAT5 [2].

A general network layer has been designed, which allows the skeleton and stub code layers to transmit data across a network in a transparent manner without specific knowledge about the network protocol used. Our design has introduced a set of restrictions on the implementation of the network layer, since assumptions on the services offered by the protocol have been introduced. These assumptions have been introduced to allow for an efficient network exchange. Furthermore, we have presented an instance of the network layer by the use of the HTTP protocol which fulfills these assumptions.

Finally, we have provided the design of an XML information set layer, which allows accessing the information contained within a XML information set in a general manner. An efficient interface to access this information has been provided.

5

Design of Service Specific Code Layer

In this chapter, we discuss the design of the service specific code of the skeleton and stub layers. The service specific code has two main responsibilities, namely performing marshalling and unmarshalling. Our motivation in this chapter is to investigate how marshalling and unmarshalling code can be generated algorithmically. We approach this problem by studying how context-free grammars can provide an abstraction that allows us the creation of unmarshalling code in an algorithmically simple manner.

Next, we study how semantic actions can be carried out during unmarshalling efficiently, by using specific knowledge about the structure of the data being unmarshalled.

We end the chapter by discussing how marshalling can be carried out algorithmically.

5.1 Unmarshalling Using Context-free Grammars

In our prior work presented in DAT5 [2], we introduced the idea of generating a state machine to handle the unmarshalling of requests and responses. An instance of a simple state machine was implemented, however it was only able to accept a single type of messages. In this section we formalize the creation of such state machines. We do this by providing a method for generating context-free grammars [1], describing the syntax of requests/responses that a specific instance of a skeleton/stub layer will understand. Our motivation for introducing context-free grammars, is to introduce an abstraction that allows us generating unmarshalling code in an algorithmic simple manner. Many *parser generator tools* exists, which allows transforming the specification of a context-free grammar into a parser.

5.1.1 Context-free Grammars

Context-free grammars provides a mechanism to describe a *language*. Formally, a language is a set of *strings*, and a string is a finite sequence of *symbols* taken from a finite *alphabet*. A context-free grammar has a set of *productions* as illustrated below:

$$symbol \rightarrow symbol \quad symbol \quad \ldots \quad symbol$$

Each production consists of zero or more symbols on the right-hand side. Each symbol is either a *terminal*, which means that it is a *token* from the alphabet, or it is a *nonterminal*. Nonterminals appear on the left-hand side in one or more of the productions of the context-free grammar, and are not in the alphabet.

Conceptually, the task of unmarshalling a request or response is the task of parsing an input string using a context-free grammar to describe a language that defines the permissible syntactic structure of a SOAP message. The language is defined by the set of permissible SOAP messages that can be sent as requests or responses from a given Web service. The alphabet of such a language is the set of possible XML information set information items that can be contained in this set of SOAP messages.

We have already specified a design that allows the service specific code to announce the set of information items that it will understand, i.e. to announce that alphabet of the language describing the syntax of the SOAP messages it will understand. This is done by the operations `xis_announce_attribute_ii` and `xis_announce_element_ii` provided by the XML information set layer.

We start this section by illustrating an example of how a context-free grammar can be constructed, which defines a language that describes the syntactic structure of the SOAP message described in Figure 4.12. Example 5.1 illustrates such a context-free grammar. Throughout this chapter, when we present a context-free grammar we mark terminals in **bold**, to simplify the reading.

$$
\begin{aligned}
\textit{Start} &\rightarrow \textit{Envelope} \\
\textit{Envelope} &\rightarrow \textbf{Envelope}_{\textbf{Start}}\ \ \textit{Body}\ \ \textbf{Envelope}_{\textbf{End}} \\
\textit{Body} &\rightarrow \textbf{Body}_{\textbf{Start}}\ \ \textit{ws\_simple}\ \ \textbf{Body}_{\textbf{End}} \\
\textit{ws\_simple} &\rightarrow \textbf{ws\_simple}_{\textbf{Start}}\ \ \textit{in\_parm}\ \ \textbf{ws\_simple}_{\textbf{End}} \\
\textit{in\_parm} &\rightarrow \textbf{in\_parm}_{\textbf{Start}}\ \ \textbf{VALUE\_CHARACTER\_II}\ \ \textbf{in\_parm}_{\textbf{End}}
\end{aligned}
$$

Example 5.1: A context-free grammar derived from the example SOAP message illustrated in Figure 4.12. This context-free grammar defines a language that describes the syntax of this SOAP message.

This example uses the set of XML information set information items described in Figure 4.13 as its alphabet. When the SOAP message illustrated in Figure 4.12 is parsed using this grammar, the parse tree illustrated in Figure 5.1 is derived.

Observing this parse tree, it can be seen that the function `ws_simple` has been requested, and that the formal argument `in_parm` has the actual value of 60. By abstracting the task of generating code for unmarshalling SOAP messages into the task of creating a context-free grammar accepting valid SOAP messages, the task of algorithmically generating unmarshalling code becomes relatively simple. Furthermore, it should be noticed using this approach, the parser reads one information item from the XML information set layer at the time (i.e. terminals). This is in line with our design of the XML information set layer and the instance of the network layer discussed in Chapter 4, since they are optimized for performing small reads at the time.

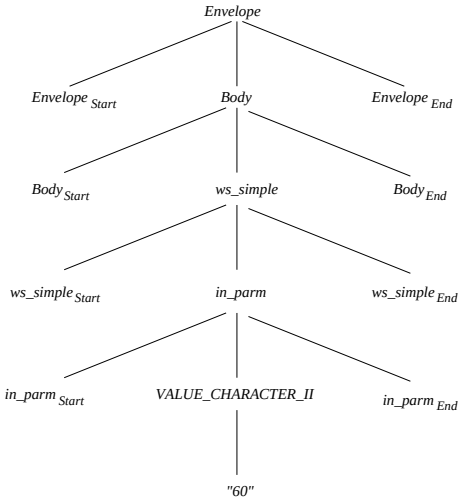


Figure 5.1: The parse tree for the grammar described in Example 5.1, when the SOAP message in Figure 4.12 is used as input string

5.1.2 General Structure of Context-free Grammars for Unmarshalling

When constructing context-free grammars, there are some considerations to be made. A grammar can be *ambiguous*, if it is possible to generate different parse trees for the same input string. When constructing a context-free grammar defining the language that the unmarshalling part of the service specific code will use to parse requests/responses, we need to ensure that we do not generate ambiguous grammars.

To get an understanding of this problem, we illustrate an ambiguous grammar in Example 5.2. When the input string is "$num + num + num$", two different parse trees can be derived, as illustrated in Figure 5.2.

$$
\begin{aligned}
Start &\rightarrow E \\
E &\rightarrow E + E \\
E &\rightarrow \textbf{num}
\end{aligned}
$$

Example 5.2: A simple ambiguous context-free grammar that has a alphabet containing the two terminal symbols + and num.

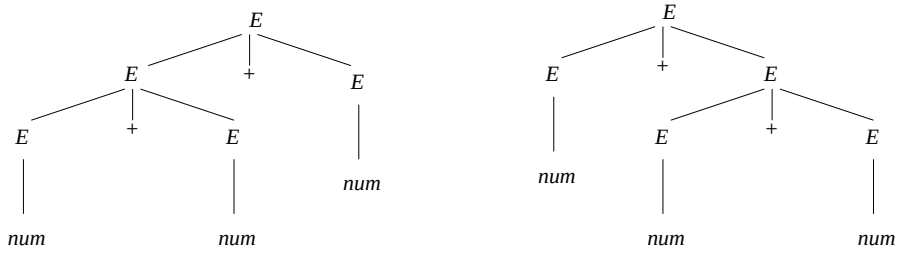


Figure 5.2: Two different parse trees derived from the grammar introduced in Example 5.2, when the input string is "$num + num + num$".

$$
\begin{aligned}
\mathit{Start} &\rightarrow \mathit{Envelope} \\
\mathit{Envelope} &\rightarrow \mathbf{Envelope_{Start}}\ \mathit{Body}\ \mathbf{Envelope_{End}} \\
\mathit{Body} &\rightarrow \mathbf{Body_{Start}}\ \mathit{request}\ \mathbf{Body_{End}} \\
\mathit{request} &\rightarrow \mathit{function}1 \\
\mathit{request} &\rightarrow \ldots \\
\mathit{request} &\rightarrow \mathit{functionN} \\
\mathit{function}1 &\rightarrow \mathbf{function1_{Start}}\ \mathbf{parm1\_1_{Start}}\ \mathit{type}\ \mathbf{parm1\_1_{End}}\ \ldots \\
&\quad\ \mathbf{parm1\_N_{Start}}\ \mathit{type}\ \mathbf{parm1\_N_{End}}\ \mathbf{function1_{End}} \\
\mathit{functionN} &\rightarrow \mathbf{functionN_{Start}}\ \mathbf{parmN\_1_{Start}}\ \mathit{type}\ \mathbf{parmN\_1_{End}}\ \ldots \\
&\quad\ \mathbf{parmN\_N_{Start}}\ \mathit{type}\ \mathbf{parmN\_N_{End}}\ \mathbf{functionN_{End}}
\end{aligned}
$$

Example 5.3: The general structure of any context-free grammar generated for unmarshalling requests.

Example 5.3 illustrates the general structure of any context-free grammar we generate for unmarshalling requests. The structure for context-free grammars generated for unmarshalling responses from a Web service will be similar, and therefore we do not describe it here. The example illustrates that any context-free grammar used for unmarshalling a request will contain productions to describe the SOAP envelope and the SOAP body. For now, we wait describing how SOAP headers are handled for the sake of clarity. One production describes the content of the SOAP body, namely the *request* production. The *request* is a set of productions that declares all the functions allowed in a SOAP message.

We now introduce the notion of FIRST(γ). Given a string γ of terminal and nonterminal symbols, FIRST(γ) is the set of all terminal symbols that can begin any string derived from γ. As an example using the grammar in Example 5.2, let $\gamma = E + E$. Any string of terminal symbols derived from γ must start with the terminal symbol num, i.e. FIRST$(E + E) = num$.

When observing the general grammar for unmarshalling in Example 5.3, we observe that the only set of productions with the same left-hand side is the set of productions with the nonterminal *request* on the left-hand side. All these productions have different FIRST sets (i.e. FIRST$(function1) = function1_{Start}$ and FIRST$(functionN) = functionN_{Start}$). As all these FIRST sets are different, an ambiguous grammar can never be produced. Consequently, context-free grammars using this structure can be used to unambiguously parse valid SOAP messages.

5.1.3 Handling Data Types

The grammar illustrated in Example 5.3 describes a general grammar used for describing a language that describes the allowed syntax of requests. However, this grammar does not describe the syntax of data types. In the following, we illustrate how data types can be described using context-free grammars.

In Chapter 2 we defined a mapping of the C data types to XML, and vice versa. Using this mapping, a general context-free grammar can be devised for describing the syntax of these types, and used for unmarshalling requests and responses.

All simple C data types can described by the terminal $VALUE\_CHARACTER\_II$ as they contain no markup. Figure 4.12 illustrates the SOAP message representing a request for the C function `ws_simple`, which takes the argument `in_parm` of the C data type `int`.

The XML information set information item representing the value of the actual argument (60) is a value information item, which we represented by the token `VALUE_CHARACTER_II`.

$$
\begin{array}{rcl}
\textit{int} & \rightarrow & \textbf{VALUE\_CHARACTER\_II} \\
\textit{float} & \rightarrow & \textbf{VALUE\_CHARACTER\_II} \\
\textit{struct\_a\_struct} & \rightarrow & \textbf{member1}_{\textbf{Start}} \ \textit{int} \ \textbf{member1}_{\textbf{End}} \\
 & & \textbf{member2}_{\textbf{Start}} \ \textit{float} \ \textbf{member2}_{\textbf{End}} \\
 & & \textbf{member3}_{\textbf{Start}} \ \textbf{member1}_{\textbf{Start}} \ \textit{int} \ \textbf{member1}_{\textbf{End}} \ \textbf{member3}_{\textbf{End}}
\end{array}
$$

Example 5.4: A context-free grammar that defines a language describes the syntax of the SOAP representation of the derived C data-type `a_struct` illustrated in Figure 2.1.

However, derived types are handled differently. In Example 5.4 we illustrate a grammar that defines a language that describes the syntax of the C struct type `a_struct`, which was shown in Figure 2.1. Two productions, *int* and *float*, are provided to describe the syntax of the simple C data types with the same names. Furthermore, the production *struct_a_struct* is provided, which describes the structure of the derived C data type (a struct) with the same name. It should be noticed that this C data type has the member (`member3`), which itself is a struct. This data type has no name, and therefore it is declared embedded into the *struct_a_struct* production describing the `a_struct` type.

5.1.4 Handling Array Types

Producing a grammar, which defines the syntax of array types needs to be handled in a special manner, since the amount of items in an array is unknown. To handle this, we use recursion.

$$
\begin{array}{rcl}
\textit{int} & \rightarrow & \textbf{VALUE\_CHARACTER\_II} \\
\textit{int\_array} & \rightarrow & \textit{int\_array} \ \textbf{item}_{\textbf{Start}} \ \textit{int} \ \textbf{item}_{\textbf{End}} \\
\textit{int\_array} & \rightarrow & \textbf{item}_{\textbf{Start}} \ \textit{int} \ \textbf{item}_{\textbf{End}}
\end{array}
$$

Example 5.5: A context-free grammar that defines a language that describes the syntax of an array of integers.

Example 5.5 illustrates how a grammar can be constructed, which defines a language describing the syntax of an array of integers. When a SOAP message containing an array of three integers is given as input, the parse tree illustrated in Figure 5.3 is derived.

5.1.5 Handling SOAP Headers

When performing unmarshalling of a request/response, SOAP headers can be sent as part of the SOAP message. The skeleton and stub code layers can not determine the semantics of those headers, since they are application specific. Therefore, when the service

Input SOAP Message	Derived parse tree

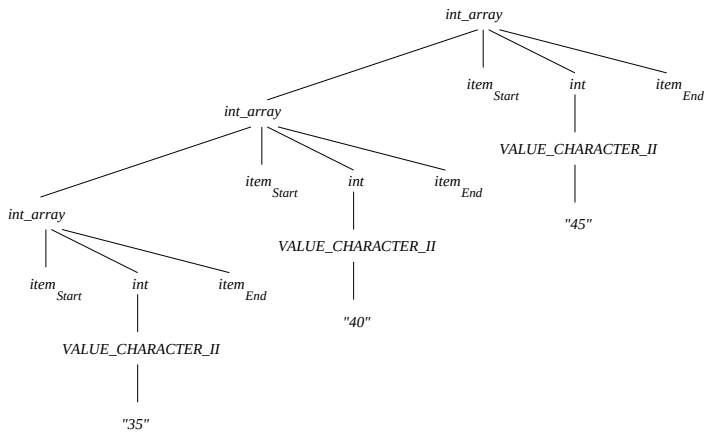
`<item>35</item>`
`<item>40</item>`
`<item>45</item>`
	

Figure 5.3: The parse tree derived from the grammar described in Example 5.5, when a SOAP message containing an array of three integers is given as input.

specific code is performing unmarshalling, it must be able to parse the SOAP headers, and determine if the `mustUnderstand` attribute is true or false. If it is false, the header is ignored, and if it is true an error is generated. Example 5.6 illustrates a simplified context-free grammar which defines a language describing the syntax of a SOAP header. We do not further discuss how SOAP headers are handled.

$$
\begin{aligned}
\mathit{Header} &\rightarrow \mathit{header\_list} \\
\mathit{header\_list} &\rightarrow \mathit{header\_list}\ \ \mathit{header\_item} \\
\mathit{header\_list} &\rightarrow \mathit{header\_item} \\
\mathit{header\_item} &\rightarrow \mathbf{UNKNOWN\_ELEMENT\_II_{Start}} \\
&\quad\ \mathbf{mustUnderstand_{Start}} \\
&\quad\ \mathbf{VALUE\_CHARACTER\_II} \\
&\quad\ \mathbf{mustUnderstand_{End}} \\
&\quad\ \mathbf{VALUE\_CHARACTER\_II} \\
&\quad\ \mathbf{UNKNOWN\_ELEMENT\_II_{End}}
\end{aligned}
$$

Example 5.6: A simplified context-free grammar that defines a language describing the syntax of a SOAP header.

5.2 Semantic Actions for Unmarshalling

So far we have discussed how we can generate context-free grammars, which defines languages that describes the syntax of SOAP messages. However, when performing unmarshalling, the service specific code needs to do more than just accepting a SOAP message as a valid input string. By accepting a SOAP message, we only know that the received SOAP message is syntactically correct, but no actions are carried out. Therefore we need to assign semantic actions to be carried out when an input string is syntactically correct.

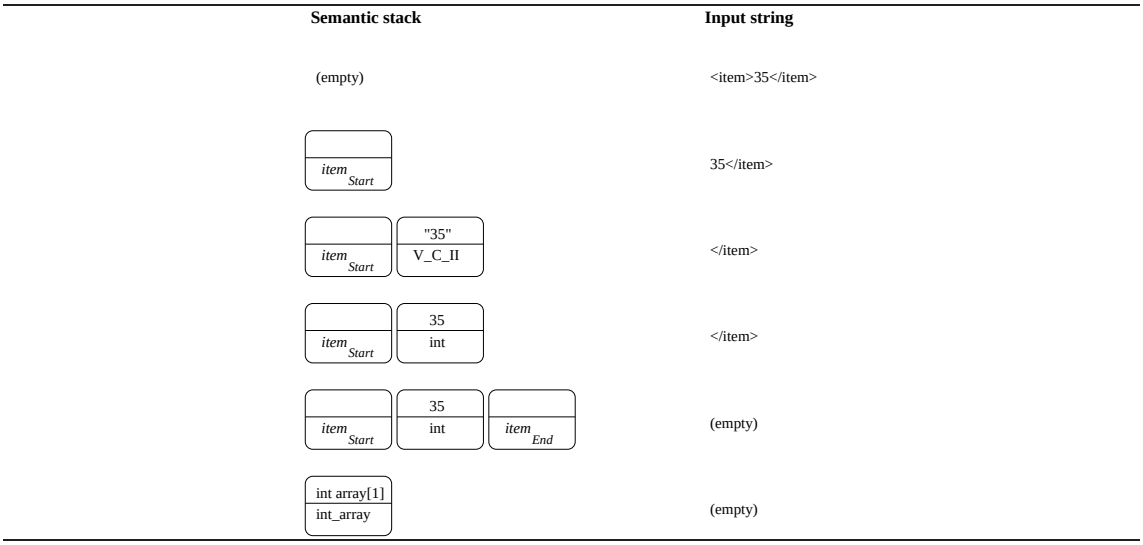
Observing the parse tree in Figure 5.3, one approach is to assign semantic values to each node in the tree (i.e. to each production in the context-free grammar). In Table 5.1 we illustrate the semantic values that each node in the tree would have using this approach.

Node/Production name	Semantic Value
VALUE_CHARACTER_II	"35"
VALUE_CHARACTER_II	"40"
VALUE_CHARACTER_II	"45"
int	`35`
int	`40`
int	`45`
int_array	`int array[1] = {35}`
int_array	`int array[2] = {35, 40}`
int_array	`int array[3] = {35, 40, 45}`

Table 5.1: Semantic values assigned to each node in the parse tree illustrated in Figure 5.3. The values enclosed in quotes denote a string.

The three $VALUE\_CHARACTER\_II$ nodes all have strings as their semantic values. No further semantics can be assigned to a $VALUE\_CHARACTER\_II$ node, since it depends on the context that it appears in. The three *int* nodes have integers as their semantic values. Finally, each of the *int_array* nodes have an array of integers as their semantic values.

In order to generate a parser from a context-free grammar, a parser generator tool is often used. These tools often provides a semantic stack to hold the semantic values associated with each node in the parse tree. However, this approach can be a cause of inefficiency.



Example 5.7: An example of a typical implementation of a semantic stack, keeping track of semantic values. The terminal symbol $VALUE\_CHARACTER\_II$ is abbreviated to V_C_II.

An example of a typical implementation of a semantic stack is illustrated in Example 5.7. An input string representing an array of integers of the size 1 is given to a

parser, which accepts the language specified by the grammar from Example 5.5. Initially, the semantic stack is empty. After the first symbol (the terminal symbol $item_{Start}$) has been read, it is placed on the semantic stack. Next a $VALUE\_CHARACTER\_II$ symbol is read, and it is placed on the stack with the semantic value "35" (denoting a string). Then the parser uses the rule $int \rightarrow VALUE\_CHARACTER\_II$ to reduce this symbol to a non-terminal *int*. This is now placed on the stack, having the semantic value `35` (an integer value). Next, the final part of the input string is read, and the terminal symbol $item_{End}$ is placed on the stack. Finally the parser applies the rule $int\_array \rightarrow item_{Start} \quad int \quad item_{End}$, and now only one semantic value is placed on the stack: The *int_array* non-terminal symbol, having the semantic value `int array[1] = {35}`.

This development of the content of the semantic stack is undesirable since it includes unnecessary copying of data. Initially, the semantic value of the terminal symbol $VALUE\_CHARACTER\_II$ is placed on the stack. Then, a reduction is made, removing this semantic value from the stack and adding a new semantic value of `35`. Next, the terminal symbol $item_{End}$ is placed on the stack and another reduction is made, removing all the current elements on the stack and adding a new—namely the integer array. This process includes the unnecessary copying of the same data. First the string "35" is converted to an integer. Then this integer is copied to another position, where it is an element in an array. This copying of data is unnecessary, and it would be ideal if the data could have been copied to it's final destination in the first place (to its place in an array in this case).

5.2.1 Introducing a Specialized Semantic Stack

In order to overcome the problem of copying unnecessary data, we introduce a specialized semantic stack. Instead of assigning a semantic value to each node in the tree, we assign a semantic action to be carried out whenever a reduction of some rule in the grammar is carried out. This semantic action has the responsibility of placing data on the specialized semantic stack. This requires that the general grammar is changed a little as illustrated in Example 5.8.

In this grammar the use of the the terminal symbols $parmN\_1_{Start} \ldots parmN\_N_{Start}$ has changed. For each of these, a new production has been added to the grammar (e.g. $stack_{parmN\_N_{Start}} \rightarrow parmN\_N_{Start}$). This new non-terminal is used when specifying the parameter list of a given $functionN$ production, instead of the $parmN\_N_{Start}$ terminal.

Whenever an input string is parsed, and one of the $stack_{parmN\_N_{Start}}$ productions are reduced, it indicates that all the following reductions that will occur, represents parameter `N` to function `N`, until the $parmN\_N_{End}$ terminal is encountered.

For each of these parameters we assign a stack, which is used to hold the semantic value of this parameter. We illustrate this concept in Figure 5.4. A SOAP message representing a invocation of a function taking `N` parameters, is parsed and a parse tree is derived. For each new parameter specified in the input string, a terminal symbol on the form $parmN\_N_{Start}$ is encountered in the input. This symbol triggers a reduction of the production $stack_{parmN\_N_{Start}} \rightarrow parmN\_N_{Start}$.

We use this production to carry out a semantic action—namely to initialize a specialized semantic stack just big enough to hold the semantic value of parameter `N`. Whenever any further reductions are carried out (e.g. a reduction of a *short* or *long* production, as illustrated in the example), the semantic value of these are saved on this particular stack.

$$
\begin{aligned}
Start &\rightarrow Envelope \\
Envelope &\rightarrow \mathbf{Envelope_{Start}}\ Body\ \mathbf{Envelope_{End}} \\
Body &\rightarrow \mathbf{Body_{Start}}\ request\ \mathbf{Body_{End}} \\
request &\rightarrow function1 \\
request &\rightarrow \ldots \\
request &\rightarrow functionN \\
function1 &\rightarrow \mathbf{function1_{Start}}\ stack_{parm1\_1_{Start}}\ type\ \mathbf{parm1\_1_{End}}\ \ldots \\
&\quad\ stack_{parm1\_N_{Start}}\ type\ \mathbf{parm1\_N_{End}}\ \mathbf{function1_{End}} \\
stack_{parm1\_1_{Start}} &\rightarrow \mathbf{parm1\_1_{Start}} \\
stack_{parmN\_1_{Start}} &\rightarrow \mathbf{parmN\_1_{Start}} \\
functionN &\rightarrow \mathbf{functionN_{Start}}\ stack_{parmN\_1_{Start}}\ type\ \mathbf{parmN\_1_{End}}\ \ldots \\
&\quad\ stack_{parmN\_N_{Start}}\ type\ \mathbf{parmN\_N_{End}}\ \mathbf{functionN_{End}} \\
stack_{parmN\_1_{Start}} &\rightarrow \mathbf{parmN\_1_{Start}} \\
stack_{parmN\_N_{Start}} &\rightarrow \mathbf{parmN\_N_{Start}}
\end{aligned}
$$

Example 5.8: A revised general structure of any context-free grammar generated for unmarshalling requests, that allows for a specialized semantic stack.

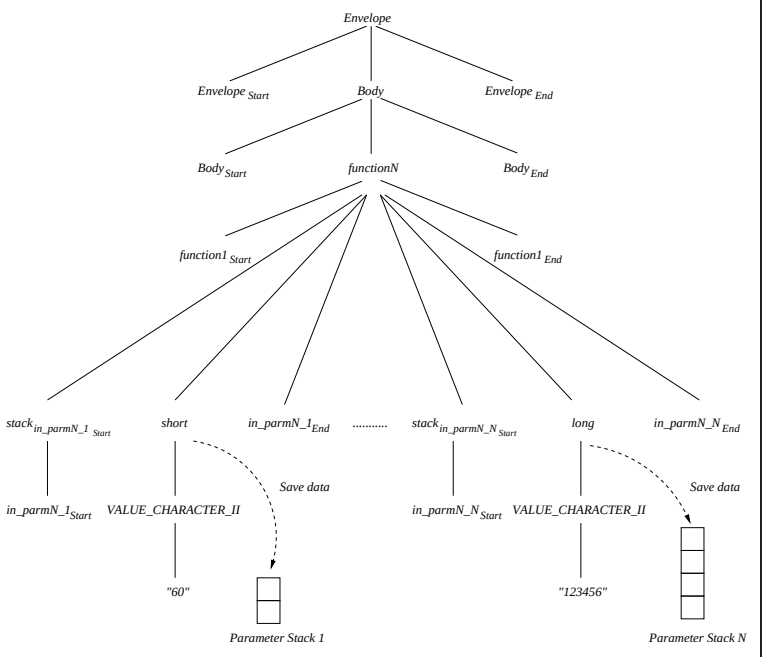
Input SOAP Message	Derived parse tree
<Envelope> <Body> <functionN> <in_parmN_1> 60 </in_parmN_1> .... <in_parmN_N> 123456 </in_parmN_N> </functionN> </Body> </Envelope>	

Figure 5.4: A parse tree derived from the grammar specified in Example 5.8, when a SOAP message representing a request for invoking a function with `N` parameters is the input string. For each parameter, semantic actions are carried out, which copies the value of the parameter to a specialized stack. The input SOAP message is illustrated without namespaces.

For each terminal symbol on the form $parmN\_N_{Start}$ encountered in the input string, we will initialize a stack to hold the semantic value of the parameter it represents.

5.2.2 Avoiding Copying of Semantic Values

Having introduced the idea of assigning a specialized semantic stack for each parameter in the requested function, we now illustrate how this can greatly affect the amount of data needed to be copied.

Input SOAP Message

```
<struct_a_struct>
  <member1>
    42
  </member1>
  <member2>
    60.2
  </member2>
  <member3>
    <member1>
      15
    </member1>
  </member3>
</struct_a_struct>
```

Derived parse tree

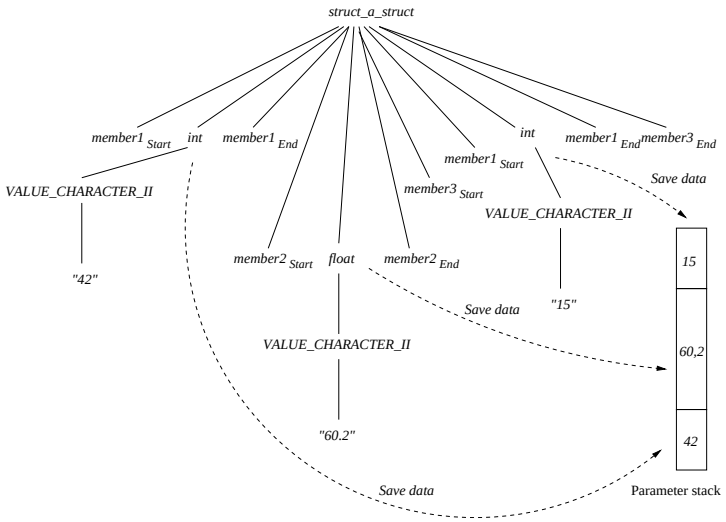


Figure 5.5: An example of a parse tree derived from the grammar specified in Example 5.4, when a SOAP message representing the C struct type illustrated in Figure 2.1 is given as input string. Whenever a reduction of a rule in the grammar is carried out, the semantic value is placed on the specialized stack.

Figure 5.5 illustrates the parse tree derived when a SOAP message representing the C struct data type illustrated in Figure 2.1 is parsed. In the figure we assume that a stack has been initialized to hold the value of the C struct type. Whenever a reduction is made, using one of the productions in the grammar which represents a native data type, the semantic value of this is placed directly on the stack. Initially, when the production $int \rightarrow VALUE\_CHARACTER\_II$ is carried out, its semantic value of `42` is placed on top of the stack. Next, when the production $float \rightarrow VALUE\_CHARACTER\_II$ is carried out, its semantic value `60.2`. Finally, the production $int \rightarrow VALUE\_CHARACTER\_II$ is carried out, and the semantic value `15` is placed on top of the stack.

Traditionally, when using a parser generator tool, all these values would have been placed on the semantic stack and then the *struct_a_struct* production would have been carried out, which would copy all these values into a new semantic value. With our approach, we build the final semantic value (the struct) immediately, without any unnecessary copying of data.

5.2.3 Memory Management

In order to handle the memory used to hold the values of the semantic stacks efficiently, we introduce a simple memory management algorithm.

A semantic stack is allocated for each parameter in the requested function, and therefore memory will need to be allocated dynamically for each of these. However problems are often associated with dynamic memory allocation. Such problems include fragmentation

of memory, the overhead of allocation and deallocation of memory, and others [27].

Therefore, we introduce a simple algorithm for handling allocation of memory. This algorithm takes advantage of the fact that we know the pattern of how memory will be allocated. Given the fact that the skeleton and stub layers have the responsibility of unmarshalling requests/responses for a limited set of functions, the maximum amount of memory needed for handling these can be calculated. For a given function, the amount of memory needed to hold the semantic value of the parameter list, is the accumulated size of the semantic value for each of the parameters. This is with the exception of parameters with dynamic sized data types, which we address later. The maximum amount of memory that will ever be needed to unmarshal a request/response will be the memory needed by the function that has a parameter list with the largest accumulated semantic value. In Figure 5.6, we illustrate a simple algorithm `allocateMaxMemory`, which allocates the maximum amount of space needed for holding the semantic values.

```
allocateMaxMemory(F):

1   max-size ← 0
    for each function f ∈ F:
      total-semantic-size ← 0
      for each parameter p ∈ f:
5       total-semantic-size ← total-semantic-size + semantic-size-of(p)
      if total-semantic-size > max-size
        max-size ← total-semantic-size
    allocate max-size
```

Figure 5.6: The `allocateMaxMemory(F)` algorithm. `F` denotes the set of functions that the skeleton/stub layer provides unmarshalling functionality for.

Initially, when a service requester or service provider is initialized, the `allocateMaxMemory` algorithm is executed. This ensures that just enough space is allocated to handle the function with the largest parameter list. Furthermore, during the initialization of skeleton code, this amount of memory needs to be allocated per thread in the thread pool of the server, to achieve thread safety.

Whenever a new stack is needed to hold the semantic value of a parameter, the memory pool allocated during initialization is used.

This is illustrated in Figure 5.7. Each semantic stack is assigned memory from the memory pool, and a reference is kept to the position in the memory pool where memory is available. Whenever a new semantic stack is needed, the memory needed to hold the stack is assigned from the memory pool, and the reference to the position where memory is available is shifted.

Whenever the unmarshalling of a request/response has been performed, deallocating the memory used is a simple matter. Since the memory is not needed any more and will not be accessed any further until the next response/request is handled, the reference to available memory is changed to the start of the memory pool. When the next request/response needs to be processed, the memory is merely reused and the values are overwritten.

This relieves the operating system the task of deallocating memory, since this is never done until the application has finished executing. Thereby, problems of fragmentation of memory and the overhead of deallocation of memory is avoided.

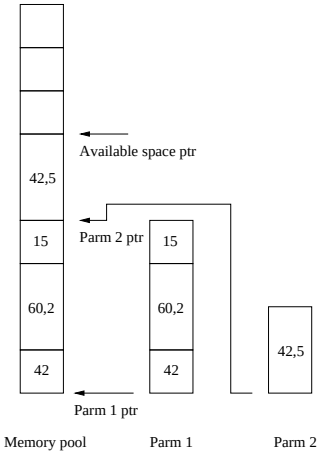


Figure 5.7: The assignment of memory for holding semantic values of parameters from the memory pool. Each semantic stack has memory allocated from the memory pool, and a reference is kept to the place in the memory pool where memory is available.

5.2.4 Dynamic Sized Semantic Values

There are exceptions to the memory management described above. Parameters can have data types which have undefined sizes, i.e. strings and unbounded arrays. Since the sizes of such parameters are unknown, memory can not be allocated to hold these during the initialization of the stub/skeleton code, and therefore needs to be allocated during unmarshalling of a request/response.

In this case we rely on the memory allocation mechanisms of the operating system. Whenever a parameter with an unknown size is being unmarshalled, a semantic stack with a certain amount of memory is allocated. If a situation occurs where the stack reaches its capacity, the size of the stack is grown with a certain amount of memory using the memory management routines of the operating system, and the process of building the semantic value if continued. If another overflow occurs, the stack is grown again, and this process continues until the semantic value is constructed.

5.3 Marshalling

In our prior work presented in DAT5 [2], marshalling of messages was performed by the middleware layer explicitly sending individual strings of XML. The implemented Web service was only able to marshal a single type of message, and only into XML. In this section we briefly describe a method to create the part of the middleware responsible for marshalling messages.

In order to be able to perform marshalling into different formats than XML, the marshalling layer needs to use the XML information set interface specified in Section 4.4.4, consisting of the two functions `xis_write_ii` and `xis_write_character_value_ii`.

A specific data type can always be marshalled using the same marshalling code. For example, a struct is always marshalled by generating an element information item for each of its members. Marshalling the members is a matter of inserting the marshalling code for their types. Marshalling of simple data types is done by generating a character value information item based on the value of the variable in question.

Figure 5.8 shows an example of marshalling code for the `struct a_struct` data type. The marshalling code consists of calls to the XML information set layer. The IDs of all

C Declaration	Marshalling Code

`struct a_struct{`
`  int member1;`
`  float member2;`
 	
`  struct {`
`  int member1;`
`  } member3;`
`};`
	`xis_write_ii(MEMBER1_START, c, s);`
	`xis_write_character_value_ii("42", c, s);`
	`xis_write_ii(MEMBER1_END, c, s);`
	`xis_write_ii(MEMBER2_START, c, s);`
	`xis_write_character_value_ii("60.2", c, s);`
	`xis_write_ii(MEMBER2_END, c, s);`
	`xis_write_ii(MEMBER3_START, c, s);`
	`xis_write_ii(MEMBER1_START, c, s);`
	`xis_write_character_value_ii("15", c, s);`
	`xis_write_ii(MEMBER1_END, c, s);`
	`xis_write_ii(MEMBER3_END, c, s);`

Figure 5.8: Marshalling code for the `struct a_struct` data type. `c` denotes the network connection that is used for sending the marshalled data, and `s` denotes a status struct containing the outcome of sending the data.

non-character value information items are available as named constants, and these names are used in the calls to `xis_write_ii`.

5.4 Summary

In this chapter we have discussed the design of the service specific code layer. We have provided a design for performing unmarshalling of requests/responses by the use of context-free grammars, and have discussed how such context-free grammars can be constructed to support unmarshalling of the different data types that we defined in our mapping between C and SOAP.

Furthermore, we have discussed how semantic actions can be assigned to the context-free grammars in an efficient manner, which avoids unnecessary copying of data by using a specialized semantic stack. An efficient memory management strategy of the memory needed for the semantic stack has been discussed. This strategy is based on calculating the amount of memory needed to hold the unmarshalled semantic values at the time that WSTOOL is executed.

Finally, we have briefly discusses how marshalling can be performed.

6

Architecture of WSTOOL

In this chapter we discuss the overall architecture of WSTOOL. This tool has the responsibility of generating the stub and skeleton layers that we have discussed in the previous chapters. Our motivation in this chapter is to provide an overview of the tasks, that the tool must perform. However, we do not provide a detailed discussion of how these tasks are carried out, since this is not the focus of this thesis. Our motivation for designing and implementing WSTOOL is to create a concrete platform for testing the principles we have discussed in the previous chapters, and to gain a better understanding of the challenges involved.

6.1 Service Provider Generator

In this section, we discuss the design of the service provider generator. This is the module in WSTOOL which is responsible for generating the skeleton layer. The servicer provider generator has two responsibilities:

- Parsing a C header file in the format discussed in Chapter 2 and build an abstract representation of the functions and data types.
- Generating skeleton layers that support unmarshalling/marshalling of the functions exposed as a Web service.
- Generating a WSDL specification of the Web service specified in the C header file.

In the following, we discuss the design of these two functionalities.

6.1.1 C Header File Parser

The function of the C header file parser is to parse a C header file in the format discussed in Chapter 2, and to build an abstract representation of it.

In Appendix A we present a modified context-free grammar for the ANSI C language. The grammar is based on the version of the ANSI C grammar presented by Kernighan and Ritchie in 1988 [10]. We have modified this grammar in order to avoid any ambiguities,

and to remove some of the syntax of the C language that is not needed in our C header files. The parts that have been removed is mainly the part supporting statements, expressions, and definitions of functions, since these are not needed in the C header file specifying the Web service.

The task carried out by the C header file parser is the building of a syntax tree representing the parsed C header file. This tree is then used for further processing by the skeleton layer generator.

6.1.2 Skeleton Layer Generator

In Figure 6.1 we illustrate an algorithm on an abstract level, which generates the skeleton layer. With our modified C grammar, a C header file basically consists of a set of *declarations*. A declaration is either a declaration of a new type such as a struct or enumeration, or the declaration of a function prototype.

```
generateSkeletonLayer(syntax-tree):

1  for each declaration d ∈ syntax-tree:
     generateInformationItemList(d)
     if isTypeDeclaration(d):
       sanityCheckType(d)
5      generateWSDLTypeDefinition(d)
       generateTypeMarshallingCode(d)
       generateTypeUnmarshallingGrammar(d)
     else if isFunctionPrototype(d):
       sanityCheckFunction(d)
10     generateWSDLOperation(d)
       for each parameter p ∈ d:
         generateInformationItemList(d)
         sanityCheckParameter(p)
         if parameterDirection(p) == (IN or IN_OUT)
15         generateWSDLOperationInputParameter(p)
           generateParameterUnmarshallingGrammar(p)
         if parameterDirection(p) == (IN_OUT or OUT)
           generateWSDLOperationOutputParameter(p)
           generateParameterMarshallingCode(p)
```

Figure 6.1: The `generateSkeletonLayer` algorithm, which on an abstract level describes the steps carried out when generating the skeleton layer.

The algorithm visits all the declaration nodes in the constructed syntax-tree. For each declaration, the appropriate XML information set information items are generated, which represents the tokens that the skeleton layer will receive/send when unmarshalling or marshalling the given declaration (line 2). These information items are used to initialize the XML information set layer. If a declaration node, represents a type definition, the following tasks are carried out:

- Sanity checking of the type.
- Generation of a XML Schema type definition for the type, to be used in the WSDL specification of the Web service.
- Generation of marshalling and unmarshalling code for the type.

Initially, sanity checking of the type is carried out (line 4). This involves checking if the type definition is valid accordingly to the conventions we discussed in Chapter 2. If the type declaration is a struct declaration it is checked that all members of the struct are data types that have already been declared, and it is checked that the data types are not pointers. Furthermore, if a member itself is a declaration of a new derived type, it is also treated as a declaration.

If a node passes the sanity checking, XML Schema type definitions are declared for the type, using the mapping we developed in Chapter 2, and these definitions are added to the WSDL document being constructed (line 5). If the type does not pass the sanity checking, the algorithm is terminated and an error message is issued.

Furthermore, a context-free grammar is generated which supports unmarshalling this given type, and marshalling code is also generated to support marshalling the type (lines 6-7).

If a declaration node represents a function prototype, a similar set of tasks are carried out:

- Sanity checking of the function.
- Sanity checking of the parameters.
- Generation of WSDL to represent the function.
- Generation of marshalling and unmarshalling code for the parameters with the appropriate directions.
- Generation of XML information set information items for each parameter.

As with type declarations, sanity checking is carried out on the function prototype declaration (line 9). It is checked that the functions has no return type, and that the parameters specified in the parameter list of the function are of valid types (line 13). Furthermore, it is checked that the parameters have parameter directions specified, and that they are specified as pointers, as discussed in Chapter 2. Finally, if a parameter is an array type, that an extra parameter describing the size of the array is present.

If the sanity check is passed, WSDL is generated to specify this function, and its parameter list (line 10, 15, 18). For all the parameters with the direction In or In-Out a context-free grammar is generated to provide unmarshalling for these parameters (lines 14-16). For all the parameters with the direction In-Out or Out, marshalling code is generated. Finally, XML information set information items are generated for each parameter (lines 17-19).

The final output of the algorithm is put into three distinct files. The WSDL specification is put in a WSDL file. The marshalling code is placed in a C file, and the unmarshalling grammar is placed in a grammar specification file.

6.2 Service Requester Generator

In this section, we discuss the design of the service requester generator. This is the module in WSTOOL which is responsible for generating the stub layer. The servicer requester generator has three responsibilities:

- Parsing a WSDL document into an abstract representation of the document, which again is used to create abstract representations of the corresponding C functions and data types.

- Generating stub layers that support unmarshalling/marshalling of the functions provided by the Web service specified by the WSDL document.

In the following, we discuss the design of these two functionalities.

6.2.1 WSDL Parser

The responsibility of the WSDL parser is to parse the WSDL input given to WSTOOL to build an abstract representation of the Web service specified in that file. The WSDL file specifies the operations which are exposed by the Web service, as well as data type definitions used by these operations. The abstract representation is used to map the operations to C function prototypes along with the implementations of these functions. The representation is also used to map the defined types to C data type definitions.

6.2.2 Stub Layer Generator

Figure 6.2 shows on an abstract level the algorithm for generating the stub layer code given the abstract representation of the WSDL document as input. Initially, the algorithm iterates through the all the type definitions in the abstract representation of the WSDL document. A sanity check is performed on the types to ensure that it is supported by WSTOOL (line 2). When this check has been passed, a C type definition is generated from the representation of the type definition (line 3). Finally XML information set information items are generated for the given type (line 4).

```
generateStubLayer(syntax-tree):

1  for each typeDefinition t ∈ syntax-tree:
     sanityCheckType(t)
     generateCTypeDefinition(t)
     generateInformationItemList(t)
5  for each operation o ∈ syntax-tree:
     sanityCheckOperation(o);
     generateCFunctionPrototype(o);
     generateInformationItemList(o)
     for each parameter p ∈ o:
10     sanityCheckParameter(p)
       generateInformationItemList(o)
       if parameterDirection(p) == (IN or IN_OUT)
         generateParameterMarshallingCode(p)
       if parameterDirection(p) == (IN_OUT or OUT)
15       generateParameterUnmarshallingGrammar(p)
```

Figure 6.2: The `generateStubLayer` algorithm, which on an abstract level describes the steps carried out when generating the stub layer.

Next, the algorithm iterates through the set of operations. A sanity check is performed on each operation, in which it is checked whether the operation type is supported by WSTOOL (line 6). Only operations using the In-Out message exchange pattern and the RPC style are supported. If the operation passes this check, a C prototype declaration is generated(line 7). Finally XML information set information items are generated for the operation (line 8).

Finally, the algorithm iterates through the list of parameters defined for an operation. Each parameter must pass a sanity check (lines 10) to ensure that it is valid. For instance,

its type must either be a built-in type or have been defined previously. XML information set information items are generated for the parameter (line 11). Then, marshalling and unmarshalling code is generated for the parameter in the same manner as for the service provider layer (lines 12-15). The only difference is, that marshalling code must be generated for In and In-Out parameters, while an unmarshalling grammar must be generated for Out and In-Out parameters.

The final output of the algorithm is put into three distinct files: The type definitions and function prototypes are placed in a C header file. The marshalling code is placed in a C file, and the unmarshalling grammar is placed in a grammar specification file.

6.3 Summary

In this chapter we have briefly discussed the design of WSTOOL. The tool consists of two main modules, namely the service provider generator and the service requester generator. The responsibilities of the service provider generator is to generate skeleton code and a WSDL specification by parsing a C header file specifying a Web service. The responsibility of the service requester generator is to generate stub code from a WSDL specification. We have provided algorithms on an abstract level, describing the tasks carried out by the service provider generator and the service requester generator.

7 Implementation

In this chapter we discuss the implementation of WSTOOL and the middleware layers generated by this tool. Our motivation is to give an overview of implementation specific details, and to describe the existing tools and implementations that we have based our work on.

Two of the features of WSTOOL described in the earlier chapters are at the time of writing not included in the implementation. The XML information set layer has been implemented for unmarshalling only. Additionally, the `ws_request_outcome` and `ws_response_outcome` are not part of the middleware interface.

The chapter starts by describing the implementation of WSTOOL. We present the tools that were used in the implementation of the header file parser and the WSDL parser.

Next, implementation details of the common code are discussed. We describe how the thread pool layer has been implemented by the use of POSIX threads, and describe how the XML information set layer was implemented by using the Expat parser library.

Next, a description of the steps involved in creating service requesters and providers using WSTOOL is given.

We end the chapter by describing how the service specific code has been implemented using the Bison parser generator.

7.1 The wstool Application

The WSTOOL application has been implemented following the design presented in Chapter 6. Since this application is to be executed on a host used for development, and not on the target platform, the implementation focuses on being maintainable and extensible, rather than performing well with respect to execution time and memory requirements. For that reason, the application has been implemented in the C++ programming language. C++ is an object oriented programming language, which aids the development of maintainable and extensible code through the use of abstraction. Additionally, the application depends on the Xerces XML parsing library for C++ [26], and the Bison parser generator tool [6]. These has been used as a basis for simplifying the implementation.

The source code of WSTOOL is available on the CD-ROM attached at the back of the thesis. To build and install WSTOOL, follow the instructions found in Appendix C.

7.1.1 C Header File Parser

The C Header file parser has been implemented by using the Bison parser generator tool [6]. Bison is a general-purpose parser generator that converts a grammar description for an LALR context-free grammar into a C/C++ program to parse that grammar [1] . We have used the Bison compatible specification of ANSI C provided by Jeff Lee [8] as the basis for the implementation of the C header file parser, and modified it to our needs. Our modified version of the ANSI C grammar, which does not include statements, expressions and definitions of functions, can be found in Appendix A.

We have implemented a complete set of classes that represent the syntax tree obtained when parsing a C header file. An example of one of these classes is illustrated in Example 7.1. The algorithm discussed in Chapter 6 is executed on this syntax tree.

```
1   namespace AST {

      /** This class represents a postfix part of a array (such as [], [5], etc.) */
      class ArrayPostfix : public Postfix
5     {
      public:
        /* Constant, representing that the size of the array prefix is */
        /* undefined (i.e the array postfix is []) */
        static const unsigned char UNDEFINED_SIZE = 0;
10
        /* Create an ArraPostfix object with a given size */
        /* (e.g in the case of [5], size is 5) */
        ArrayPostfix(unsigned long int size);

15      /* Generates a string representing the necessary C code */
        /* to declare the type defined by this array postfix */
        string generateTypeDeclarationCode();

        /* Gives the size specified in the array postfix (e.g. in */
20      /* the case of [5], arraySize() gives you 5) */
        unsigned long int arraySize();

      private:
        /** Internal representation of the size of the array postfix */
25      unsigned long int size;

      };
    }
```

Example 7.1: An example of a C++ class representing part of the syntax tree of a C header file.

7.1.2 WSDL Parser

The WSDL parser has been implemented using the Xerces XML parsing library for C++. The Xerces parser provides both a SAX and a DOM interface. The DOM interface, which builds a tree representation of XML documents, is used in the implementation. The DOM interface is less efficient than the SAX interface, but the generated tree structure eases algorithmic processing of WSDL documents.

The parsing of a WSDL document consists of two tasks. The first task is to parse all type definitions in the document. The second task is to parse the part of the document which specifies the defined operations.

The WSDL parser retrieves information about the types which are defined in a WSDL document by accessing the relevant parts of the DOM tree, which the Xerces parser has produced. The type definitions are represented using the XML Schema standard [24]. The DOM tree produced by the Xerces parser is traversed seeking for all XML Schema type definitions. For each type found, a syntax tree of the corresponding C data type definition is created, based on the mapping defined in Section 2.1. This syntax tree is later used to create both marshalling and unmarshalling code. Unsupported type definitions are ignored. The syntax tree representations of the supported types are stored in a symbol table for later reference.

The WSDL parser retrieves information about the operations defined in a document by accessing the relevant parts of the DOM tree. Each operation must consist of an input and output message pair. Each of these messages has a type. This type must have been found while parsing type definitions. Otherwise, WSTOOL exits with an error message. When the operation has passed all semantic checks, a list of abstract representations of the operations which have been found is generated.

7.2 Common Code

This section describes details about the implementation of the common code layer. Various tools, libraries, and techniques used in the implementation of the common code are described, as well as the concrete interface by which applications can interact with the generated middleware.

7.2.1 Thread Pooling

The design of the thread pooling strategy presented in Section 4.1 was reused from DAT5 [2]. However, the implementation of this strategy has been updated.

The new implementation of the thread pool is based on POSIX threads, commonly referred to as pthreads, which is a standardized API for thread handling, mutual exclusion, and synchronization. The pthreads API aims at providing a portable interface for multi-threaded applications. Using the pthreads API instead of eCos native threads enables the middleware layers generated by WSTOOL to be ported to other operating systems than eCos.

The migration to pthreads is fairly straightforward, except for a single detail. The implementation based on eCos threads use semaphores to signal a processing thread when a new connection has been established. In this implementation the network listener thread unlocks semaphores that are owned by processing threads. Doing this with pthreads may result in undefined behavior according to the pthread specification. The new implementation instead uses condition variables to signal processing threads when a new connection must be handled.

7.2.2 XML Information Set Layer

In the following, we discuss the implementation of the XML information set layer. We have based the implementation of this layer on an existing implementation of an XML parser. We present this parser, and discuss how its interface can be modified to provide the interface of the XML information set layer, as we discussed in Chapter 2.

Expat

The implementation of the XML information set layer was based on the open source XML parser "Expat" [7]. Expat is a stream-oriented XML parsing library written in C. In our previous work presented in DAT5 [2] we also based our prototype implementation on this parser library, with satisfactory results.

The Expat library consists of an API, which allows parsing XML documents in small parts at the time. This is the main reason that we chose Expat, since it supports the concept of streaming, which is in line with the design philosophy discussed previously.

In Figure 7.1 we illustrate the API provided by Expat for parsing part of an XML document, by using the function `XML_Parse()`.

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

Figure 7.1: The interface for Expat's `XML_Parse()` function.

`XML_Parse()` takes four arguments as illustrated in Figure 7.1.

The first argument, `XML_Parser p`, is a data structure representing the instantiation of an Expat XML parser. Expat provides the `XML_ParserCreateNS()` function to create a XML parser, which understands XML Namespaces. The second argument, `const char *s`, contains a pointer to a buffer containing part of an XML document 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.

Expat uses a call back interface to handle the XML data, e.g. if a start tag is encountered a handler function is invoked by Expat. This function then has the responsibility of processing this tag. Three handler functions can be assigned to Expat, and we illustrate their interfaces 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 holds the character data itself. Furthermore, it takes the `length` parameter, which holds the length of the character data.

Token Queue

The interface offered by Expat can not be mapped directly to the interface of the XML information set layer that we designed in Chapter 4. That interface allows the service specific code layer to request the next information item in the XML information set one

information item at the time. This is a *pull* interface, where the upper layers requests informations as needed. The interface offered by Expat is a *push* interface, since Expat parses a buffer of XML and invokes a number of functions which has the responsibility of handling the data.

Therefore, we have implemented a token queue strategy, which allows us to provide a pull interface to the service specific code layer. The concept of this strategy is illustrated in Figure 7.2.

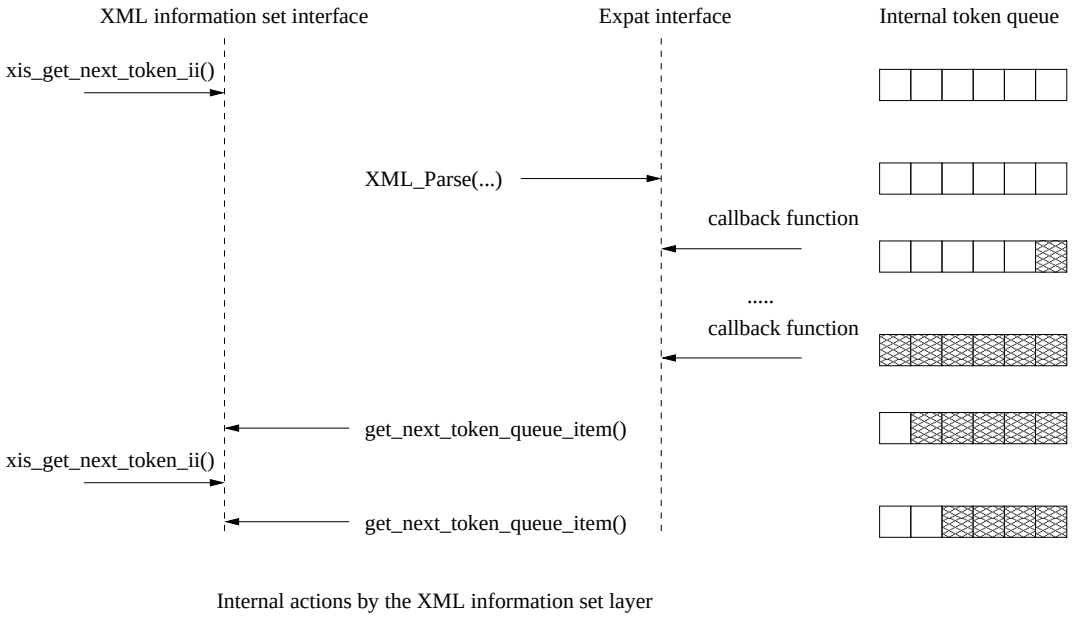


Figure 7.2: The internal operation of the XML information set layer, by the use of Expat and a token queue.

Initially when a request is made to the XML information set layer for an information item token from the XML information set, Expat is executed, and a buffer containing part of an XML document is parsed. Expat then executes a number of the call back functions described above. Each of these functions produces a token, which is then installed in the internal token queue. When Expat has finished processing, the first token from the queue is returned and removed from the queue. When another requests arrives for the next token, this is fetched from the queue without invoking Expat. This process is repeated until the queue is empty, and then Expat is executed again.

7.2.3 Middleware Interface

Since the the common code layer is exactly the same for any Web service generated by WSTOOL, the common code layer is implemented in two libraries. The libraries are installed along with the WSTOOL program itself. Additionally, a set of C header files declaring the interfaces of these libraries are installed. There are separate libraries for the client and server middleware.

Server Interface

The common code for servers is made available through the `soap-server` library. Servers using the WSTOOL generated middleware must link with this library, either statically or

dynamically. Static linking is required in cases where the target platform does not support dynamic loading. This is for instance the case with eCos.

```
1  void ws_imaginary_operation(int *in_out_value);
```

Example 7.2: Contents of `simple_server.h`, which specifies the interface to be exposed as a Web service.

Example 7.2 shows an example of a C header file, which is used to specify an interface to be exposed as a Web service. The Web service exposes a single function, `ws_imaginary_operation`, which takes a single In-Out parameter. This header file is given as input for WSTOOL to generate the server middleware layer required to expose the `ws_imaginary_operation` function as a Web service.

```
1  #include <wstool/soap-serverutil.h>      /* Provides ws_init */
   #include "simple_server.h"               /* Provides service function prototypes */

   void ws_imaginary_operation(int *in_out_value) {
5    *in_out_value = *in_out_value + 1;
   }

   int main(int argc, char **argv) {
     ws_init();                            /* Initialize middleware layer */
10   ...                                   /* Perform other application functions */
     return 0;
   }
```

Example 7.3: Contents of `simple_server.c`, which is an example of an application using the `soap-server` middleware library.

Example 7.3 shows the implementation of the `ws_imaginary_operation`, along with the `main` function of the server application. To initialize the middleware layer, the `main` function must call the `ws_init` function (line 9). This function creates a listener thread, which listens for new network connections, and a set of processing threads for the thread pool. This function returns as soon as the threads have been created. The server application can then perform any other initialization and computation needed.

Client Interface

The common code for clients is made available through the `soap-client` library. As for servers, clients using the WSTOOL generated middleware must link with this library, either statically or dynamically.

Example 7.4 shows an application interfacing with a client middleware layer generated by WSTOOL to invoke the remote procedure exposed by the service specified above. A single header file needs to be included, which is the `soap-client.h` file. This header file was generated by WSTOOL for the specific service, which this client is accessing. The generated header file provides declarations of functions and data types, which are used for invocation of remote procedures. The remote procedure `ws_imaginary_operation` is invoked (line 7) with a pointer to `i` as the only parameter. As seen above, this parameter is an In-Out parameter. After invoking the remote procedure, the value of `i` may have changed, so its value is printed.

```
#include <stdio.h>                      /* Provides printf */
#include "soap-client.h"                /* Provides generated service declarations */

int main(int argc, char **argv) {
  int i = 2;
  printf("Value of i before: %d\n", i);
  ws_imaginary_operation(&i);           /* Invoke remote procedure */
  printf("Value of i after: %d\n", i);
  return 0;
}
```

Example 7.4: Example of application using the `soap-client` middleware library.

7.3 Unmarshalling in Service Specific Code

In order to generate a parser that is responsible for unmarshalling requests/responses we have used the parser generator tool Bison. This tool allows specifying a context-free grammar and specifying the semantic actions that should be carried out when an reduction is made, using a given production in the grammar. From the specification Bison generates a thread safe LALR(1) parser.

In example Example 7.5 we illustrate an example of the production *c_unsigned_long* which represents unmarshalling the data type `unsigned long`. The semantic action to be carried out is specified in braces (lines 3-16). This semantic action is responsible for converting a $VALUE\_CHARACTER\_II$ information item to a C data type of the type `unsigned long`, and place it on the semantic stack (line 9). If the information item $VALUE\_CHARACTER\_II$ is invalid, the semantic action is responsible for generating a SOAP fault (lines 12-15).

```
/* Bison production representing unmarshalling of a unsigned long data type */
c_unsigned_long
     : VALUE_CHARACTER_II  { /* Semantic action */
         char *errorString;
         /* Convert VALUE_CHARACTER_II to an unsigned long,
          * and copy it to semantic stack.
          */

         *VALUE_TARGET(unsigned long) =  (unsigned long)strtoul($1, &errorString, 10);

         /* Check if any errors occurred during conversion */
         if (*errorString != '\0') {
           /* Generate soap fault */
           generate_soap_fault(CODE_SENDER, "Invalid argument representation");
         }
       }
       ;
```

Example 7.5: A production in a context-free grammar using Bison, including the semantic action carried out when a reduction is made using the production.

7.4 Summary

In this chapter we have discussed the implementation of WSTOOL and the middleware generated by the tool. A description of the tools and libraries we have based our imple-

mentation on has furthermore been given.

We have used the Bison parser generator tool for C/C++ as the basis for implementing the service provider generator module of WSTOOL. Furthermore, we have modified an existing grammar specification for Bison of the ANSI C programming language. This modified grammar has been used as the basis for our C header file parser. Our implementation of the service requester generator has been based on the Xerces XML parsing library for C++. This tool provides a DOM interface to access the information in an XML document.

We based our implementation of the thread pooling layer in the common code on the POSIX thread library, which provides a standardized API for handling threads, mutual exclusion, and synchronization. For implementing the XML information set layer, we have used the low-level XML library Expat, which provides a stream oriented XML parsing API for C. This parsing library offers a call-back interface, which has imposed the need for implementing a token queue to obtain the interface that we have specified of the XML information set layer.

Finally, we have used the Bison parser generator tool for implementing unmarshalling in the service specific code.

8 Performance Evaluation

In this chapter we describe the performance experiments that have been carried out to measure the performance of the middleware layers generated by WSTOOL. We present two alternative tools for developing Web services in ANSI C, namely gSOAP and CSOAP. The performance offered by these tools is measured and compared to that of WSTOOL. Based on the results of these experiments, an evaluation of WSTOOL is given.

8.1 Example Web Service

An example Web service is used to perform the experiments of the performance evaluation. The Web service is a road direction service, which is similar to a service that Aeromark would like to offer to their customers. Figure 8.1 shows the interface of this Web service specified in a C header file.

The service exposes a single function, `ws_get_route_description`, which provides a look-up facility for location-aware mobile agents. Such an agent may for instance be a truck belonging to a delivery company. When a delivery is to be made, the driver of the truck may request a route description from the Web service. This route description is based on the current geographical location of the truck and the postal code of the destination address of the delivery (e.g. in Great Britain postal codes cover very small areas). The function returns the geographical location of the destination and an array of waypoint descriptions, which specify a possible route to that location. The size of this array is also returned. In addition, the number of waypoints which did not fit in the waypoint array is returned. Notice that the size of the waypoint array is defined for the `waypoints_set_t` type.

In the performance experiments this Web service will be used to compare the performance of the middleware layers generated by WSTOOL to other tools implementing Web services in ANSI C. The size of the list of waypoints will be varied to see how this affects the efficiency of the implementations being compared. Note that the size of request messages will be constant, while the size of response messages will vary depending on the size of the list of waypoints.

```
typedef struct {
  float latitude;
  float longitude;
} coordinate_t;

typedef struct {
  struct  {
    float latitude;
    float longitude;
  } position;
  unsigned int road_distance;
  enum {
    DIR_FORWARD,
    DIR_LEFT,
    DIR_RIGHT,
    DIR_TURN_AROUND
  } action;
} waypoint_set_t[50];

void ws_get_route_description(coordinate_t *in_source_pos,
                              unsigned long *in_destination_postal_code,
                              coordinate_t *out_destination_position,
                              waypoint_set_t **out_waypoints,
                              unsigned long *out_waypoints_size,
                              unsigned short *out_remaining_waypoints);
```

Figure 8.1: Web service used for performance experiments specified in a C header file.

8.2 Web Service Tools

The performance evaluation is based on a comparison between three different tools for implementing Web services in ANSI C, one of these tools being WSTOOL. The Web service just described has been implemented using each of these tools. The two existing tools, gSOAP and CSOAP, are now described.

8.2.1 gSOAP

gSOAP [13] is a generating tool for developing Web services in C and C++. It is able to take a C header file as input and from that produce a SOAP server and client. In this respect its architecture is quite similar to WSTOOL, however it does not include the network representation abstraction found in WSTOOL.

8.2.2 CSOAP

CSOAP [3] takes a different approach. Instead of generating middleware layers, CSOAP provides SOAP specific functionality through a library and an associated API. CSOAP does not provide a mapping of types between C and XML Schema. The marshalling and unmarshalling data types such as structs and arrays must be handled by the application developer. All service specific code resides in the client and server applications themselves.

8.3 Test Design

The purpose of the performance experiments is to investigate the efficiency of the stub and skeleton layers generated by WSTOOL. In particular the speed at which requests can

be handled and the amount of memory needed to do so. An efficient implementation of low-level network communication has not been in focus. For that reason, the experiments are designed to not measure the efficiency of low-level communication.

As mentioned above, an example Web service is used as a case for the performance experiments. Using each of the Web service tools described above, the following experiments are carried out:

- **Marshalling and unmarshalling:** The purpose of this experiment is to measure the time taken to marshal and unmarshal the messages which are transferred when invoking the remote procedure defined in the example Web service. The size of the `waypoint_set_t` type is varied in the interval 50 to 1000 elements. The marshalling times are measured in the server, and the unmarshalling times are measured in the client. Changing the size of the list of waypoints should reveal how the marshaling and unmarshalling time depends on the size of the messages. No worse than a linear dependency should be observed.

- **Profiling of wstool generated unmarshalling on client:** This experiment breaks the measurement of the unmarshalling time on the WSTOOL generated client into smaller parts. The purpose is to investigate which parts of the unmarshalling code takes up most of the time. This information may reveal parts which need optimization. Again, the size of the list of waypoints is varied in the interval 50 to 1000 elements.

- **Memory requirements:** In this experiment the stack and heap memory usage is measured on clients. The amount of heap memory allocated by servers is measured as well. Due to the multi-threaded implementations of servers, we where not able to establish any methods for measuring their stack memory usage of the servers. The amount of memory measured is the maximum amount required at any time during execution of the program. The message size is varied to reveal how this changes the memory requirements.

All experiments involving time measurement are performed five times and the times are recorded. The best and the worst time is removed and the mean value of the remaining three times is used for the test results. This is done to reduce the risk that unusually slow or fast times will affect the final results.

Measurements of heap memory usage is performed using the "valgrind" [9] tool, which is a memory debugger for Linux on x86 platforms. Valgrind is able to trace dynamic memory allocations and deallocations, which makes it possible to measure the maximum amount of heap memory used.

Measuring stack memory usage is more complicated. In Linux it is possible to set a maximum limit for stack usage on a process, using the `setrlimit` system call. A process which tries to allocate more stack space than permitted will receive the `SIGSEGV` signal (segmentation fault). By running a program repeatedly, while decreasing its permitted stack allocation, gives a crude estimation of the stack memory it needs to allocate. This method was used for all stack memory measurements.

Times are measured as elapsed time. The time is calculated from a count of elapsed clock cycles and the clock frequency of the test machine. The clock cycle measurement is implemented using assembler instructions specific to the IA32 (Intel 32-bit) architecture, which provide a resolution of approximately $0.05 - 0.1\mu s$ with an approximate overhead of $0.1\mu s$ [16].

8.4 Test Environment

All tests described in Section 8.3 are performed in a controlled environment. All tests are run on a machine dedicated to performing tests. This machine is equipped with an Athlon XP1800+ (1533 MHz) processor and 512 MB RAM. The operating system is Linux 2.6.4. Since the performance of the network communication is not in focus in any of the tests, all communication is performed using the loop-back interface, meaning that all data is transferred locally without using a network interface. This avoids any unrelated traffic to interfere with the tests.

8.5 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 8.3. Tables listing detailed results of the experiments are listed in Appendix B.

8.5.1 Marshalling and Unmarshalling

Figure 8.2 shows the marshalling time on the server-side as a function of the size of the response message which is marshalled. Note that this size is given by the number of elements in the array that stores the list of waypoints. The actual size in bytes of a message for a fixed array size may vary slightly between the three different implementations, since they do not produce the exact same XML document. However, this difference is linear as a function of the array size, and will therefore not worsen the asymptotic time complexity of the marshalling algorithm.

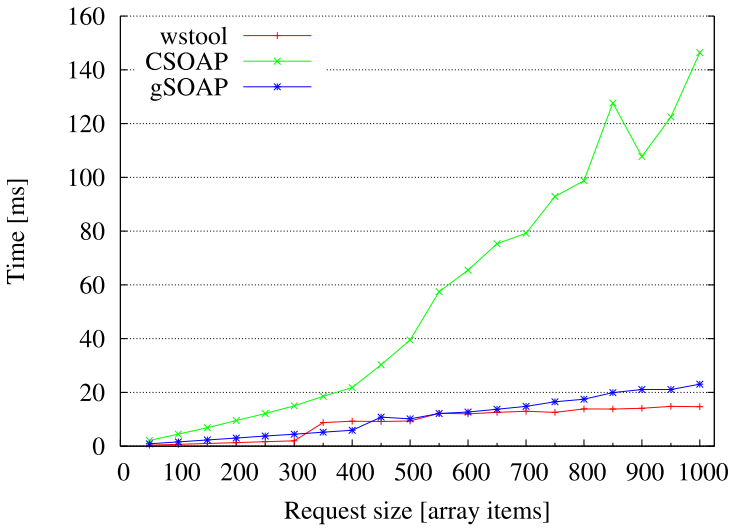


Figure 8.2: Comparison of server marshalling times.

It is seen in the figure that CSOAP performs significantly worse than gSOAP and WSTOOL. gSOAP and WSTOOL have quite similar execution times, however WSTOOL seems to be the fastest for large messages.

The algorithm used in WSTOOL for marshalling arrays consists of m `for`-loops, where m denotes the number of dimensions of the array. The within the innermost `for`-loop the value of the elements of the array is marshalled. In the experiment the size and structure of these elements is constant. All other data to be marshalled have constant size. Therefore,

the asymptotic time complexity should be no worse than $O(n)$, where n denotes the array size. The experiment shows a linear growth as a function of n, however, there is a jump in execution time for $n > 300$. The reason for this jump is unknown, but it may be related to the internal workings of the Linux kernel. Notice in Table B.1 of Appendix B that the calculated standard deviation is high for the measurements in the interval $n \geq 350$ and $n \leq 500$.

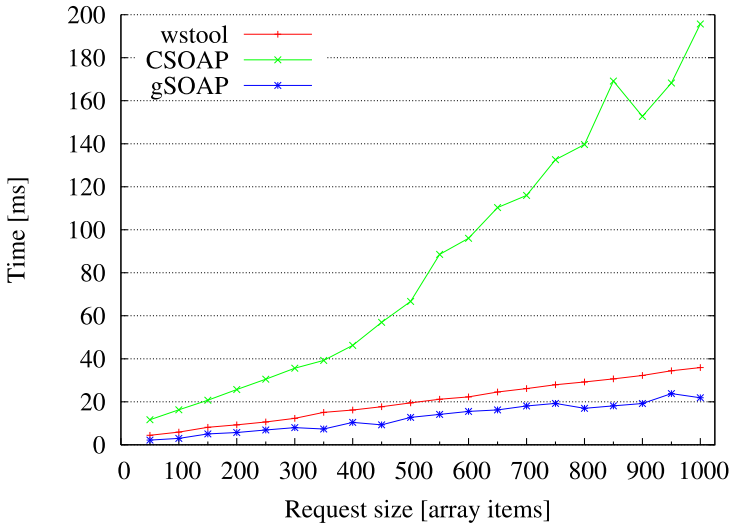


Figure 8.3: Comparison of client unmarshalling times.

Figure 8.3 shows the result of measuring unmarshalling time on the clients. Again, CSOAP shows to be significantly slower than gSOAP and WSTOOL. gSOAP is slightly faster than WSTOOL in this experiment. The results suggest that the time required for unmarshalling in clients in linear as a function of the number of elements in the list of waypoints.

8.5.2 Profiling of wstool Generated Unmarshalling on Client

Figure 8.4 shows the results of the experiment in which the execution time of three different parts of the unmarshalling code generated by WSTOOL was measured. Additionally, the figure shows the remaining amount of time spent in other parts of the unmarshalling code ("Other"). This time represents the time spend unmarshalling the information items received from the XML information set layer. The measurements were performed on client code. Theoretically, unmarshalling on the server side should share these characteristics.

Out of the three measured parts, most time is spent waiting for the Expat XML parser to return new events. It may be possible to reduce the parsing time by implementing a more efficient XML parser. Expat has XML parsing features which are not needed to perform unmarshalling of SOAP messages. A simpler implementation may result in better performance.

A substantial amount of time is spent maintaining the token queue. As described in Section 7.2, the token queue is used to store a queue of tokens returned by Expat. As described in DAT5 [2], XML Pull parsers enable an application to request tokens directly, as opposed to receiving tokens by call-back. The token queue would not be necessary if for example an XML Pull parser was used instead of Expat. Consequently, the total unmarshalling time may be reduced by making use of a different parsing technique. However we where not able to find an implementation of a XML pull parser for C.

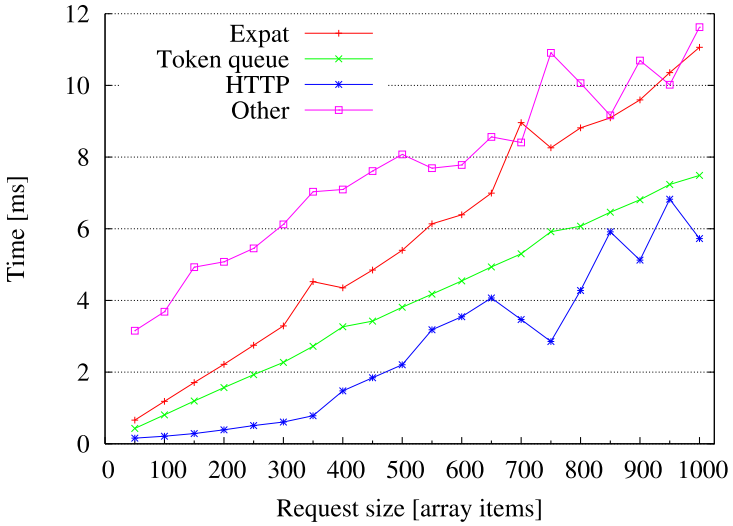


Figure 8.4: Execution time of three different parts of the client unmarhalling code generated by WSTOOL.

The least time consuming part of the unmarshalling code measured is the HTTP communication handling. The responsibility of this part is to read SOAP messages from the network layer and handle the transfer encoding of the HTTP data. In this experiment, the chunked transfer encoding was used, as described in Chapter 4. Notice in Table B.3 of Appendix B that the calculated standard deviation is in general high for these measurements. This is most likely due to the way the network socket calls are handled by the Linux kernel.

The remaining amount of time in the unmarshalling code is spent parsing the stream of tokens from the token queue and executing the semantic actions of the parser's productions ("Other").

8.5.3 Memory Requirements

Figure 8.5 shows the maximum amount of dynamically allocated memory (heap) when running the servers. The CSOAP server clearly allocates the most heap memory. By inspecting the source code of the CSOAP library it is seen that the entire XML representation of each message is stored in dynamically allocated memory before it is parsed. Both gSOAP and WSTOOL instead stream the messages to the XML parser using a fixed-size statically allocated buffer.

The WSTOOL generated server allocates 41 KB heap when returning 50 waypoints. This value remains constant for all array sizes. This matches the expected behavior, since the `out_waypoints` parameter is an Out parameter, which does not need to be allocated on the parameter stack. The increased size of the data structure instead results in increased stack usage.

gSOAP allocates 2 KB of heap memory for all array sizes. Since the stack usage of servers have not been measured, we can only assume that space for parameters is allocated on the stack for this type of message.

Figure 8.6 shows the maximum amount of heap memory allocated when running the clients. It can be seen that CSOAP allocates the most memory for messages containing 150 waypoints or more. The memory consumption increases linearly with the array size.

Again it is seen that gSOAP and WSTOOL allocates a fixed amount of heap memory, regardless of the number of waypoints. This is caused by the fact that the parameters are

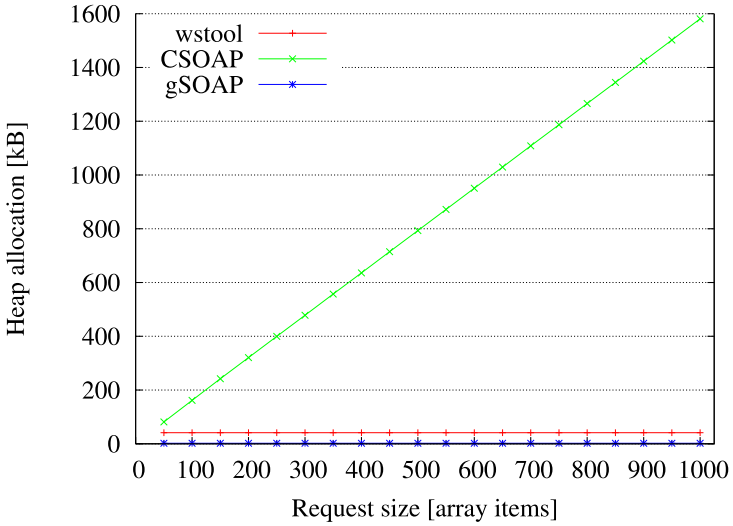


Figure 8.5: Comparison of the amount of heap memory allocated by servers.

allocated in stack memory. gSOAP allocates 45 KB, while WSTOOL allocates 44 KB.

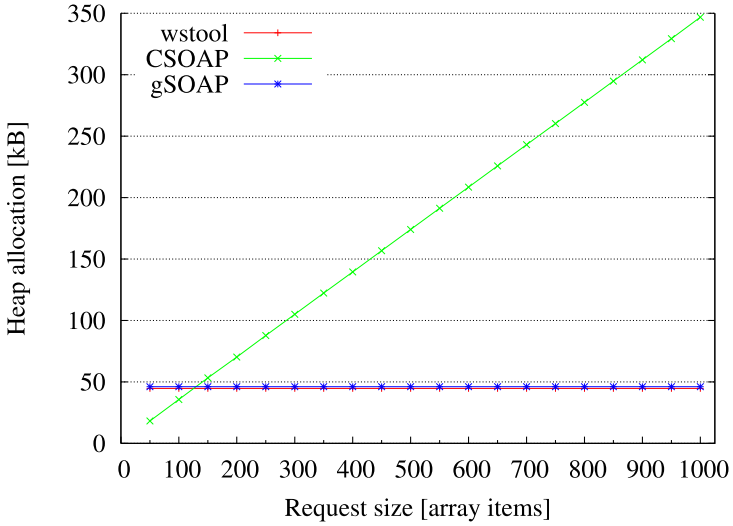


Figure 8.6: Comparison of the amount of heap memory allocated by clients.

Figure 8.7 shows an approximation of the size of the stack that is needed for running the clients. This approximation is based on an experiment, in which the maximum stack size of the client process is limited. The minimum stack size limit at which the process completes without error is noted and used in this result. In the Linux version used, and on the IA32 architecture, the page size is 4 KB, meaning that the stack requirement in measurable to a precision of 4 KB.

The WSTOOL generated code allocates the most stack memory. This space is mainly used for buffers and for the parameter data which the client uses in requests. One of these buffers is the token queue, which it may be possible to remove if another XML parsing technique is used. Additionally, an unknown amount of stack memory is allocated by Expat. Notice that all clients seem to increase their stack allocation by the same amount when the message size increases. It also seems that this increase corresponds to the increased parameter size.

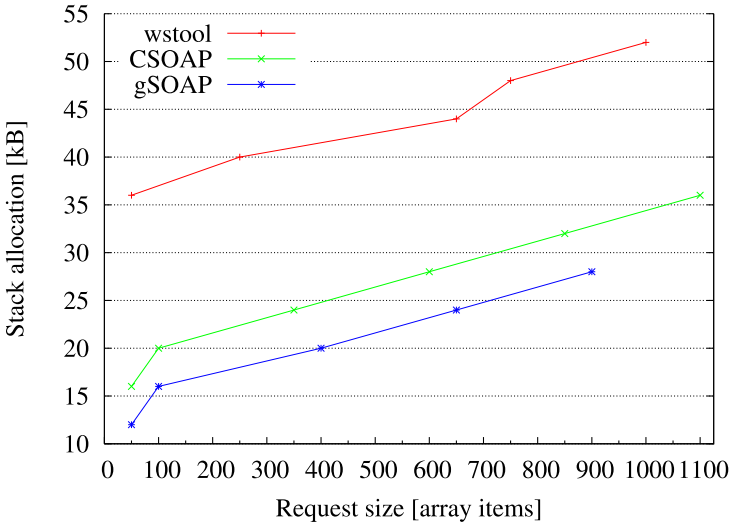


Figure 8.7: Comparison of the amount of stack memory allocated by clients.

8.6 Summary and Discussion

A series of experiments have been carried out to measure performance characteristics of the middleware layers generated by WSTOOL. These experiments have investigated execution speed as well as memory requirements. All experiments have been carried out using each of the three Web service tools presented, CSOAP, gSOAP, and WSTOOL. An example Web service has been defined and used for all experiments.

The experiments have revealed that, when good performance is of importance, the service-specific generating strategy of WSTOOL and gSOAP is preferable ahead of the generic run-time approach that CSOAP takes, both with regard to execution speed and memory consumption. The experiments also reveal that WSTOOL generated clients and servers perform almost as good as those generated by gSOAP. It even performs better than gSOAP for marshalling large messages. Most importantly, the execution times have a no worse than linear dependency on the size of the messages which are handled.

By measuring how much time is spent in different parts of the unmarshalling code it is possible to point out which parts of the implementation need optimization to give the a better performance. Especially the Expat XML parser is responsible for a large part of the execution time. Exchanging Expat with a parser having an XML Pull interface will eliminate the need for at token queue. A simpler XML parser implementation may even perform better than Expat in itself, adding to the possible performance improvement. Removing the token queue from the implementation will also result in a smaller stack allocation, since the buffer it uses will no longer be needed.

The memory strategies for WSTOOL, which were introduced in Chapter 5, have been implemented and has shown to work as intended. The heap allocation requirements for WSTOOL servers are linearly dependent on the size of the data structures that are being conveyed.

Stack usage is heavily dependent on the size of the buffers used in the middleware layers. Reducing the sizes of these buffers will reduce the stack allocation, however the execution time may be increased. One possibility would be enable the user of WSTOOL to set the buffer sizes to tune the middleware layers for the particular application.

It can be concluded that the middleware code generated by WSTOOL is competitive with respect to execution time compared to the current alternatives, and that it scales

well for handling large messages. The current implementation of WSTOOL and the code that it generates does have room for improvement. Implementing the suggested changes could potentially reveal improved performance.

9 Conclusion and Future Work

This thesis is the result of a study of how the overhead associated with the abstractions provided by Web services can be minimized. A tool, named WSTOOL, has been developed, which generates skeleton and stub layers facilitating the development of service providers and service requesters in the C programming language in a transparent manner. In this chapter we draw conclusions on this work and discuss possible future work.

9.1 Conclusion

In this thesis we have continued our prior work presented in DAT5 [2], which studied the applicability of Web services in distributed embedded systems. We did this by asking the question: *How can the overhead resulting from the abstractions provided by Web services be minimized?*

We have approached this question by developing a prototype implementation of a tool, named WSTOOL, for developing Web services (service providers) and clients of Web services (service requesters) in distributed embedded systems in the ANSI C programming language. Our motivation for developing this tools has been to gain an understanding of the problem area and its requirements, and to give us a concrete platform for testing our algorithms, architectures, and designs.

Our design and implementation of WSTOOL provides two main contributions to reduce the overhead of applying Web services in distributed embedded systems:

- The design of a service specific layer, which performs marshalling/unmarshalling using specific knowledge about a given instance of a service provider or service requester.
- The design of an efficient interface, that allows representing SOAP messages in a format different than XML, using the notion of XML information sets.

We have presented a design for algorithmically generating unmarshalling layers using context-free grammars as an abstraction. This design takes advantage of the knowledge about the service provider or service requester that the unmarshalling layer is generated for, which presents a number of opportunities for optimization.

A unique context-free grammar is generated for each type of message that must be unmarshalled, and this grammar is used to generate a parser which accepts exactly the set of syntactically valid messages. Furthermore, a special memory allocation strategy has been designed and implemented to reduce the amount of memory allocation and copying performed by the service specific layer. This reduces both the amount of memory required as well as the execution time of the parser.

A central issue with Web services is in the size of the SOAP messages that are exchanged. These are represented as XML documents, which introduces a significant overhead. For that reason, an efficient interface has been developed which enables the use of other representations of SOAP messages than XML. This interface is based on the fact that the structure of SOAP messages is formally defined by an XML information set, which allows alternative serializations. The interface is based on the exchange of a limited set of simple tokens. This set of tokens is constructed from the knowledge about the SOAP messages that will be exchanged between service providers and service requesters for a specific Web service. The knowledge of which tokens can appear in valid messages limits the set of valid tokens, and provides the possibility for defining an efficient interface.

A prototype implementation of WSTOOL has been completed, using the C++ programming language. Using WSTOOL, an example Web service has been developed, which has been used as a basis for performance experiments. The same Web service has been implemented using two alternative tools, gSOAP and CSOAP. A comparative performance evaluation between the three Web service solutions has been completed. From this evaluation it is concluded that the middleware generated by CSOAP is clearly the slowest and most memory consuming. This indicates that the per service generating strategy of WSTOOL and gSOAP is preferable, when compared to the generalized approach of CSOAP.

From the performance evaluation it is also concluded that WSTOOL marshalling performance is superior to that of gSOAP for large requests. The unmarshalling performance of WSTOOL is not quite as good as that of gSOAP, however a number of suggestions for improvements have been presented, which should improve the unmarshalling performance.

Finally, the performance evaluation shows that the required amount of dynamically allocated memory is constant for a specific service, also when varying the size of the message transferred. This is made possible by the streaming architecture of the generated middleware. The amount of stack memory required depends on the size of the data structures that must be handled.

As a final conclusion on our studies of the applicability of Web services in distributed embedded systems, we conclude that Web services is an applicable middleware technology in distributed embedded systems. We consider our implementation of WSTOOL as a proof of concept, which has illustrated that the overhead associated with Web services can be reduced.

9.2 Future Work

There are several areas of interest when considering possible future work, some of which concern improving performance, and some of which concern extending the set of features of WSTOOL.

The performance evaluation revealed that a major part of the execution time of generated unmarshalling code was spent in the XML parser and in its associated token queue. Exchanging the Expat XML parser with an XML Pull based parser will eliminate the need for the token queue. Incorporating an efficient implementation of an XML parser having the XML Pull interface into the middleware generated by WSTOOL may reveal significant

performance improvements. Furthermore, another master's thesis at Center for Embedded Software Systems, is currently studying how a more efficient representation of XML information sets can be constructed. Using such a more efficient representation could be of interest to integrate in the middleware generated by WSTOOL.

Furthermore, we have used a LALR(1) parser generator tool for generating parsers to unmarshal SOAP messages. Observing that the context-free grammars that we generate to describe the syntax of SOAP messages are simple, it could be of interest to investigate if a less powerful class of parser could be applied, such as recursive descent parsers. This would allows us generating more efficient parsers, which features smaller memory usage.

In its current state, WSTOOL does not support C pointer types. A possible feature addition is to allow pointer types by letting the middleware layer dereference such pointers as needed. This would allow the use of recursive data structures, such as structs containing a member, which is a pointer to a data object having the type of the struct itself.

Another possible feature addition is to extend the mapping between C and SOAP. For generated service requesters to be able to communicate with any service provider, a complete mapping from SOAP to C is required. This requires an extended mapping of the type system of XML Schema to C. For instance, the `date` type in XML Schema could be mapped to a C struct. Such a mapping is required for all types which can be declared using XML Schema.

On a more general level, it would be interesting to investigate how a higher level of abstraction can be achieved in distributed embedded systems, such as studying how an object oriented approach could be integrated in the development of distributed embedded systems. Also, Web services allows exchanging SOAP messages in two styles, namely RPC style and document style. In this thesis we have studied the RPC style. It could be studied how document style Web services can be integrated efficiently in the development of distributed embedded systems, and how this could help achieving a higher level of abstraction.

A

Modified ANSI C Grammar

In the following, we present our modified version of the grammar for the ANSI C language. In order to avoid ambiguities, unnecessary parts of the language has been removed such as expressions, statements and function definitions. These are not needed when specifying the interface of a Web service in a C header file.

Terminal symbols in the grammar are marked with a **bold** font.

constant_expression	→	**CONSTANT**
declaration	→	*declaration_specifiers* **SEMIKOLON**
	→	*declaration_specifiers init_declarator_list* **SEMIKOLON**
declaration_specifiers	→	*storage_class_specifier type_qualifier type_specifier*
	→	*storage_class_specifier type_specifier*
	→	*type_qualifier type_specifier*
	→	*type_specifier*
init_declarator_list	→	*init_declarator*
	→	*init_declarator_list* **COMMA** *init_declarator*
init_declarator	→	*declarator*
storage_class_specifier	→	**TYPEDEF**
	→	**EXTERN**
	→	**STATIC**
	→	**AUTO**
	→	**REGISTER**

struct_or_union_specifier → *struct_or_union*
IDENTIFIER LBRACE *struct_declaration_list* **RBRACE**
→ *struct_or_union* **LBRACE** *struct_declaration_list* **RBRACE**
→ *struct_or_union* **IDENTIFIER**

struct_or_union → **STRUCT**
→ **UNION**

struct_declaration_list → *struct_declaration*
→ *struct_declaration_list struct_declaration*

struct_declaration → *specifier_qualifier_list struct_declarator_list* **SEMIKOLON**

specifier_qualifier_list → *type_qualifier type_specifier*
→ *type_specifier*

struct_declarator_list → *struct_declarator*
→ *struct_declarator_list* **COMMA** *struct_declarator*

struct_declarator → *declarator*
→ **KOLON** *constant_expression*
→ *declarator* **KOLON** *constant_expression*

enum_specifier → **ENUM LBRACE** *enumerator_list* **RBRACE**
→ **ENUM IDENTIFIER LBRACE** *enumerator_list* **RBRACE**
→ **ENUM IDENTIFIER**

enumerator_list → *enumerator*
→ *enumerator_list* **COMMA** *enumerator*

enumerator → **IDENTIFIER**
→ **IDENTIFIER EQUALS** *constant_expression*
type_qualifier → **CONST**
→ **VOLATILE**

declarator → *pointerdirect_declarator*
→ *direct_declarator*

type_specifier → **VOID**
→ **CHAR**
→ **SIGNED CHAR**
→ **UNSIGNED CHAR**
→ **SHORT**
→ **SIGNED SHORT**
→ **SHORT INT**
→ **SIGNED SHORT INT**
→ **UNSIGNED SHORT**
→ **UNSIGNED SHORT INT**
→ **INT**
→ **SIGNED**
→ **SIGNED INT**
→ **UNSIGNED**
→ **UNSIGNED INT**
→ **LONG**
→ **SIGNED LONG**
→ **LONG INT**
→ **SIGNED LONG INT**
→ **UNSIGNED LONG**
→ **UNSIGNED LONG INT**
→ **LONG LONG**
→ **SIGNED LONG LONG**
→ **LONG LONG INT**
→ **SIGNED LONG LONG INT**
→ **UNSIGNED LONG LONG**
→ **UNSIGNED LONG LONG INT**
→ **FLOAT**
→ **DOUBLE**
→ **LONG DOUBLE**
→ *struct_or_union_specifier*
→ *enum_specifier*
→ **IDENTIFIER**

direct_declarator → **IDENTIFIER**
→ **LPAREN** *declarator* **RPAREN**
→ *direct_declarator* **LBRACKET** *constant_expression* **RBRACKET**
→ *direct_declarator* **LBRACKET RBRACKET**
→ *direct_declarator* **LPAREN** *parameter_type_list* **RPAREN**
→ *direct_declarator* **LPAREN RPAREN**

pointer → **STAR**
→ **STAR** *type_qualifier_list*
→ **STAR** *pointer*
→ **STAR** *type_qualifier_list pointer*

type_qualifier_list → *type_qualifier*
→ *type_qualifier type_qualifier*

parameter_type_list → *parameter_list*
→ *parameter_list* **COMMA ELLIPSIS**

parameter_list → *parameter_declaration*
→ *parameter_list* **COMMA** *parameter_declaration*

parameter_declaration → *declaration_specifiers declarator*
→ *declaration_specifiers parm_abstract_declarator*
→ *declaration_specifiers*

parm_abstract_declarator → *pointer*
→ *parm_direct_abstract_declarator*
→ *pointer parm_direct_abstract_declarator*

parm_direct_abstract_declarator → **LPAREN** *parm_abstract_declarator* **RPAREN**
→ **LBRACKET RBRACKET**
→ **LBRACKET** *constant_expression* **RBRACKET**
→ *parm_direct_abstract_declarator* **LBRACKET RBRACKET**
→ *parm_direct_abstract_declarator*
LBRACKET *constant_expression* **RBRACKET**

start → *translation_unit*

translation_unit → *external_declaration*
→ *translation_unit external_declaration*

external_declaration → *declaration*

B

Results of Experiments

B.1 Marshalling and Unmarshalling

Table B.1 shows the results of the experiment, in which the marshalling time was measured on the servers. All measurements are in ms, and are calculated mean values based on three out of the five measured times. The two measurements not used are the lowest and the highest. For each mean value, the unbiased standard deviation is given in percent of the mean value.

Size	wstool	stddev	CSOAP	stddev	gSOAP	stddev
50	0.350	1.86	2.119	0.12	0.862	0.52
100	0.683	2.02	4.497	1.93	1.607	0.31
150	0.993	2.07	6.883	1.74	2.316	1.04
200	1.292	1.27	9.517	2.73	3.033	0.42
250	1.672	6.37	12.16	2.14	3.776	0.98
300	1.972	1.29	15.00	2.24	4.445	0.35
350	8.789	20.5	18.54	0.95	5.204	0.25
400	9.301	32.7	21.78	1.68	5.926	0.33
450	9.165	36.5	30.26	19.3	10.78	3.39
500	9.352	39.2	39.50	16.2	10.16	9.62
550	12.20	1.84	57.45	13.9	12.13	0.59
600	12.00	0.36	65.49	17.0	12.68	10.3
650	12.60	1.53	75.31	15.5	13.75	1.28
700	12.96	2.86	79.20	6.06	14.82	1.87
750	12.57	11.4	92.87	9.80	16.53	7.13
800	13.84	1.96	98.75	5.40	17.45	8.17
850	13.80	0.40	127.7	3.60	19.91	6.34
900	14.07	0.12	107.8	1.63	21.07	1.98
950	14.81	1.99	122.4	14.6	21.09	1.97
1000	14.67	0.59	146.4	3.51	23.06	0.11

Table B.1: Mean values and associated standard deviation in percent for marshalling on servers.

Table B.2 shows the results of the experiment, in which the unmarshalling time was measured on the clients. Data and standard deviation is calculated the same way as for Table B.1.

Size	wstool	stddev	CSOAP	stddev	gSOAP	stddev
50	4.400	7.42	11.64	1.29	2.196	5.76
100	5.886	4.55	16.28	4.10	2.967	37.0
150	8.117	17.5	20.72	1.88	5.037	9.30
200	9.253	4.12	25.69	0.84	5.680	0.03
250	10.64	1.64	30.48	0.71	6.879	0.12
300	12.29	1.67	35.62	0.29	7.973	0.23
350	15.06	2.82	39.22	5.56	7.359	46.6
400	16.19	1.80	46.25	0.40	10.42	0.13
450	17.72	2.78	56.94	10.5	9.263	39.7
500	19.49	1.95	66.65	7.05	12.74	0.36
550	21.18	0.35	88.52	9.48	14.16	1.78
600	22.26	1.42	96.01	9.93	15.51	2.82
650	24.56	1.19	110.3	10.7	16.23	1.77
700	26.14	1.19	116.0	3.63	18.04	1.50
750	27.94	1.71	132.6	6.98	19.22	1.55
800	29.23	0.46	139.6	4.08	16.99	30.9
850	30.64	1.63	169.2	1.24	18.12	31.0
900	32.22	2.63	152.7	1.06	19.21	30.0
950	34.44	0.39	168.3	8.95	23.83	1.68
1000	35.90	0.85	195.7	2.30	21.86	25.8

Table B.2: Mean values and associated standard deviation in percent for unmarshalling on clients.

B.2 Profiling of wstool Unmarshalling on Client

Table B.3 shows the results of the experiment, in which the elapsed time in three different components of the unmarshalling code on a WSTOOL generated client was measured. Data and standard deviation is calculated the same way as for Table B.1.

Size	Expat	stddev	Token queue	stddev	HTTP	stddev	Other
50	0.661	1.27	0.428	1.33	0.157	39.0	3.155
100	1.184	0.74	0.806	1.69	0.212	6.95	3.684
150	1.708	0.38	1.194	2.33	0.285	0.35	4.930
200	2.217	0.20	1.568	1.84	0.391	1.60	5.078
250	2.749	0.53	1.929	1.38	0.508	1.42	5.454
300	3.289	0.53	2.270	0.31	0.606	1.83	6.123
350	4.524	3.30	2.720	1.02	0.783	6.46	7.032
400	4.352	0.45	3.264	12.0	1.476	39.6	7.094
450	4.849	0.32	3.423	1.59	1.843	39.8	7.609
500	5.400	1.51	3.810	0.60	2.206	40.6	8.074
550	6.136	6.60	4.177	0.21	3.182	2.47	7.691
600	6.391	0.37	4.545	0.97	3.544	0.65	7.781
650	6.993	0.37	4.934	0.25	4.069	0.54	8.562
700	8.962	22.9	5.300	0.32	3.467	46.2	8.407
750	8.259	4.16	5.917	6.32	2.854	63.0	10.91
800	8.815	5.03	6.073	1.00	4.276	48.6	10.06
850	9.095	0.39	6.462	0.41	5.914	1.44	9.162
900	9.597	0.54	6.810	0.50	5.125	39.7	10.69
950	10.36	4.61	7.236	0.22	6.823	0.75	10.02
1000	11.06	3.20	7.489	0.36	5.725	48.9	11.62

Table B.3: Mean values and associated standard deviation in percent for profiling of WSTOOL unmarshalling on client.

C

Guide to WSTOOL

A step-by-step guide to installing and running WSTOOL is now given, along with a short description of how to generate the road direction service presented earlier. The source code for this service is installed along with WSTOOL.

C.1 Building and Installing wstool

The WSTOOL application and associated libraries are built using the GNU build system. On the CD-ROM attached found the back of the thesis a distribution of WSTOOL can be found. The source code is compressed in the file `wstool-0.9.9.tar.gz`. Follow the steps in Figure C.1 to uncompress, build, and install WSTOOL.

```
$ tar zxvf wstool-0.9.9.tar.gz
$ cd wstool-0.9.9/
$ ./configure
$ make
$ make install
```

Figure C.1: Building and installing WSTOOL and associated libraries and data files.

WSTOOL depends on the Expat and Xerces libraries, and you may need to specify paths to the location where these are installed, as well as the location of the include files for these libraries. Type `./configure --help` for help on doing this. Furthermore, WSTOOL depends on the Bison parser generator tool. Expat, Xerces, and Bison is distributed on the CD-ROM.

C.2 Generating an Example

After installing WSTOOL, an example has been installed as well. This example is the road direction service used for the performance experiments. If no special options were given to `configure`, this example should be located at:

```
/usr/local/share/wstool/examples/road_network/
```

To build and run the example, enter this directory and follow the instructions in Figure C.2. Notice that the `route_client` application will connect to `localhost`.

```
$ make
$ ./route_server
```

In another shell:

```
$ ./route_client
```

Figure C.2: Building and running the route direction example.

Bibliography

[1] Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, 1998.

[2] Michael Sig Birkmose, Lars Haugaard Kristensen, Jakob Møbjerg Nielsen, and Hans Hvelplund Odborg. Applying Web Services as Middleware for Integrating Embedded Systems in Loosely Coupled Environments. Project report, Aalborg University, 2004.

[3] C library for SOAP (Simple Object Access Protocol). Ferhat Ayaz [online, cited June 9th 2004]. Available from World Wide Web: `http://csoap.sourceforge.net/`.

[4] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, third edition, 2001.

[5] eCos development team. eCos Home Page [online, cited June 9th 2004]. Available from World Wide Web: `http://sources.redhat.com/ecos/`.

[6] Free Software Foundation. Bison [online, cited June 9th 2004]. Available from World Wide Web: `http://www.gnu.org/software/bison/`.

[7] James Clark. The Expat XML Parser [online, cited June 9th 2004]. Available from World Wide Web: `http://expat.sourceforge.net/`.

[8] Jeff Lee. ANSI C Yacc grammar [online, cited June 9th 2004]. Available from World Wide Web: `http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`.

[9] Julian Seward. Valgrind [online, cited June 9th 2004]. Available from World Wide Web: `http://valgrind.kde.org`.

[10] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1988. 2nd edition.

[11] Microsoft. Microsoft .NET [online, cited June 9th 2004]. Available from World Wide Web: `http://www.microsoft.com/net`.

[12] Asis Nasipuri and Kai Li. A directionality based location discovery scheme for wireless sensor networks. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 105–111. ACM Press, 2002.

[13] Robert A. van Engelen. gSOAP: SOAP C++ Web Services [online, cited June 9th 2004]. Available from World Wide Web: `http://www.cs.fsu.edu/~engelen/soap.html`.

[14] R. Schantz and D. Schmidt. Middleware for distributed systems: Evolving the common structure for network-centric applications [online, cited June 9th 2004]. Available from World Wide Web: `http://www.dist-systems.bbn.com/papers/2001/EvolutionOfMiddleware/evolutionofmiddleware.pdf`.

[15] Sun Microsystems. Java Web Services Developer Pack [online, cited June 9th 2004]. Available from World Wide Web: http://java.sun.com/webservices.

[16] The National Center for Supercomputing Applications. Timing, Profiling and Debugging [online, cited June 9th 2004]. Available from World Wide Web: http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/IA32LinuxCluster/Doc/timing.html.

[17] World Wide Web Consortium. SOAP 1.2 Recommendation, Part 1: Messaging Framework [online, cited June 9th 2004]. Available from World Wide Web: http://www.w3.org/TR/2003/REC-soap12-part1-20030624/.

[18] World Wide Web Consortium. SOAP 1.2 Recommendation, Part 2: Adjuncts [online, cited June 9th 2004]. Available from World Wide Web: http://www.w3.org/TR/2003/REC-soap12-part2-20030624/.

[19] World Wide Web Consortium. Web Services Architecture [online, cited June 9th 2004]. Available from World Wide Web: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[20] World Wide Web Consortium. Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language [online, cited June 9th 2004]. Available from World Wide Web: http://www.w3.org/TR/2003/WD-wsdl12-20030611/.

[21] World Wide Web Consortium. Web Services Description Language (WSDL) Version 1.2 Part 2: Message Patterns [online, cited June 9th 2004]. Available from World Wide Web: http://www.w3.org/TR/2003/WD-wsdl12-patterns-20030611/.

[22] World Wide Web Consortium. Web Services Description Language (WSDL) Version 1.2 Part 3: Bindings [online, cited June 9th 2004]. Available from World Wide Web: http://www.w3.org/TR/2003/WD-wsdl12-bindings-20030611/.

[23] World Wide Web Consortium. XML Information Set [online, cited June 9th 2004]. Available from World Wide Web: http://www.w3.org/TR/xml-infoset.

[24] World Wide Web Consortium. XML Schema Part 1: Structures [online, cited June 9th 2004]. Available from World Wide Web: http://www.w3.org/TR/xmlschema-1/.

[25] World Wide Web Consortium. XML Schema Part 2: Datatypes [online, cited June 9th 2004]. Available from World Wide Web: http://www.w3.org/TR/xmlschema-2/.

[26] Xerces C++ Parser. The Apache Software Foundation [online, cited June 9th 2004]. Available from World Wide Web: http://xml.apache.org/xerces-c/.

[27] Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. *ACM Trans. Model. Comput. Simul.*, 4(1):107–131, 1994.