MSc Distributed Computing Systems Engineering
Department of Electronic & Computer Engineering

Brunel University

DISSERTATION

PROTOCOLS AND ARCHITECTURE FOR INTERROGATION AND CONTROL OF REMOTE EMBEDDED SYSTEMS, WITH PARTICULAR REFERENCE TO GENERIC DIAGNOSIS AND CONFIGURATION TOOLS

Ismail Dagli

September / 2001

A Dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science MSc
Distributed Computing Systems Engineering
Department of Electronic & Computer Engineering

Brunel University

DISSERTATION

PROTOCOLS AND ARCHITECTURE FOR INTERROGATION AND CONTROL OF REMOTE EMBEDDED SYSTEMS, WITH PARTICULAR REFERENCE TO GENERIC DIAGNOSIS AND CONFIGURATION TOOLS

Ismail Dagli

Supervisor: Mr. Roger Prowse

September / 2001

A Dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science

Abstract

This dissertation discusses the main issues involved in remote interrogation of embedded systems and the efficient development of interrogation tools. It focuses on remote interrogation over internet infrastructure and evaluates the applicability of such an approach to the cost-sensitive and dependable-oriented domain of embedded systems. As a result of the assessment of existing technologies, standards and models, an architectural framework will be defined which supports and encourages the development of reliable, generic and remote interrogation software.

In the first part, an survey about existing distributed systems will performed to identify design issues and appropriate models for architecture, protocols and communication. The second part of the dissertation is an analysis of the design issues and the applicability of existing technologies to embedded systems. With aid of the results gained in the analysis, the third part defines and specifies an architectural framework for remote interrogation of embedded systems with particular reference to generic interrogation tools. The practical work exemplifies the development of generic and remote diagnosis, configuration and control tools using this architecture with aid of a prototypical implementation. The last part of this dissertation will assess the approach taken here due to performance and memory requirements, maintainability and portability.

Acknowledgements

I would like to thank my parents and my family for their love, support and patience during my studies in graduate school. I am grateful to my advisor Mr. Roger Prowse for invaluable guidance, encouragement and support throughout my dissertation. I am thankful to Mrs. Cornelia Weiss for her valuable advice and constructive criticism regarding my project work. I want to thank Kirsty Parker and Hakan Oezaslan for proof-reading and comments. Remaining mistakes are mine.

Contents

Summary

Service and maintenance of embedded systems software makes up a significant part of the overall development costs of embedded devices. Firstly, the service and maintenance of very specialised devices require expert knowledge about the internal structure of embedded devices and therefore a service technician to perform interrogation of devices at-site. Moreover, the development of sophisticated interrogation tools is very complex and adapting those to special versions of device software and types is difficult.

This work here addresses these problems by suggesting remote and generic tools that are independent of the device and enable remote interrogation of embedded devices. Hence, the aim of this project is the provision of an architectural framework which encourages the production of reliable, generic and remote interrogation tools. This architecture will base on internet standards because internet infrastructure provides a cost-effective way for remote access on embedded devices.

As in principle, using standard distributed systems technology is advantageous from many criterions, but the realisation of these ideas in context of performance and memory constrained embedded systems and low-quality public inter-networks is very challenging. Therefore, this dissertation will assess the applicability today’s distributed systems technology to the domain of embedded systems and will identify limitations of this approach.

1 Introduction

A general purpose definition of embedded systems is that they are devices used to control, monitor or assist the operation of equipment, machinery, or plant. The term '*embedded*' reflects the fact that they are an integral part of a system, which may comprise several cooperating embedded sub-systems, working together in order to perform the task of the overall system. The development of the embedded sub-systems requires very specialised knowledge, therefore they are mostly provided from different vendors, using specialised hard- and software.

The emergence of Internet technologies into the cost-sensitive and dependable-oriented domain of embedded systems/applications facilitate global availability that allows cost effective remote access to resources. The rapid growth of the Internet and the high commercial interest on Internet applications has lead to very innovative technologies and rapid standardisation in this area. At the other extreme, software on Embedded Systems is developed under performance and memory constraints and focuses on the functionality of the device. The configuration and diagnosis of devices requires high expert knowledge about the internal structures of the embedded system but also about the specific interrogation tool and is therefore mostly done by service technicians. Moreover, the development of sophisticated interrogation tools makes up a significant portion of the overall costs for a embedded device, as devices are very special and tools must be developed for every device type and version. Therefore, the aim of this project is to provide an architectural framework, that supports and encourages the production of reliable, generic and remote interrogation software.

Remote interrogation over the Internet will profit from the advantages provided by internet infrastructure, enabling cost effective remote access and also providing standards that support the development of interrogation tools but especially real-time interrogation as diagnosis will also suffer from the restriction given by today's inter-networks: Low bandwidth Internet connections and large unpredictable delays are common and can not be influenced because packets send over the internet travel over arbitrary real-world networks. Additionally, big headers of TCP/IP and application oriented protocols build upon them cause high overheads in communication and also implementing such protocols on memory and performance constrained embedded system sets an unique set of challenges. Another issue in using public networks for remote access on embedded systems is that it will increase the vulnerability to malicious attacks. Contemplating the drawbacks and restrictions given by the internet infrastructure it is clear that a solution for remote interrogation must consider all aspects of inter-networks and also the restrictions given by real-world embedded systems.

1.1 Context

The project was initiated by suppliers and customers of embedded systems because of the high cost of service and maintenance of devices and the complexity of the development of configuration, diagnosis and control tools. The referred devices are used in different environments, so that different versions of software (or even different versions of hardware) might be required to embed (and adapt) the system in its environment. The existing tools for configuration or diagnosis are developed special to the device and its version, therefore tools have to be modified according to the peculiarities of the device.

Device vendor companies might provide many devices which means that a large amount of effort has to be taken for organisation and development of tools in order to keep them in step with the software versions running on the embedded systems. Another drawback of existing interrogation tools is that some changes or modifications to a device require special knowledge of the design and internal structure, so that the customer can not perform them. Hence, a service technician must be ordered which means high costs and increased time to repair or change.

This dissertation provides solutions that reduce costs for both, the development of interrogation tools and the service and maintenance of devices. The first point requires that tools should be independent of the devices (generic), the latter implies remote access to devices. The term generic in this context means that the interrogation tool has no prior knowledge of the device, so that it has to adapt to the device at runtime. With that the tools are independent of the systems to be configured. The following figure illustrates a scenario for remote interrogation

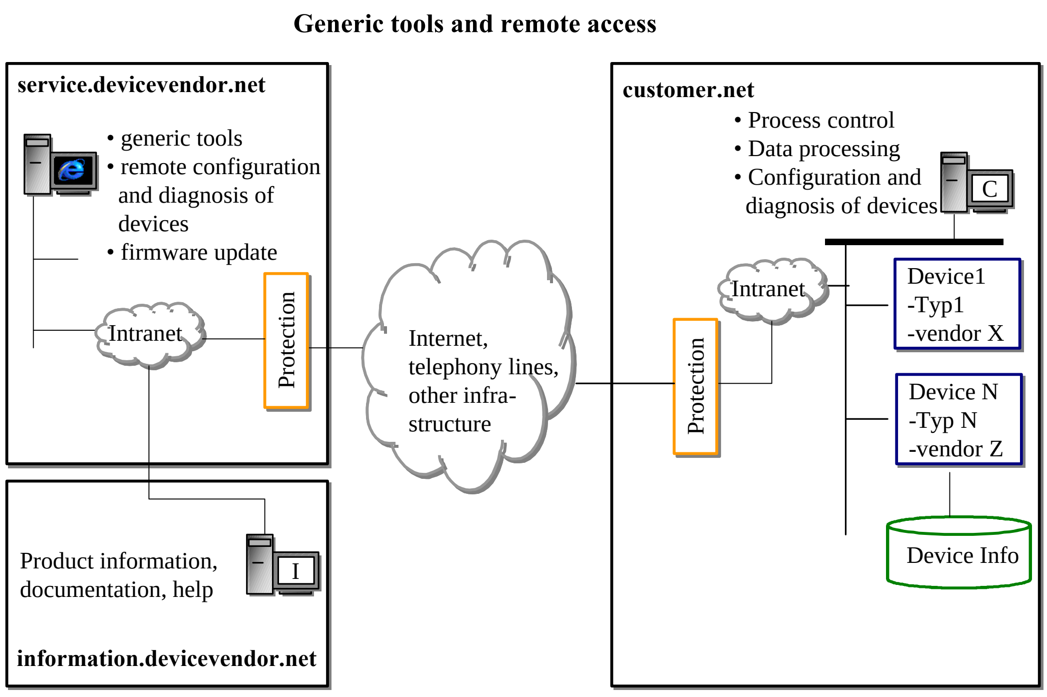

Generic tools and remote access
service.devicevendor.net
• generic tools
• remote configuration and diagnosis of devices
• firmware update
Intranet
Protection
Internet, telephony lines, other infra-structure
customer.net
• Process control
• Data processing
• Configuration and diagnosis of devices
C
Intranet
Protection
Device1 -Typ1 -vendor X
Device N -Typ N -vendor Z
Device Info
Product information, documentation, help
I
information.devicevendor.net

Figure 1-1 Generic tools and remote access

The figure shows a scenario, where several devices from different vendors are used by a customer company. The customer is provided with interrogation tools to access the devices locally over the company's intranet. The customer company uses (virtual) generic tools which are able to access all devices, no matter what type. The figure further indicates that some kind of information associated with the devices is needed which can be used by a generic tool to access the devices properly. Because the customer is not able to solve all configuration and diagnosis problems, the vendor of the device provides extra service and maintenance facilities. To decrease the cost of service and maintenance, the figure suggests remote configuration and diagnosis using the existing infrastructure of the Internet. The device vendor could provide additional services with devices, like online documentation or product information.

In contrast to this future scenario, maintenance and service of as performed today is highly inefficient. A service technician who is ordered to maintain (update software version, change settings, etc.) the device must first find out the version of the embedded software running on the device and the install the corresponding interrogation software version on her/his mobile computer. Configuration and diagnosis is then performed at the customer, which means increased costs for travelling and time to repair. If the overall system at the customer is dependent on the device, the system might be shut down for hours (or even days) which is unacceptable inmost cases.

1.2 Aims and objectives

The aim is to provide application oriented protocols and an architectural framework that supports and encourages the production of generic and remote interrogation tools for e.g. configuration and diagnosis. This requires the provision of a reasonable abstraction of the embedded systems for this purpose, the specification of the application oriented protocols and the identification of key components, providing security, transparency and openness.

One important objective is that the complexity of network communication is hidden from the interrogation tool developer. This requires that networking functionality should not reside at the application but components must be provided which enable data exchange for heterogeneous (PC and devices) systems transparently. This includes the provision of common data exchange formats, various communication modes for specific application preferences and masking of network and communication errors. Although the focus here is internet infrastructure, the application oriented protocols discussed here should be independent from the underlying network oriented technologies, because it can not be assumed that all devices are internet-capable. The efficiency of communication is the most critical part for real-time interrogation (e.g. diagnosis) of embedded systems. Therefore an extensive evaluation of existing technologies is required due to their adaptability to embedded systems with small memory and little processor performance.

Another important issue concerning the high cost of developing interrogation tools is the device independence. The objective here is to provide a reasonable device abstraction for generic interrogation tools. Important to this subject is the assessment of object oriented concepts like encapsulation, abstraction and communication over well defined interfaces in order to a provide uniform way to access the device by generic tools.

The practical part of this dissertation is about design and implement prototypes for interrogation tools , which exemplify the development of generic and remote tools, using this framework. The objective here is to assess the feasibility of the approach taken here. Hence, performance and memory requirements for the device and requirements concerning quality of service of network will be evaluated with aid of these prototypes.

2 Survey about relevant topics

2.1 Motivation

The distributed object model as described in [CDK00] and [BACON98]is very attractive because of its simplicity and flexibility. It enables all resources (here devices) to be viewed in a uniform way, which is essential to the requirement that interrogation tools should be generic. In this model, remote objects are accessed by their operations whereby the details of objects are hidden from the resource user.
In implementations of the distributed object model, the communication is integrated into a programming language paradigm and therefore is transparent for the application programmer. Remote method invocations are performed as local ones which provides access transparency and simplifies the development of distributed applications.

The basic idea here is adopt such a model in simplified form to remote interrogation of embedded systems, by the means of that the communication between interrogation tools and device is performed over remote method invocation (RMI). Hence, the following survey will focus on the distributed object model and the associated communication mechanisms and will assess the adoptability to memory and performance constrained embedded systems. Moreover, the ideas and concepts underlying object model will be evaluated in general in context of generic interrogation tools.

2.2 Approach

The approach taken here is to examine existing distributed system architectures and application level protocols with focus on '*light-weight'* and simple protocols and architectures in order to identify design issues for remote interrogation of embedded systems. The second important issue is to identify the requirements and design issues for generic interrogation tools.

There are distributed object model implementations available for PC based systems, like CORBA[1], Java-RMI or Microsoft DCOM[2], which could meet the given requirements here. Specifications for scaling down these solutions to better fit into memory-constrained

[1] Common Object Broker Architecture
[2] Distributed Component Model

embedded environments are in preparation and might be applied in future systems [EMBEDCORBA00], but at this time, none of them fit into a memory and performance constrained embedded system as required here. In this project, here these solutions are used to get ideas how to model the embedded system and allow communication for remote access, transparently.

Hence, a more '*light-weight*' protocol SOAP will be introduced which allow invocation on remote objects. It is likely, that this solution does not fulfil all requirements set by this project, but it forms a reasonable basis to define a architecture for remote access on embedded systems, since it is simple and extensible.

The first part of the survey will examine several approaches from the protocol, communication and architecture point of view. It will evaluate general requirement of remote method invocation and briefly outline how these concepts are implemented in today's distributed systems.

The second part of the survey is concerned with a reasonable device abstraction that aims to provide a uniform view of devices in context of generic interrogation tools.

2.3 RMI – Remote Method Invocation

The term RMI is used here to refer to remote object invocation in general. RMI extends the object based programming model to allow objects in different processes (or computers) to communicate with one another by means of remote method invocation. Hence, RMI is an extension of local method invocation that allows an object living in one process to invoke the methods of an object living in another process.

Software that provides a programming model above the basic building block of processes and message passing is called middleware[CDK00]. The middleware layer uses protocols based on messages between processes to provide its higher level of abstractions such as remote method invocation. Basically, from the protocol point of view these functionality are represented by the OSI (Open System Interconnection) Layers 5 and 6 ([HALSALL96], [BINSTOCK00]). Therefore, I will later refer to the OSI model, because it gives a more consistent definition from the protocol point of view as the term middleware. The term middleware might be defined from a application or infrastructure point of view [BINSTOCK00]. The first would comprise the OSI Layers 5-6 as defined in [CDK00] and

[BINSTOCK00], the latter would also consider all services which enable the distribution of components. The following figure depicts the layers involved as defined in [CDK00].

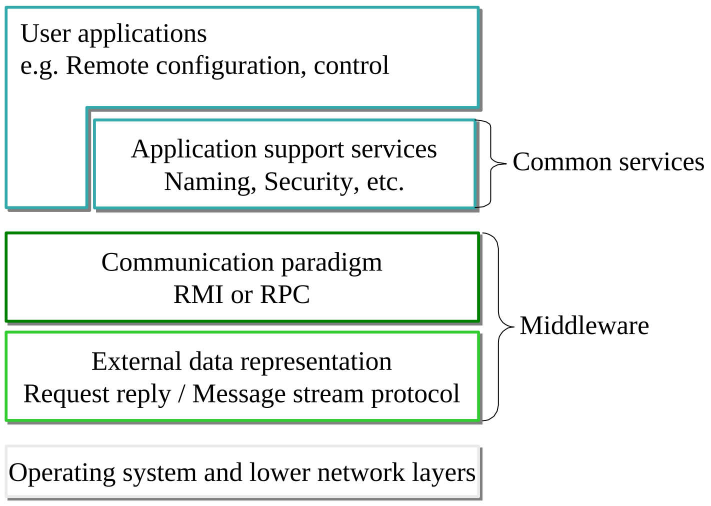

User applications
e.g. Remote configuration, control
Application support services
Naming, Security, etc.
Common services
Communication paradigm
RMI or RPC
External data representation
Request reply / Message stream protocol
Middleware
Operating system and lower network layers

Figure 2-1 Middleware layer

In the following we will examine some components and protocols relevant for a middleware layer for remote method invocation. Respecting the fact that object oriented languages and RMI are absent on embedded systems, the discussion here provides only the key design issues, which may be relevant for the design of protocols and architectures in this project.

2.3.1 Request-Reply protocol

The request-reply protocol provides the basic communication mechanism for every RMI or RPC. The protocol is very simple for straightforward communication, but gets complex if fault tolerance plays a role [CDK00], [BACON98]. The protocol is based on a trio of communication primitives: `doOperation`, `getRequest` and `sendReply`.

The `doOperation` method is used by clients to invoke a remote operation. Its arguments specify the remote object and which method to invoke. Its result is a RMI reply. It is assumed that the client, calling the `doOperation` marshals the arguments into an array of bytes and unmarshals the result from the array of bytes that is returned (see next chapter for more detail). The `doOperation` invokes a blocking `receive` operation and waits for the reply message. The `getRequest` operation is used by the server process to acquire service requests. After the method is executed on the specified object the server process invokes the `sendReply` operation:

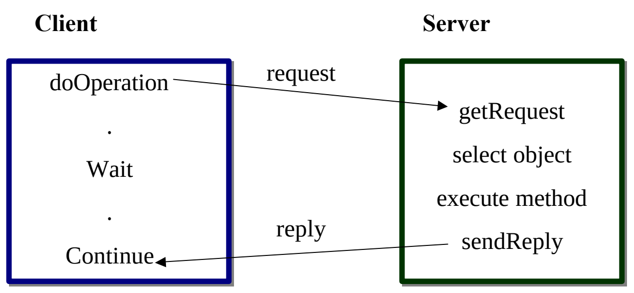


Figure 2-2 Request-reply protocol operations

Data transfer requirements vary by applications and devices, e.g. if it is required to configure a large set of configurable parameters on a device, or to update the firmware of the device the requirement is to reliably transfer bulk data in a reasonable amount of time, but if it is required to observe the device state or the produced process data for diagnosis purposes, it must be guaranteed that the diagnosis tool observes the most recent data at a rate which allows reasonable diagnostics. Because the communication links for remote access are slow compared with the rates data is produced or states are changed, it will be required to develop strategies concerning transfer of data and protocol design, which reduce the required bandwidth sufficiently. The request/reply protocol introduced here can not fulfil all requirements by interrogation applications, especially it is obvious that it does not give sufficient support for diagnosis in general. Hence, more suitable protocols must be found which coexist with the request/reply protocol to support a wider range of applications.

2.3.2 External data representation

The information stored in running programs is represented as data structures – for example by sets of interconnected objects – whereas the information in messages consists of a sequence of bytes [CDK00]. Irrespective of the form of the communication, the data structures must be flattened (converted to a sequence of bytes) before transmission and rebuild on arrival 0, 0. The individual primitive data items transmitted in messages can be data values of any type, and not all computers store primitive values such as integer in the same order. The floating point representation may differ and characters may be encoded with different codes. Two distinct methods enabling data exchange between two heterogeneous computers are given in literature ([CDK00] and [BACON98] for middleware architectures and [HALSALL96] for OSI):

- The values are converted (marshalling) into an agreed external format before transmission and converted to the local form (unmarshalling) at receipt.

- The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary.

The first alternative provides more openness and the second alternative is considered to be more efficient, as it requires only conversion at the receiver. Chapter 2.4 will introduce the external data format and serialisation in SOAP which uses structured ASCII-text (XML) as external data format. This form of serialisation will be compared and contrasted to other technologies like CORBA, DCOM or Java-RMI which use similar but not ASCII based techniques.

2.3.3 Modules and objects involved in RMI

Several separate objects and modules are involved in achieving a remote method invocation. The next figure gives an overview of the components involved in RMI [CDK00].

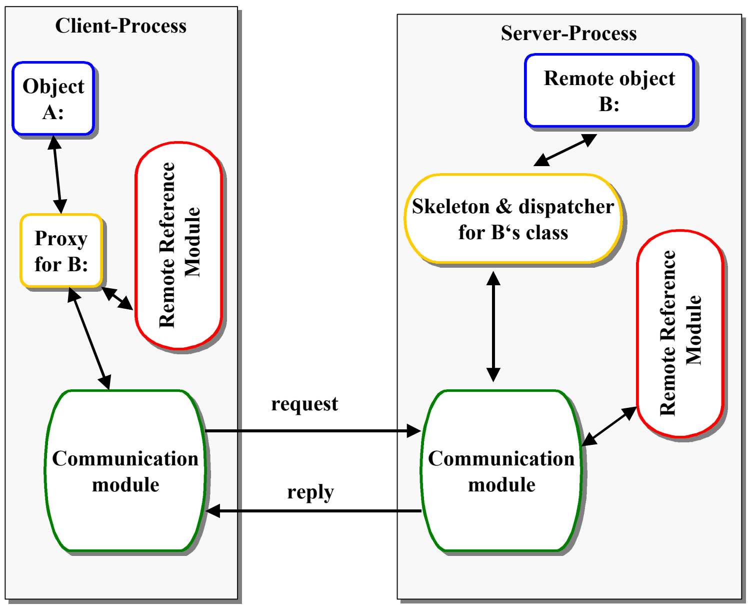


Figure 2-3 Modules and objects in RMI

The modules and object are explained here very briefly. Please refer to [CDK00] for a detailed discussion.

- The communication modules carry out the request/reply protocol as described in the last chapter
- Remote Reference Module manages remote object references (unique identification of the remote object) . This includes the generation of unique remote object references and the mapping to local object references.

- The proxy or stub for a remote objects provides transparency for the client object. The client object invokes methods on the proxy, which then carries out the remote invocation . This involves the marshalling of arguments and unmarshalling of results.
- The skeleton and the dispatcher resolve the remote object's reference with aid of the remote reference module and invoke the specified method on the object. First the dispatcher selects the corresponding skeleton of a class and transfers the arguments and the methodID (unique identification of the method within a class) to the skeleton. The skeleton unmarshals the arguments and invokes the specified method. If the method returns a result, the skeleton marshals it before passing it to the communication module.

A system which provides RMI must define such modules or objects, or at least must provide components, which perform similar functionality. So this section is very important from the design point of view, if a general mechanism is required to access remote devices. To make the discussion more concrete SOAP will be introduced as a simple (light-weight) approach to RMI. SOAP will be compared with more '*heavy-weight*' approaches like CORBA or Java-RMI, which define not only RMI, but a full architecture for distributed systems and hence will be important from the architecture point of view. Important to this discussion here will be the assessment advantages and disadvantages of approaches to RMI in context of embedded systems, in order to support design decisions later on.

2.4 SOAP - Simple Object Access Protocol

With CORBA, DCOM or Java RMI, it is required to install the proper runtime environments, have the users configure their systems to accommodate the distributed infrastructure, and administer the systems in addition to managing your application's needs. Also, Firewalls will likely to be reconfigured to allow the system-specific packets to enter and leave the local network, which is a major issue for remote configuration and diagnosis. In that sense, these distributed architectures are "***heavy-weight***" systems, making the decision to use them and their protocols an expensive one in many cases[SCRIBNER00].

SOAP provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML. SOAP does not itself define any application semantics such as a programming model or implementation specific semantics; rather it defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules.

2.4.1 Design goals of SOAP

The original SOAP specification outlines two major design goals (from the 1.0 specification [W3CSOAP1.0]):

> *"Provide a standard object invocation protocol built on the internet standards, using HTTP as the transport and XML for data encoding."*
>
> *"Create an extensible protocol and payload format that can evolve."*

The 1.1 specification simply states[W3CSOAP1.1]:

> *A major design goal for SOAP is simplicity and extensibility."*

This means that there are several features from traditional messaging systems and distributed object systems that are not part of the core SOAP specification[SCRIBNER00]. Such features include.

- Distributed garbage collection (orphaned objects)
- Bi-directional HTTP communications (callbacks)
- Boxcarring or batching of messages (Pipelined messages or multiple call requests)
- Objects-by-reference (which requires distributed garbage collection)
- Remote object activation (which requires objects-by-reference)

The areas the specification does not address are left to the individual SOAP implementation's architect to design into specific implementation. This suites well to the requirements of remote access on embedded systems, because SOAP defines only a lean, lightweight object access protocol and additional services and modules may be added only if necessary for this purpose. With that, it is possible to design a architecture around SOAP, which is efficient and fits into the constraints of embedded systems.

2.4.2 Advantages and disadvantages of SOAP

This chapter will list some advantages of SOAP that make it a reasonable protocol for remote access on embedded systems. But it also shows the limits of SOAP and outlines extensions which may overcome these restrictions.
Using SOAP as a object access protocol has the following advantages:

- SOAP is built upon open technologies, rather than vendor-specific technologies and facilitates true distributed interoperability. No single vendor dominates the SOAP market, at least not at this time.
- SOAP typically uses HTTP but does not require its use, at least not with the 1.1 version of the specification. This make it possible to use SOAP over more complex communication protocols, than the simple request/reply approach given in HTTP. For remote diagnosis for instance it would be beneficial to have conversation like mechanisms as defined in [CDK00] and [HALSALL96].
- SOAP will likely work out-of-the-box in a wide range of locations that enable HTTP port 80 Post access. This probably the most important advantage of SOAP, since today's security architectures will only allow access to internal resources over standard, text-based protocols.
- Changes to the SOAP infrastructure will likely not affect applications using the protocol, unless significant changes are made to the SOAP specification.

As mentioned above there are also disadvantages and restrictions associated with SOAP.

- SOAP was initially tied to the HTTP protocol. This mandated a request/reply architecture that is not appropriate for all situations. Hence, most SOAP implementation use HTTP, and if more complex protocols are required, they must be defined and implemented.
- There are no true validating XML processors with full Schema[3] support , hence general-case method de-serialisation must be done by discovery [SCRIBNER00], [MEAN00].
- SOAP serialises by value and does not support serialisation by reference at this time. With that, multiple copies of the object will, over time, contain state information that is not synchronised with other dislocated copies of the same object.

[3] In order to place constraints on the data, or to facilitate typed data, DTDs or Schemas can be used to provide data about the data –this is called metadata. Although DTDs are beneficial to many XML applications they do not have the characteristics necessary for describing constructs such as inheritance or complex data types. At the time of this project XML-Schemas specification was only available as a draft paper.

- ASCII serialisation is not efficient in terms of computation time required for marshalling and unmarshalling and bandwidth required for messages.

Whether or not the applications of using SOAP outweigh the disadvantages will likely depend to a great degree upon the requirements set upon the distributed architecture. The most important advantage for SOAP is that SOAP communication modules can be self-implemented for the embedded system, containing only the most important functionality of the specification. Considering porting of heavy-weight solutions to embedded systems it is clear that this is not feasible with today's embedded systems. Hence, SOAP is considered to be the most appropriate way to implement remote object access on embedded systems.

2.5 Generic tools interrogation tools

The following chapters examine the distributed object model in context of generic tools. Important to this discussion here is the device abstraction, which is needed to access device uniformly. This issue will be further discussed in chapter 5.4 of the analysis and in chapter 6.3.3 in context of the architecture specification and design.

2.5.1 The device abstraction

The concept introduced here assumes that a device is able to introduce or publish its interfaces for configuration, diagnosis and control to other applications or tools. The basic idea is to abstract a device to a configurable, controllable and diagnosticable component, which provides various interfaces for different purposes. The following figure shows an abstract view of a device as used here.

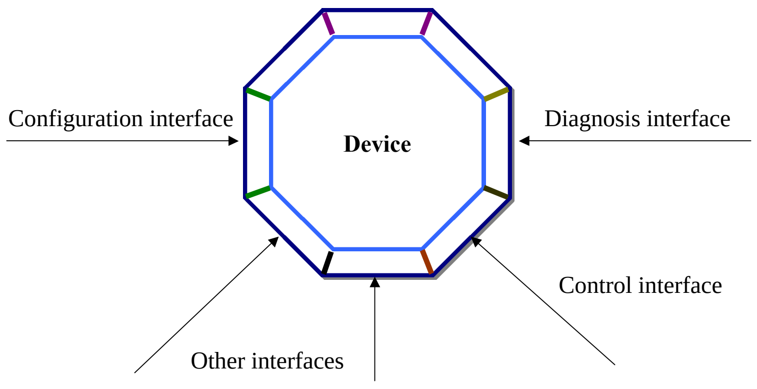


Figure 2-4 Device abstraction

For this purpose, an efficient way is needed to describe the interfaces, configurable parameters, and diagnosticable states and process data of a device. A similar concept like in

CORBA with the IDL (Interface Definition Language) can be used to describe the components and their operations residing on a device.

To describe data and states, the approach of SNMP(Simple Network Management Protocol) can be adopted. SNMP uses a tree structure to manage parameters of network devices [COMER00]. SNMP is very simple and can be used for a wide range of networking devices because it provides only the simple and general methods *get* and *set* on all data and states. Both approaches: IDL and the management of parameters, states and process data in tree structures like in SNMP are essential for the development of generic tools and will be introduced in chapter 5.4.1 and 5.4.2.

2.5.2 The notion of adaptive stubs

A generic interrogation tools, which is independent of the device has no prior knowledge about it. There is major problem which arises from this fact, because the communication modules residing at the interrogation tool must be able to adapt to the device at runtime, by the means of that it has to invoke the correct operation together with the right arguments. But every device is different and interrogation services might provide different operations. In the following, a conventional way is shown for creating stubs and skeletons for client and server objects as applied in many RMI implementations.

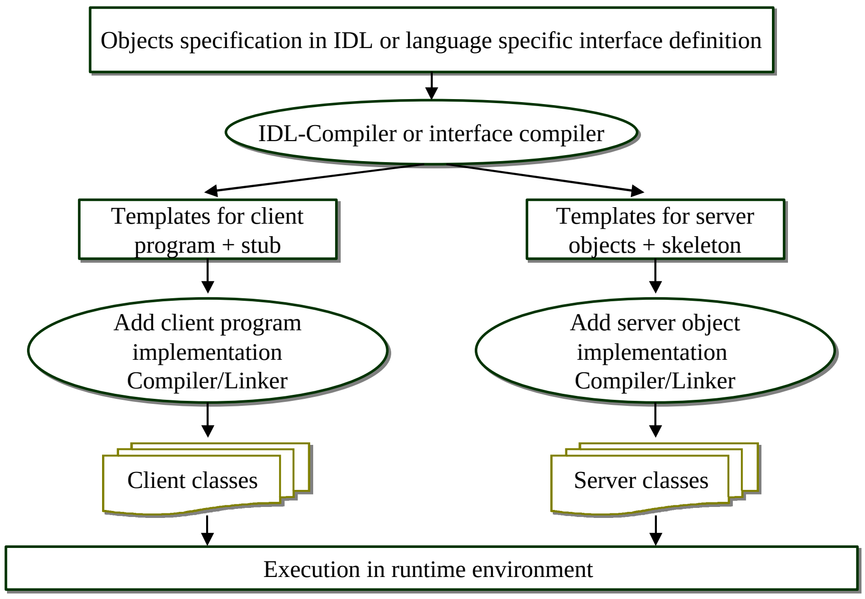

Objects specification in IDL or language specific interface definition
IDL-Compiler or interface compiler
Templates for client program + stub
Templates for server objects + skeleton
Add client program implementation Compiler/Linker
Add server object implementation Compiler/Linker
Client classes
Server classes
Execution in runtime environment

Figure 2-5 Example development process

The example development process shows that the stubs and skeletons are created according to a interface definition. Stubs and skeletons must be regenerated if the interface definition changes. In this project the generic interrogation tool is intended to work with several devices providing various interfaces to access the objects on a device. Hence, if the approach above is followed, the interrogation tools must be equipped with all possible stubs to communicate with different devices. The following figure illustrates the approach where each skeleton is associated with one stub at the interrogation tool.

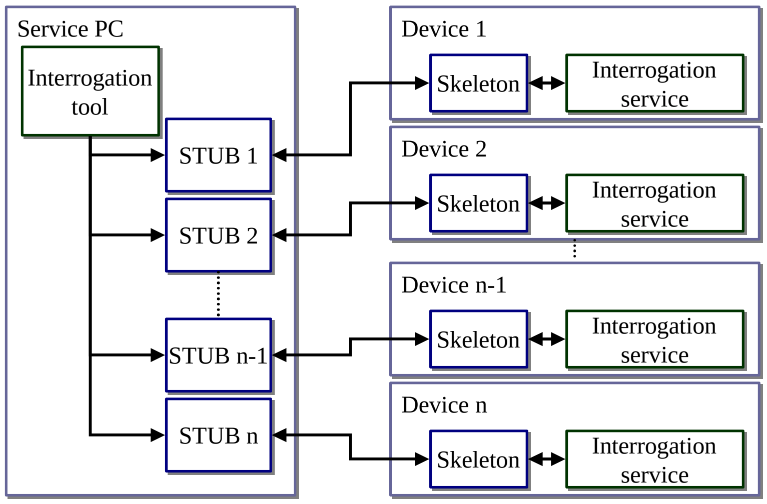


Figure 2-6 Multiple stubs and skeletons

The stubs in this scenario could be created once automatically and stored with the device. Using a programming language as Java for the interrogation tool, which allows download of programs at runtime, these stubs can be downloaded from the device to the interrogation tool when required.

A more flexible approach is to use a configurable and adaptive stub, which is able to work with the skeletons (or communication modules) on the device without prior knowledge. The basic idea here is that the stub gets meta-information about the provided operations and the required arguments (data objects) at runtime and creates RMI-requests (e.g. using SOAP) dynamically according to this meta-information. The following figure illustrates this alternative approach.

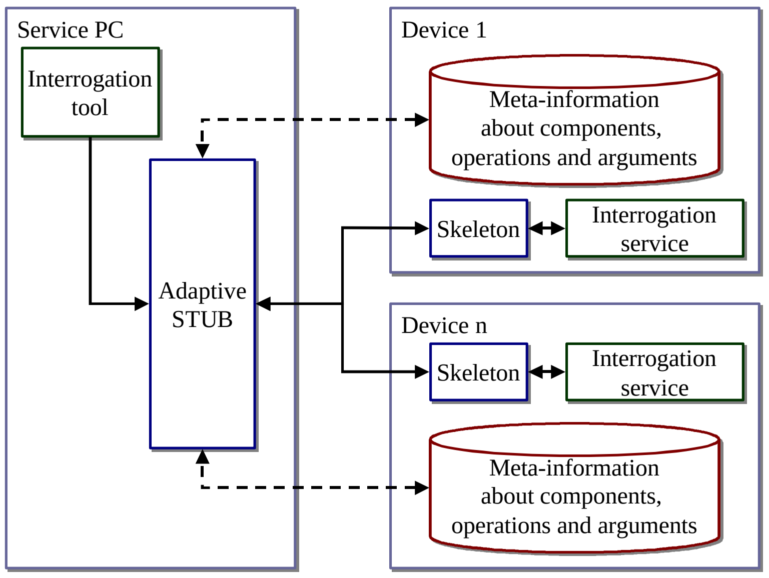

Service PC
Interrogation tool
Adaptive STUB
Device 1
Meta-information about components, operations and arguments
Skeleton
Interrogation service
Device n
Skeleton
Interrogation service
Meta-information about components, operations and arguments

Figure 2-7 Adaptive stubs

The adaptive stub acquires the meta-information about interfaces and data objects at runtime and assembles the messages for RMI-request dynamically. This is similar to the dynamic invocation interface used in CORBA[OMGCORBA01], where the meta-information at the device represents a interface repository. The advantage of this approach is the flexibility and simplicity which it achieves. It provides real-independence but at the expense of complexity in developing adaptive stubs.

Both approaches above do not consider algorithms required in the interrogation tools to make service requests dynamically. Before the adaptive stub can dynamically assemble RMI-request messages and send them to the skeleton located on the device, the interrogation tool must initiate the service request by calling methods on the stub. But if the services on the device change the interrogation tool must also adapt dynamically. This issue is addressed by dynamic graphical user interfaces discussed in chapter 7.3.2. The basic idea is to map the meta-information into an appropriate graphical user interface and let the user select operations and arguments at runtime.

2.6 Summary

RMI provides a very robust and reasonable way for communication and provides a better abstraction for the programmer than simple message passing. With SOAP, a light-weight RMI implementation is given, which is also suitable for embedded systems. ASCII-text serialisation as suggested in SOAP requires high computation time and is associated with large overheads but represents the one and only feasible approach for interrogation over internet, because today’s security architectures will only allow HTTP port 80 access to internal resources. Using more efficient serialisation which is not based on ASCII-text alternatively to SOAP will not work because Firewalls will only allow access to internal resource over standard protocols, which are based on ASCII-text. This issue will be further examined in chapter 5.1.

The object-based model for resources is flexible and simple because it allows uniform access to devices. The three most important concepts in object orientation are abstraction, encapsulation and communication over well defined interfaces, which achieves decoupling of objects. This concepts are applied here to achieve a generic approach for interrogation tools. Chapter 5.4 in the analysis will discuss the design issues for generic tools in more detail and chapter 6.3.3 will specify and define information required by generic tools.

3 Methodology

3.1 Basic approach

This project here is a feasibility study which outcome will be further refined and deployed into real world embedded systems. The company which initiated this project produces various barcode readers, rotor scans, light-barriers and other related devices for industrial automation systems and is one of the market leaders in this sector. Since the approach here is generic, a discussion about configuration, diagnosis and control of such complicated devices in this dissertation is not serving to assist the discussion about interrogation of embedded systems in general. Hence, the analysis, specification and design will use examples for a simple virtual HVAC controller device, to assist the discussion about design issues and prototypes. The solution provided for this virtual device will be ported to a barcode reader device for assessment of portability, performance and memory requirements, maintainability etc. in order to give a more realistic feasibility statement. Therefore, the prototypes and tests in the analysis, architecture specification and design must be seen in the context of a virtual HVAC controller application using the target platform and the assessment later on in chapter 8 will examine the solution for real-world embedded system applications.

3.2 The software development process

The software development process here is performed after the spiral model, as discussed in [SOMMER01] and [PRESSMAN01]. The approach taken here for protocols and architecture are associated with major risks (see chapter 5.6). The spiral model is appropriate for such projects as it allows the customer and developer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as risk reduction mechanism but, more important, enables the developer to apply the prototyping approach at any stage in the evolution of the product[PRESSMAN01].

Using the spiral model, software is developed in a series of incremental releases. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered systems are produced.

A spiral model is divided into a number of framework activities, also called task regions [PRESSMAN01]. Typically there are between three and six task regions.

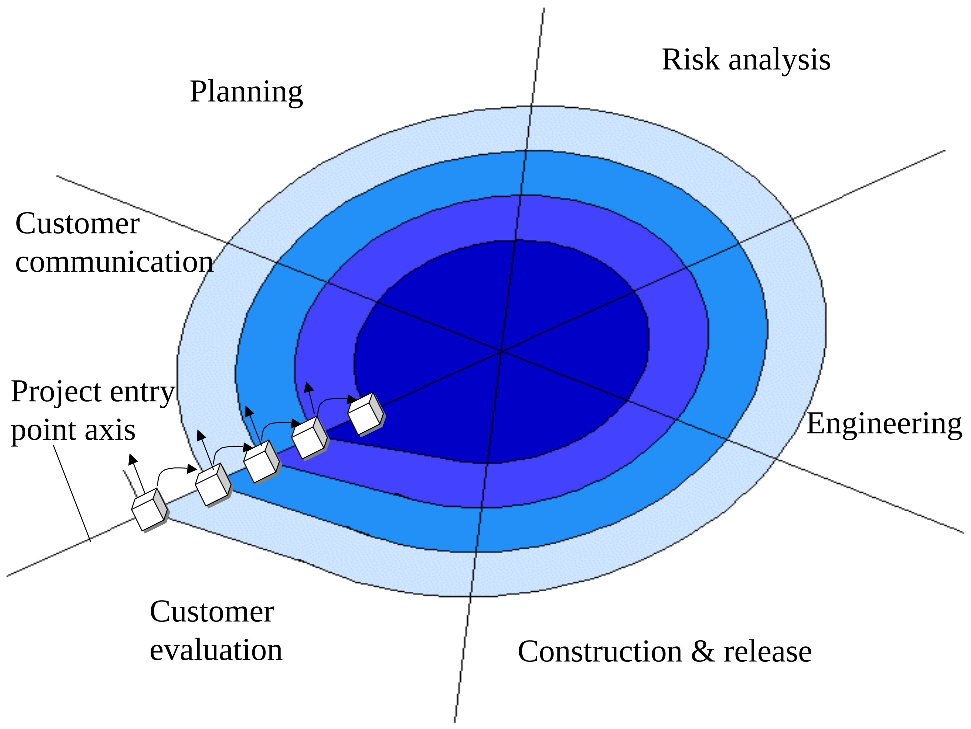


Figure 3-1 The spiral model

The cubes in the project entry axis represent sub-projects. The cubes in the inner orbits in the spiral model represent concept development and new product development phases and the cubes in the outer orbits represent product enhancement and maintenance projects. Each rotation is associated with a deliverable which can be a document (specification, design, test plan) or a prototype in the inner orbits and implementations, running software releases or maintenance concepts in the outer obits. Most important to this dissertation here is that planning is preformed for each sub-project and prototyping for risk analysis is integral part of the model. This allows an early assessment of the intended approaches. Moreover planning for smaller sub-projects helps to manage complexity and increases reactivity in case of hold ups and problems. Another point that was successfully used in this project is graceful degradation or loop backs as indicated with the arrows from outer to inner cubes. During the development some points are identified to be inappropriate, so that an additional concept development (inner orbit) phase has been added. In literature the spiral model is considered to be too complex to manage for real-world projects because of its evolutionary nature and the large set of framework activities. This can not be confirmed here, at least not for this project. Compared with other models it suits best to research or feasibility projects: As the projects evolves the knowledge of the developer does and with new concepts and ideas another phase in a inner orbit (conceptual phase) can be started even if design and implementation have been performed partly.

3.3 Problems and limitations

The approach taken here with SOAP serialisation is not the most efficient one but as shown in the survey part and examined in the analysis part it is the only feasible for remote interrogation over the internet, because today's security architectures will only allow access to internal resources (devices) over standard, text-based protocols. ASCII-based serialisation requires higher bandwidth and computation overhead than other approaches. Direct modem access without using the internet infrastructure will allow compression of messages, as we will see in chapter 6.2.5, which can reduce the bandwidth requirements. Other enhancements like buffering or various transmission strategies (as discussed in the same chapter) will further mitigate the bandwidth and delay problem but communication (and also the device performance) will remain a bottleneck. Especially for real time diagnosis (with high data rates, e.g. sending a bunch of data every 10msec) over low bandwidth communication links where guaranteed quality of service is absent, this means, that the solution here will provide only a nice add-on to existing local interrogation solutions (see analysis and assessment part). Hence, this work also will show the ranges in quality of service of networks, device performance and memory where this approach is applicable.

3.3.1 Strength of the dissertation

From the company's point of view, the notion of remote and generic tools is very attractive in terms of cost reduction for development of tools and service/maintenance of device. With the intended solution here especially customer satisfaction will increase because it provides a standard way for device configuration and diagnosis and decreases time to repair/maintain devices. Moreover, it provides a way for the device vendor to distinguish the device from similar competitor products, which do not provide remote access facilities.

Important for the dissertation here is that the project requires research in many different areas. It might seem to be straight-forward to make the system work, but there is a high potential for improvements. Therefore, high background knowledge in distributed systems architectures, communications systems and networks as well as in distributed systems and related topics as compression and information security will be required to build an optimal solution. The practical part of the project provides the opportunity to apply the knowledge gained during the project and the lectures on these topics in context of embedded systems, considering performance and memory aspects, as well as network communication constraints.

4 Requirements for protocols and architecture

The requirements can be derived from the aims and objectives defined in chapter 1.2. The requirements specified here refer to the specification of architecture and protocols and not to the development of software prototypes discussed in chapter 7. The aim of this project is specified in chapter 1.2 as:

> *The aim is to provide an architectural framework that supports and encourages the production of reliable and generic remote configuration and diagnosis software.*

The requirements derived from this aim can be partitioned into requirements for remote interrogation, for generic tools, and requirements which are specific to this project here. This partitioning is performed in the following sections.

4.1 Requirements for remote interrogation

100 The architecture and application oriented protocols must support a wide range of remote interrogation software for embedded systems.

101 The protocols should be independent of underlying network oriented layers.

102 The protocols should facilitate remote access on embedded systems over various infrastructures (internet, telephony etc.)

103 The overall system should be usable for various scenarios and topologies.

104 The protocols and software components should be efficient and must fit into the memory and performance constraints of embedded systems.

105 Network oriented functionality should not reside on the applications itself. A reasonable programming model for communication should be provided.

4.2 Requirements for generic tools

106 The architecture must support generic interrogation tools, which are independent of the device and the device version.

107 The defined services and components on a device should be efficient and must fit into the memory and performance constraints of embedded systems.

4.3 Requirements special to this project

The project is an initial feasiblity study and is intended to analyse the requirements of remote access on embedded systems. Besides, the intention is to evaluate standard internet technology for this purpose. The following requirements must be seen in this context:

108 The interrogation tools should run on a standard web-browser.

109 The interrogation tools should allow dynamic interaction with the user.

110 Prototypes must be written for defined components. The development of generic and remote interrogation tools must be illustrated with prototypes.

111 Software must be assessed on maintainability, performance, portability to other target embedded systems and extensibility.

112 Limitations of the approach taken here must be shown

The requirements for the software developed here will be further refined in chapter 7.

5 Analysis

5.1 Topologies and scenarios

This chapter examines basic topologies for remote configuration and diagnosis. The emphasis here is to illustrate the constraints for remote access on embedded devices and to evaluate advantages and disadvantages of different alternatives.

5.1.1 Remote access using Internet infrastructure

Remote interrogation of embedded systems using the internet infrastructure provides a cost efficient and widely usable approach. Compared with other alternative as direct modem access it is cheaper and if we assume that customer and device vendor networks are interconnected with high bandwidth links, it is also faster.

Providing access to internal resources, as devices in the customer network, is a potential risk, because the resource itself could be attacked, or the access facilities could be misused to access further internal network resources which carry sensitive information. Hence, companies use firewalls, proxies and other mechanism which allow controlled access of internal resources from the internet. Typically, access to computers or resources within the internal network is prohibited, and only dedicated hosts and applications (services) may be accessed from outside. This scheme is very sensible from the security point of view, but will also aggravate the remote access on embedded devices. The next figure illustrates how typically embedded devices are interconnected at the customer network and depicts the difficulty of accessing them.

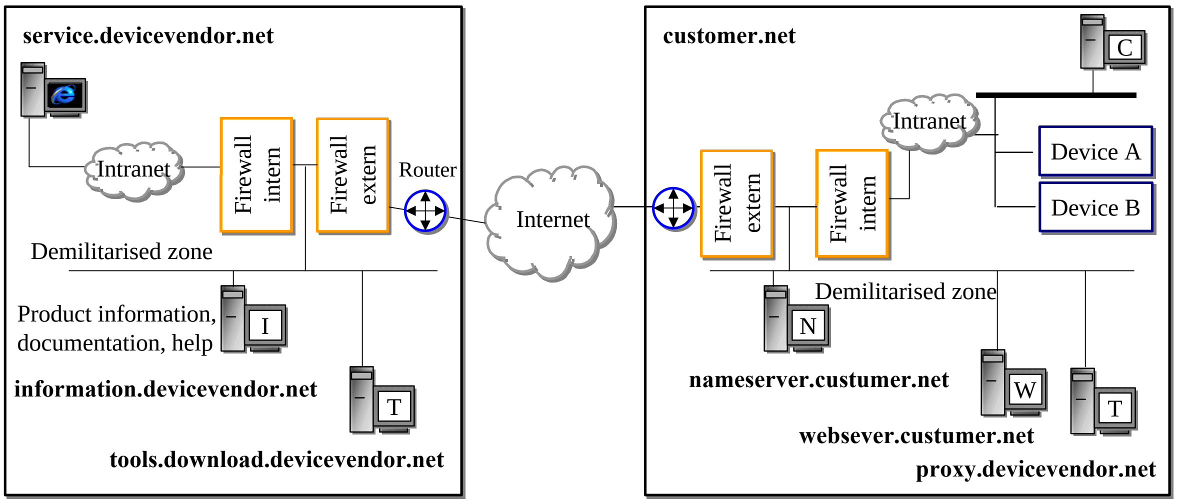


Figure 5-1 Basic topology

The figure shows two firewalls at each side, protecting the internal networks - an internal and an external. The external firewall allows only access to resources or hosts residing in the demilitarised zone and only over certain ports. The services like web-service or file services available to the outside world reside in this demilitarised zone[POHLMANN00]. Resources protected by the internal firewall are typically not accessible from outside. Devices which reside at the customer's network are protected by the internal firewall and therefore not accessible by a service technician in the device vendor network.

One possible solution could be to reconfigure the internal firewall, to gain access to the devices inside the customer's intranet, but the customer might not agree with for good reasons. Firstly, reconfiguring the internal firewall to allow access to internal resources is indeed a decrease in security, and secondly, the customer might not be able to reconfigure the firewall, which means that extra costs for firewall service is required. A better alternative would be to install a application gateway in the demilitarised zone for remote configuration and diagnosis. With that, the service technician in the vendor's network would first connect to the application gateway and this would transmit the packets to the corresponding device. Besides the application gateway could provide an extra level of authentication and confidentiality, by checking access permissions and encrypting messages. Nevertheless, the external firewall must be reconfigured anyway to allow access to the application gateway. If SOAP and HTTP is used this is less critical because only port 80 access will be required.

With the scenario described above, it is required to make some modifications to the customer network to access device from the internet. Without, an application gateway as described above it is not possible. Extra effort for the development of an application gateway and the reconfiguration of the external firewall. Alternatives must be found if modification required here at the customer's network are not acceptable.

5.1.2 Remote access over internet using a modem

An alternative is to install a modem in the customer network which can be used by devices to dial a internet service provider (ISP), in order to be independent of the peculiarities of the customer network. The device could be plugged to the modem only for the time the configuration and diagnosis takes place. The next figure shows how remote access using internet infrastructure can be provided without using the customer network. The device B in the figure is connected to the customer's network to communicate with other devices or hosts internally, but it uses a second link over a modem to access the internet, which can be used for remote service and maintenance.

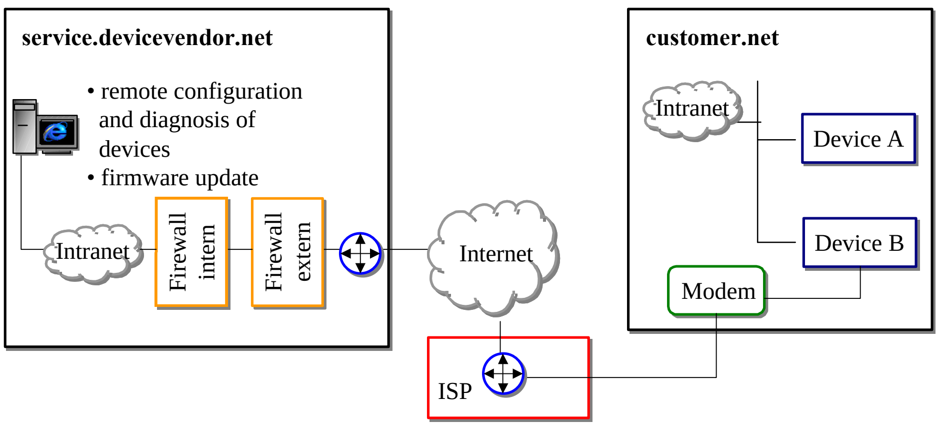


Figure 5-2 Remote service and maintenance over internet, using a modem

The problem with this, is that every time the device accesses the internet over a internet service provider, its IP-address will change, unless a fixed IP address is reserved. The service technician can not know to which IP address the device is connected. This can be solved by either allowing the device to connect to the service technicians computer, which requires that the firewalls at the device vendor's network must be reconfigured, or the IP address must be manually told by phone.

This approach requires some manual interaction, firstly as discussed above, if the IP address is acquired from the ISP this must somehow transferred to the service technician, and secondly at the customer side someone must connect the device, which is to be configured, to the modem. Another drawback is that an additional modem must be provided, which means extra significant cost, especially if only few devices are used by the customer. One possible advantage is also that the TCP/IP stack is not necessarily required, reducing memory and performance requirements on the device, if an internet modem is used which includes a TCP/IP stack. There are few modems available which already have a TCP/IP stack, but the cost for such modems is even higher. The cost for dialling the ISP is negligible, because only a local call is required and the device must only be connected to the internet during service and maintenance. Another disadvantage is that the modem approach will give less bandwidth as in the approach given in the last chapter. Modern companies are interconnected to the internet with high bandwidth communication links of several megabits per second, whereas modern modems typically provide less than 64Kbits/s. Nevertheless, this approach avoids changes to infrastructure of the customer, which is very important goal, in order to convince the customer of this product.

5.1.3 Direct modem access

The approaches discussed in the preceding two chapters assume a full TCP/IP implementation running on the embedded device. Some devices will not be able to run such a complicated stack because of restricted memory or performance. Besides, optimised TCP/IP implementations for embedded systems are very expensive and require month to port the implementation to the target embedded system. The approach discussed here does not requires a TCP/IP implementation and hence is cheap and is appropriate even for very small devices. The idea is to use modems at both sites and access the device directly, by using telephony lines. The following figure illustrates this approach.

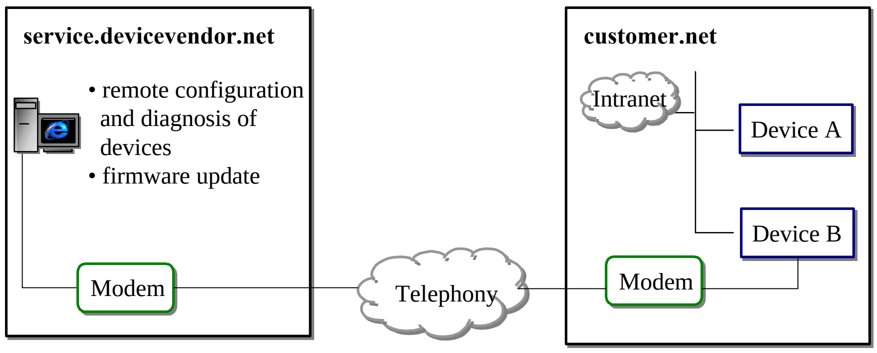


Figure 5-3 Direct modem access

A very fast and lean protocol could be defined instead of TCP/IP in this case which carries SOAP traffic, therefore this solution will perform better than the one introduced in the last chapter. The major drawback of this approach are the costs for interconnection, because standard telephony lines are used. If the devices are distributed over the whole world, inter-continental telephone connections will cause significant costs for service and maintenance. Hence, additional to the costs of extra modems, telephone fees make this approach less attractive. Nevertheless, this approach is the most secure approach, because it uses a dedicated channel for remote access, and it also does not require any modifications on the customers network.

5.1.4 Private Networks Interconnection

Assuming that the given approach for remote access on embedded systems over the internet will be widely accepted and used in e.g. the automation or car industry, the limited address space given by IP version 4 prohibits that each internet-capable device is equipped with its own IP-Address. Besides, for security issues some companies could require that their devices are interconnected with the internet only for the time of configuration and diagnosis, and additionally requiring privacy of exchanged data. The concepts given here solve the problem of limited address space and offer increased functionality in the form of privacy that prevents outsiders from viewing the data.

5.1.4.1 VPN – Virtual Private Networks

If a company comprises multiple sites, or in general if confidential data has to be exchanged between two sites and if it is also required for these sites to access services from the internet, a two level architecture can be used, which distinguishes internal or confidential data from external data. This kind of topology is often called hybrid network, allowing the establishment of private network but also providing access to the internet [COMER00]. The following figure shows a hybrid network.

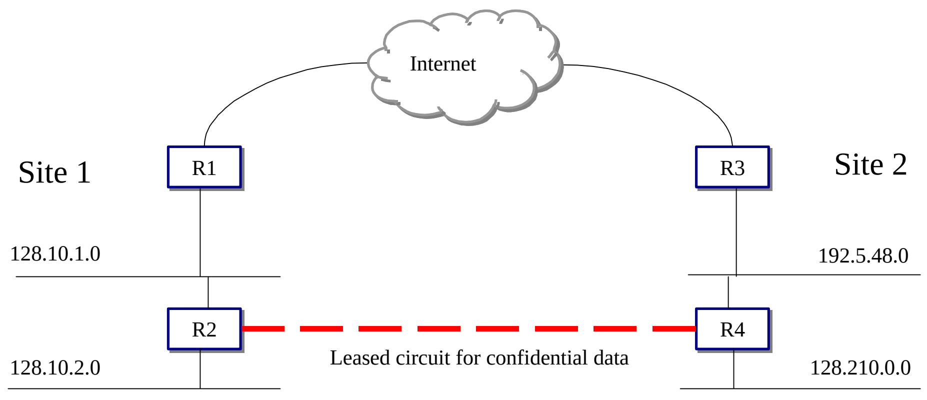


Figure 5-4 Hybrid networks

The chief disadvantage of the hybrid scheme arises from the high cost of leased and dedicated circuits. VPN (Virtual Private Network) provides a low cost alternative for transmitting confidential data, by using the internet but encrypting messages before transmission. AVPN is private in the same way as private networks, because the messages are concealed from outsiders. The term virtual denotes that VPN does not use leased circuits to interconnect sites.

For detailed discussion about security issues in VPN, see [COMER01], [POHLMANN00]. Here, I will discuss VPN from topology and private addressing point of view.

A VPN can be configured to run with private addresses on each site, requiring only one global IP-address for each site. This mode addresses the problem of the limited address space in IP. Hence, only the edge router is equipped with global IP-Address and the hosts use private addressing. The next figure illustrates this approach.

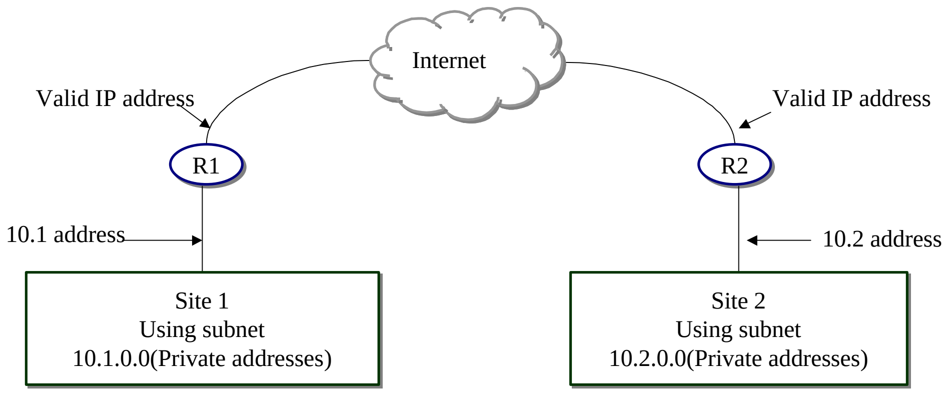


Figure 5-5 VPN with private addressing scheme

The figure assumes that both routers R1 and R2 have prior knowledge of the private addressing on each site, so they can encapsulate packets addressed with private addresses into packets with global addresses of the corresponding routers. But it might be possible that one site uses global addressing and has no prior knowledge about the private addressing on the other site. Hence, we need an address translation scheme which supports private addressing in general. This point is discussed in the next section, where NAT (Network Address Translation) is introduced.

5.1.4.2 NAT - Network Address Translation

With NAT, a technology has been created that solves the general problem of proving IP-level access between hosts at a site and the rest of the Internet, without requiring each host at the site to have a globally valid IP address [COMER01]. The technology requires a site to have a single connection to the global Internet and at least one globally valid IP address. The global address is assigned to a computer (a multi-homed host or router) that connects the site to the Internet and runs NAT software. This computer is often referred as *NAT box*.

The NAT box translates the addresses in both outgoing and incoming datagrams by replacing the source address in the outgoing datagram with the global IP address and replacing the destination address in each incoming datagram with the private address of the correct host.

Thus, from the view of an external host, all datagrams come from the NAT box and all responses return to the NAT box. From the view of internal hosts, the NAT box appears to be a router that can reach the global internet. Hence NAT technology provides transparent IP-level access to the internet from a host with a private address.

The overview of above omits an important detail because it does not specify how NAT knows which internal host should receive a datagram that arrives from the internet. Actually, there are several modes or configurations of the NAT box. Basically, the differ in how translation tables are build and how connections are identified. [COMER01] provides a full discussion about all approaches.

5.1.4.3 VPN and NAT for remote access on embedded devices

The next figure shows topology the resulting for remote access on embedded systems, using VPN and NAT.

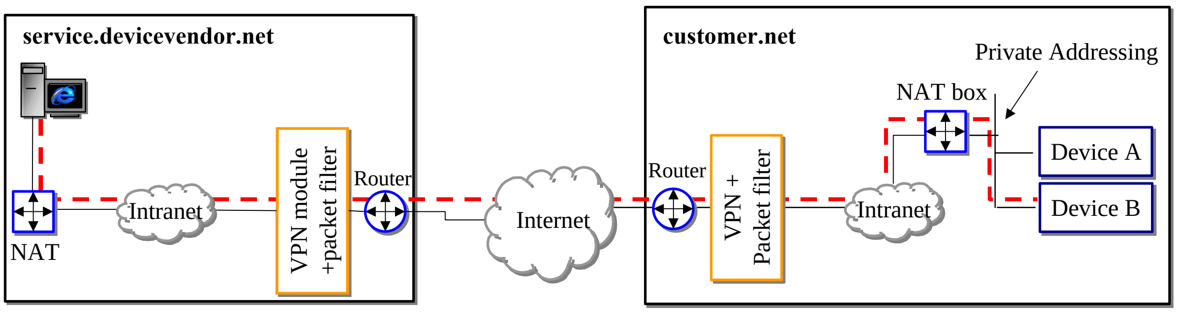


Figure 5-6 VPN and NAT for remote access and private addressing

The figure shows that the customer uses a private network to interconnect the devices. The thick dashed line illustrates the path of the packets between the host of the service technician and the device. The customer offers an access point to her/his local network over a VPN module, which is responsible for encrypting messages and also filtering packets. The service technician's host uses the same private addressing scheme as the customer for the devices. The private addresses are translated into global valid IP-addresses at the NAT boxes. The packet exchanged between the service technician's host and the device is additionally encrypted using e.g. IPsec [POHLMANN00]. Authentication headers could be included to the packets to ensure that nobody else can access the customer's network except the service technician.

The major disadvantage of this approach is that additional hardware and software has to be installed for VPN module and the NAT box, unless these components are already in use for other purposes. At this time there are only few companies which posses a infrastructure as shown in the figure above, but more and more provide their employees the opportunity to access the local company network from outside the company, using similar mechanisms like discussed here. Hence, if such a infrastructure is available, remote access on embedded devices will become more secure and easy to set up.

5.1.5 Summary

The solutions provided here for architecture and protocols must work for any infrastructure discussed in the preceding chapters in order to be widely applicable, because different companies will have different preferences and requirements. Some of the alternatives shown here have advantages in bandwidth but require changes in customer networks, other as direct modem access are simple to set up but require extra hardware (modem). Hence, there is no best topology for this project but the best fit alternative must be chosen for the specific needs of the application and the customer.

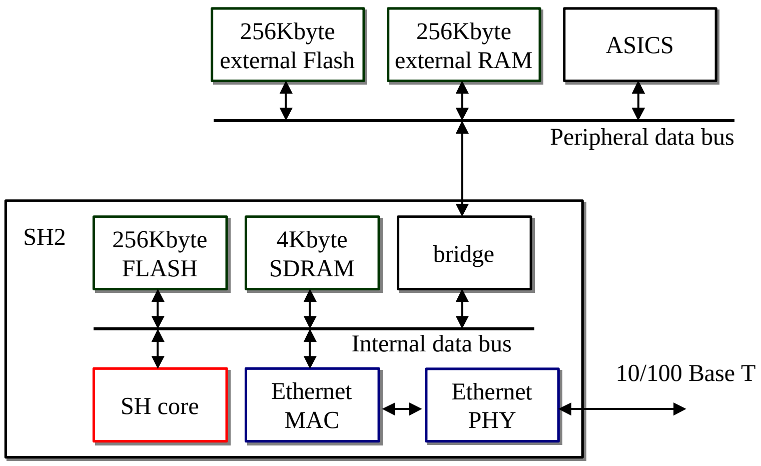
5.2 Target embedded system

There are a wide range of embedded systems, varying in memory size and performance. Typically, the embedded systems are categorised in small-scale and large-scale embedded system. As a general rule, the small-scale embedded systems are typically employed in consumer products and produced in much greater number and more cheaply than the large-scale systems, which are mostly used in industrial applications. The emphasis in large-scale systems is therefore on reducing development cost, while in the case of small-scale systems the emphasis tends to fall on lowering the cost of producing the end product.

The TCP/IP protocol stack has taken on increasing significance in the field of embedded systems, recently. Though there are problems associated with the stack, which make it inappropriate for small-scale embedded systems. The problems arise because of its large overhead and the necessity of dynamic memory management within the protocol stack. Hence, remote interrogation of small-scale embedded systems using the internet is not practicable.

The focus of this work are large-scale embedded devices with at least 256Kbyte Flash and RAM Memory, which perform better than 30MIPS (million instruction per second). Such devices are typically used in the car industry or for process control in the automation industry, but also mobile devices like PDA’s satisfy these memory and performance requirement. The development of remote interrogation software - especially over the internet - for smaller device is not practicable at this time, because a very optimised TCP/IP implementation will still require a significant fraction of flash memory. Furthermore, serialisation in ASCII-text as suggested in SOAP is slow and will require a minimum CPU-performance to work appropriately. In order to get the system working, a relatively powerful embedded system platform is chosen for the initial project here. At the end of the project, an estimate will be given, which pinpoints the minimum requirements for a device, that can support the architecture and protocols discussed here.

The remote configuration and diagnosis in this project is carried out on a embedded device which is equipped with a Hitachi SH-2 processor (32bit RISC processor performing 30 MIPS), 256Kbytes internal flash, 4 Kbytes internal RAM and 256Kbytes external Flash and RAM memory. Therefore, all memory and performance assessments must be seen in the context of this hardware. The block diagram and the micro controller unit capabilities are shown in the next figure.



Processor	Bus Size (data/address)	Clock	Memory Interface	IO Pins	ADC (res/channels)	Comms	Power (Volts)
Hitachi SH2	32/32	30Mhz	DRAM, SRAM, SDRAM	70	10bit/8ch	SCI (serial controller interface)	5/3

Figure 5-7 Target embedded system

Important to the discussion about the performance and memory is the operating system. In this project a very specialised, self-written operating system is used, which supports multiple tasks (actually there is one task running multiple threads), shared memory, synchronisation mechanism like semaphores and mail-boxes. It is a very lean operating system, which can performs task switches in the order of several microseconds. The discussion later about embedded system performance must consider the fact that the operating system here is very optimised and specialised, and that running this solution here on more advanced operating systems will require even more resources .

5.3 Protocols and communication

Chapter 2.3.1 has shown that data transfer requirements vary by applications. For instance diagnosis will require time-based transfer of small packets, whereas configuration requires only reliable transfer. The following chapters will examine design issues of application oriented protocols needed here, analyse the underlying network requirements and will assess performance and bandwidth requirements with initial prototypes.

5.3.1 Underlying network layers

As shown in chapter 5.1 internet infrastructure is only one alternative for accessing remote devices. Hence, the application oriented protocols for remote interrogation of embedded systems discussed here, must be usable independently from the network oriented layers.

HTTP defines TCP as underlying transport layer. This is suitable for configuration, which requires reliable bulk data transfer. Diagnosis might also be performed in offline mode where the data is collected over a certain period of time and observed afterwards. For analysing the real-time behaviour of process data and states, offline mode - which is less critical to implement - is unsuitable. In this real-time mode it is required to analyse the behaviour of the device to some interaction initiated by the user of the diagnosis tool.

If TCP is defined as underlying transport protocol, error correction by retransmission of packets will lead to, that the continuous data stream is interrupted. Besides, overhead introduced by mechanisms like acknowledged communication concerned with the reliability of the transport protocol will introduce further delays and slow down overall diagnosis performance. Hence, UDP is more suitable for diagnosis because it contains less overhead. Running HTTP over UDP is possible but the communication can suffer from various failure modes like omission failures, delivery of duplicates or out of order delivery. Some of these modes are tolerable for diagnosis and some should be prevented. Therefore, application oriented protocols can not assume reliable communication. Fault tolerance and masking network errors is a major issue for the protocols defined here. On the other hand, including such mechanisms in the application oriented layers, when a reliable transport protocol like TCP is used would mean that functionality and also overhead is doubled, unnecessarily.

Therefore, a flexible scheme is beneficial, which allows to customise the protocol functionality, either by negotiation between sender and receiver, or manually by the user.

5.3.2 Design issues for application oriented protocols

As shown in the last chapter the application oriented protocols for remote interrogation must support various applications, must cope with different transport layers and must provide several options for optimisations.

The following table summarises the design issues for the protocols introduced here. Some of the design issues will be addressed by SOAP (and HTTP), but since SOAP is a simple and extensible protocol, additional functionality will be defined if necessary.

Design issue	Approach
Providing serialisation for simple and compound data types.	SOAP defines serialisation for simple types as well as for complex types as arrays or lists. Compound types can be defined and serialised using XML-Schemas or DTDs.
Providing fault tolerance and masking network errors.	Using extended information in the SOAP-Header and exception handling in communication software modules. This will be discussed in chapter 6.2.3.
Synchronisation of communication peers.	Extended protocol information in the SOAP-Header.
Performance optimisations to reduce bandwidth requirements and overall communication latency.	Consideration of compression and buffering techniques. This is addressed by chapter 6.2.5.
Application specific optimisations.	Real-time diagnosis requires careful consideration of which data is transmitted and when. For example, process control systems may involve periodic data but also data, which is produced on unpredictable times. Hence, careful analysis of the possible kinds of diagnosis data must be performed. This is also addressed in chapter 6.2.5.

Table 5-1 Design issues for application oriented protocols

Some of the design issues like masking network failures will affect the SOAP messages, requiring additional information in the message. Others must be considered in the design of the communication modules. The design issues and corresponding approaches are further discussed in chapter 6.

5.3.3 Initial prototypes for communication modules

Two possible bottlenecks are identified during the analysis phase here, which might become a problem especially for diagnosis, where data must be transferred at certain rates and with maximum predictable delays.

- Bottleneck device performance:
  In diagnosis the application performed by the embedded system (e.g. process control) runs in parallel with the communication. Communication involves overheads for marshalling and unmarshalling of arguments and processing header information. The performance of the embedded system will be critical in this context, because it will restrict the data rates with which data can be produced, processed and transfered.
- Bottleneck communication:
  If the device is able to perform the application on the device in parallel to the activities required for diagnosis, then the available bandwidth will restrict the data rates with which data can be observed by the diagnosis tool.

To mitigate the risk that overall performance will not be sufficient to perform diagnosis kind applications, initial prototypes has been build which are used to identify bottlenecks and show limitations of this approach. The strategy of the evaluation is as follows: Firstly the communication is performed on a LAN with 10Mbit/s to identify the maximum data rates the device can produce. Secondly, the bandwidth is reduced step by step to identify the minimum bandwidth requirement for reasonable diagnosis. In this measurements, we do not consider delays and variation in delays.

For both measurements we use a very simple but representative diagnosis example for a HVAC device: The diagnosis software requests the temperature observed by the HVAC device. The temperature is transmitted at certain rates from the device to the diagnosis tool. The data rate is varied to identify the maximum data rate where temperature values are lost because the communication task can not process the messages.

Before explaining the software prototypes on the device and the PC, the SOAP reply message is shown, to illustrate the overheads in ASCII-based serialisation as used in SOAP:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schmemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle=
                "http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header>
    <q:ValueID xmlns:q="Reply-URI">
      7
    </q:ValueID>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <get xmlns=" HVACHeatControllerInterface ">
      <result> 37.5</result>
    </get>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 5-1 Example SOAP message used for communication module prototypes

The temperature supplied by the device is given in a range of 0.0-99.9 requiring 3 bytes in ASCII for representation. The associated SOAP message is given in Listing 5-1, including a valueID, making up a SOAP message size of 380 ASCII-character. The valueID is included to find out how many messages are not processed by the communication task.

Note, that the following measurements and the evaluation of the performance requirements are special to the microcontroller, the operating system, the embedded TCP/IP implementation and the settings used for the TCP/IP stack (e.g. large sliding window size). They provide merely a estimate, whether the target embedded system can provide enough performance for remote diagnosis using TCP/IP or not. Besides, they provide a reproducible approach for evaluating system requirements for future projects, but the measurements should not be used to predict the feasibility of other embedded system platforms. Hence, similar measurements should be carried out, if this approach for remote access is used for other embedded devices.

The prototype requires two tasks on the embedded device – one dummy task which creates a random temperature value, and one task which carries out the communication using TCP Sockets – and a program running on a PC which collects the temperature values and evaluates how many temperature values has been missed. The next figure illustrates the experiment set up.

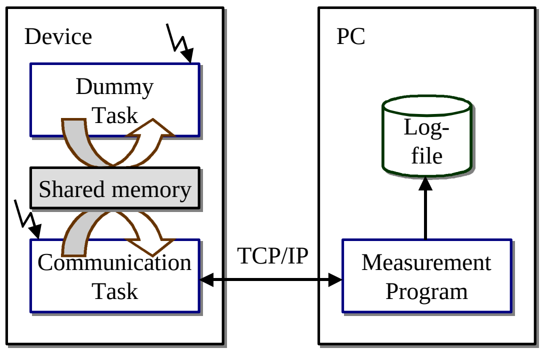


Figure 5-8 Performance test of device

A self-written and very optimised operating system is used to run the tasks on the device, which performs task switches in the order of several microseconds. The dummy task is higher prior than the communication task. The dummy task produces a temperature value (simple modulo addition) and adds increasing IDs to it. The communication task takes this value at the same rates, converts it to ASCII, packs it in an predefined SOAP message, and transmits it to the measurement program running on the PC (written in C). Note, that the communication module here is not able to create arbitrary SOAP messages, but only the predefined message in the listing above. Therefore a generic SOAP module will be much slower. The measurement program counts missing valueIDs and logs them to a file afterwards. Because the communication task is less prior, it will miss some temperature values, if the computation time available for the communication task falls below a certain threshold.

The measurements have shown that the device is able to produce, process (converting integer values to ASCII and packaging in SOAP messages), and transfer (TCP/IP communication) with rates less than 3ms. This is of course a best case scenario, where the (process control) application – the Dummy task – requires minimum computation time (several microseconds). Hence, the data rates, with which data can be transferred (assuming infinite bandwidth) will highly depend on the processor time required for real-world embedded system applications and the processor time left for communication (also considering operating system overheads).

More important to this discussion here is the bandwidth required to transmit the SOAP messages. As shown in Listing 5-1 a SOAP message for a single temperature value requires 380 bytes, which is a enormous overhead. The TCP/IP and MAC overhead requires another 58 bytes (20 bytes TCP-Header + 20 bytes IP-Header + 18 bytes MAC header and trailer), making up an overall packet size of 438 bytes on the wire. To transmit packets of the given size every 10ms a bandwidth of:

$$B = message\_size \times message\_rate = 438 bytes \times 8 bit \times \frac{1}{10 ms} = 350{,}4 Kbit/s$$

Formula 5-1

is required. The bandwidth requirement given here will be satisfied by a LAN, or if the service technician's network and the customer's network are interconnected over fast and dedicated inter-network links. But in most cases (using ordinary internet connection or modems) this bandwidth will not be available. Hence, in most cases the critical bottleneck is the bandwidth available for remote access but not the additional computation time required to implement more sophisticated protocols.

5.4 Design issues for generic tools

Chapter 2.5 has shown that a reasonable device abstraction is essential for development of generic tools. It pointed out that the device is accessed by the interrogation tools over well defined interfaces and that interfaces and data descriptions must be provided by the device, so that the interrogation tool can adapt to the device at run-time. The following chapters address these points and define how interfaces and data can be described in context of SOAP.

5.4.1 IDL – Interface Definition Language

CORBA provides a Interface Definition Language (IDL), which is a descriptive language (not a programming language) to describe the interfaces being implemented by the remote objects. As with the IDL in CORBA, XML-Schemas can be used in conjunction with SOAP to describe methods and primitive or compound data types. Because XML-Schema specification is only available as working draft at the time this work takes place, there are no validating parsers available and changes may occur in the specification more frequently. DTDs (Document Type Definition) language is a reasonable alternative to XML-Schemas [MEANS01], because validating parsers are already available and the specification has been already published. DTDs provide less functionality than XML-Schemas, but they provide sufficient support for this project. With a DTD, we could specify how a interface definition in XML must look like, so that tools could interpret them uniformly. This approach is taken in chapter 6.3.3 where a IDL is defined with aid of DTDs and interfaces are described with XML according to the IDL definition.

5.4.2 DIB – The Device Information Base

In analogy to SNMP (Simple Network Management Protocol), which defines a Management Information Base (MIB), we can describe configurable parameters, diagnosticable states and process data in a device information base (DIB). The SNMP MIB is represented with management information tree (MIT) [COMER01]. Similarly, the Device Information Base can be described in a device information tree (DIT). Every XML document has a tree structure so that XML is also suitable to build a DIB. To enable configuration with generic tools, each device has to define and store its DIB in memory and supply it if required. The next figure shows an example device information base represented by a tree.

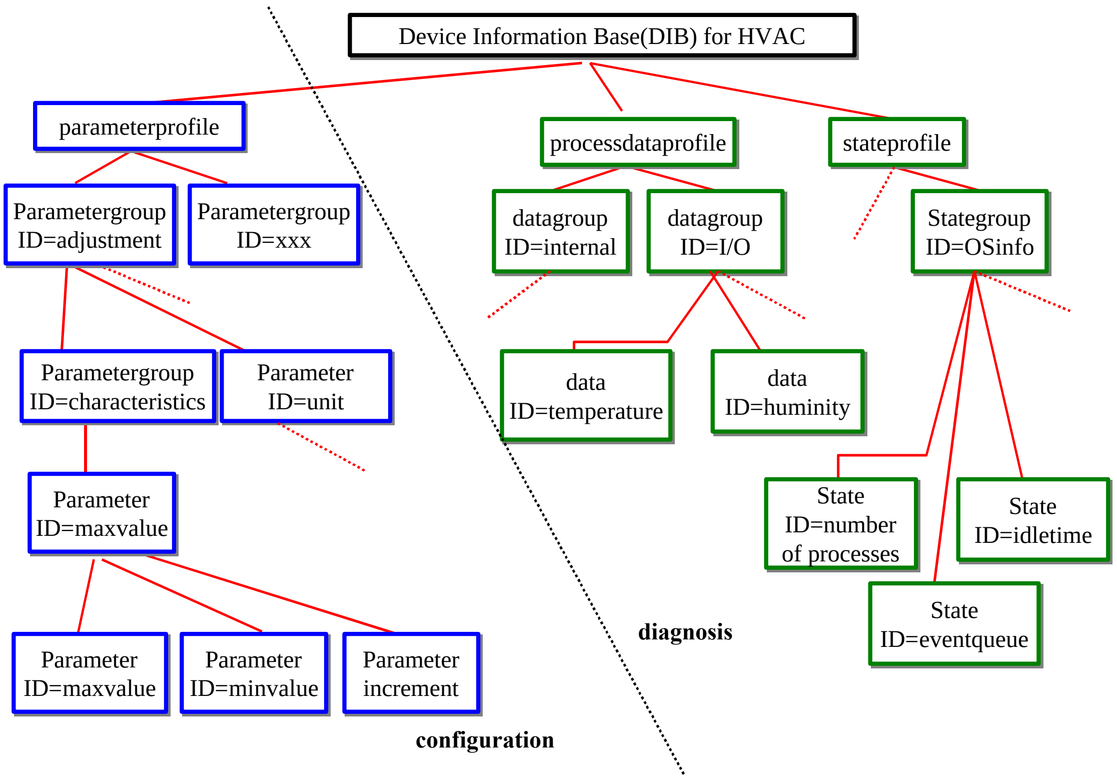


Figure 5-9 Example DIB of a HVAC controller device

The leaves in the DIT represent parameters, states or process data which generally referred here as device data objects or DIB objects and are defined as follows:

1. Parameter objects represent configurable data objects within the device. For instance, for a HVAC device, a parameter object could be the temperature reference value.

2. Process data objects represent e.g. I/O data or generally data which is produced by the application during processing. The temperature value which is measured by a sensor of the HVAC device is an example for process data objects.

3. State objects represent internal states in a device. Each data objects has a state, so it could be modelled by a state object. But with internal states we here refer to major states of a device. In case of the a HVAC device we could describe states of I/O resources, or process states etc.

The other nodes represent a grouping of these object. A DIT entry about a parameter, state or process data object has a unique ID in its context and could contain descriptive information of the object which can be used to populate the user interface of the corresponding tool. This is further discussed in chapter 6.3.3.

5.4.3 Adaptive stubs

A tool which has no prior knowledge of the device can acquire the DIB and the interface description of a device and supply the information to the user with an appropriate user interface, dynamically. With the interface description of the last section and the DIB here, the user can select the operation and the operation parameters (DIB objects) in the user interface and initiate the remote method invocation on the device. An adaptive stub as discussed in chapter 2.5.2 can pack the given operation and the parameters in a SOAP request message with little modifications. The following figure depicts how a adaptive stub can assemble request messages, when the user has selected the operation and the arguments.

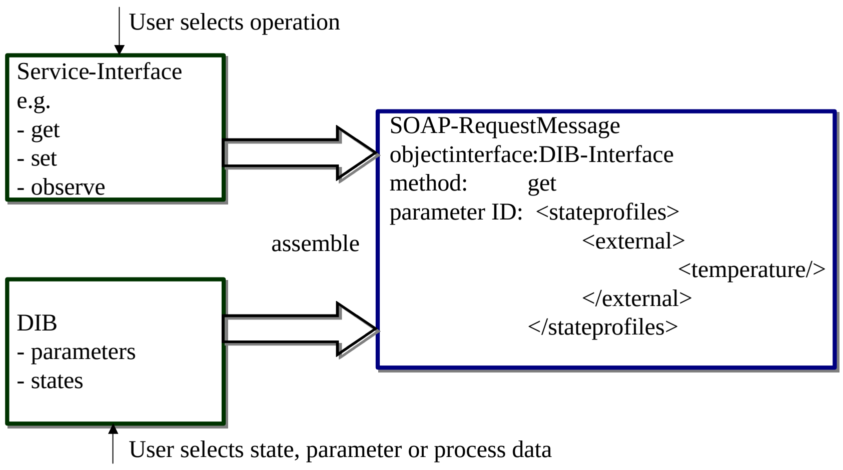


Figure 5-10 Assembling request messages from interface definition and DIB

5.4.4 Summary

To allows access to a device in a generic way, the preceding chapters have identified major information which describe the device and is needed by a generic interrogation tool:

DIB: The DIB is the representation of data and state objects residing in the main software components (see chapter 6.1) of a device, which can be accessed remotely by a generic interrogation tool.

IDL: The IDL as defined here defines how a interface description must look like. The interfaces definition (ID) of major components/services of the device can be used to populate user interfaces and assemble request messages dynamically with aid of a adaptive stub. Some interfaces (of services) will define only common operations on the DIB to set, get or observe data object. See chapter 6.1 for further discussion.

5.5 Approach

There are two distinct approaches, which differ in how the generic software is loaded and how the device information is acquired. The first is to store the device information (which contains DIB and interface definition) and interrogation tools on the device, which will require a large amount of storage space, the second is to provide this information on a dedicated server, which sets additional requirements on the availability of this server. Both approaches will be discussed here, and a third approach will be introduced which is in-between these two, acquiring some information from the device and the rest from the server.

5.5.1 Device information stored on the device

A user (service technician or customer) who wants to access the device remotely needs the generic interrogation software (tool) and must be able to access the device information somehow, to initialise the generic tool. This chapter suggests that both the tools and the device information is stored with the embedded device and that the software is loaded dynamically in form of an applet. This requires only a standard browser to be installed at the user’s host, but requires that an (micro) embedded web server is installed on the device, and that the tools are implemented with standard web-technologies as HTML and Java-Applets. The following figure illustrates the remote access process.

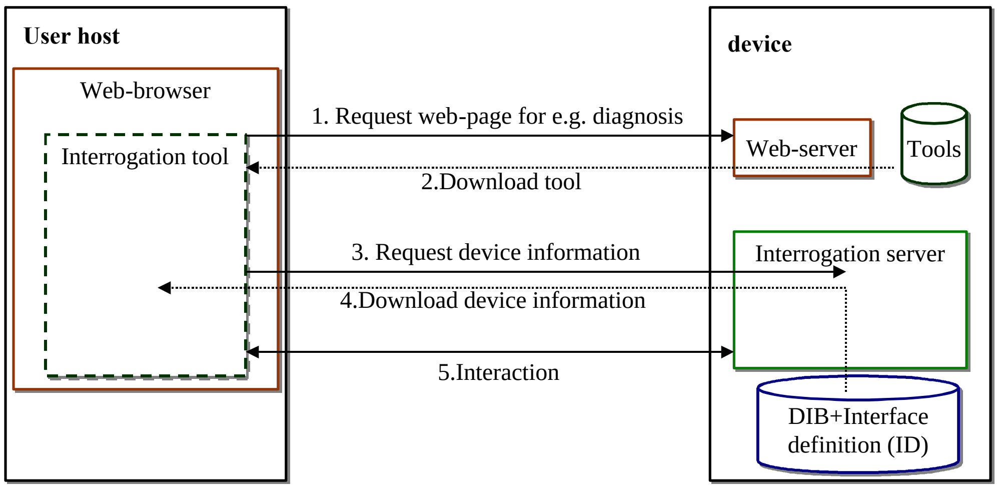

User host
Web-browser
Interrogation tool
device
Web-server
Tools
Interrogation server
DIB+Interface definition (ID)
1. Request web-page for e.g. diagnosis
2.Download tool
3. Request device information
4.Download device information
5.Interaction

Figure 5-11 Tools and DIB stored with the device

The process is defined as follows:

4. The user requests the tool (e.g. diagnosis or configuration) located on the device by requesting the URL of the tool from the embedded web-service. The interrogation software (tool) is HTML-page with an embedded applet.

5. The web-server delivers the a HTML page with the embedded (applet).

6. The interrogation tool automatically connects to the device and requests the DIB (in form of an XML-document) of the device.

7. The interrogation servers (e.g. diagnosis server, configuration server) running on the device returns the device information (XML)

8. The generic tool populates its user interface with the information in the DIB+ID. The user sends commands to the device and observes e.g. data and state values.

The advantage of this approach is that the user needs nothing more than a standard browser and is able to perform configuration and diagnosis to any device. This approach is also suitable for scenarios where internet access is absent, as with the direct modem access approach. The disadvantage here is obviously the additional memory space needed to store the tools and the device information on the device. Although compressing the software (in a JAR- Java Archive) and the XML-documents (DIB+ID) to save memory on the device is uncritical because the device software does not need to access them (otherwise compression algorithms must be installed on the device) the memory space will be enormous. Therefore, this approach will be appropriate for devices, which have sufficient memory to store the whole information.

The consideration above neglects the fact that both, the embedded web-server and the interrogation server, requires port 80 as entry point, because both (web-service and SOAP) use HTTP over port 80 as common protocol. This conflict can be resolved by using a dispatcher module as single point of entry for port 80 access. As shown in the figure above the web-server is only used to download HTML pages, hence only the HTTP-GET method will be used. SOAP uses HTTP-POST, therefore the dispatcher could forward HTTP-GET requests to the web-service and POST request to the interrogation server. Another way to resolve the conflict will be to use standard CGI (Common Gateway Interface) which is commonly supported by all web-servers. The interrogation software on the device could be implemented as CGI process, which is then invoked by the web-server. This approach has the drawback that CGI processes are only used for a single HTTP request/reply pair, but for example diagnosis requires a persistence channel, where a request might be responded by multiple replies. So this scheme will not work out with standard web-servers.

5.5.2 Remote service and maintenance server

The last chapter has shown a flexible and integrated approach, which requires additional memory space on the device for tools and device information. This chapter introduces an approach which is also suitable for smaller devices, because an additional server is provided to store the tools and the device information. The server maintains a database of generic tools (may be three or four) and the DIBs+IDs of all devices and device versions. In contrast to the last chapter, where the device information database is partitioned and stored with the device, this approach represents a centralised database concept. The server must contain all DIBs+IDs for every device and device version sold by the vendor company and is a critical point of failure, because if it crashes, remote access on embedded devices will be not possible. The following figure illustrates the remote access process in this case.

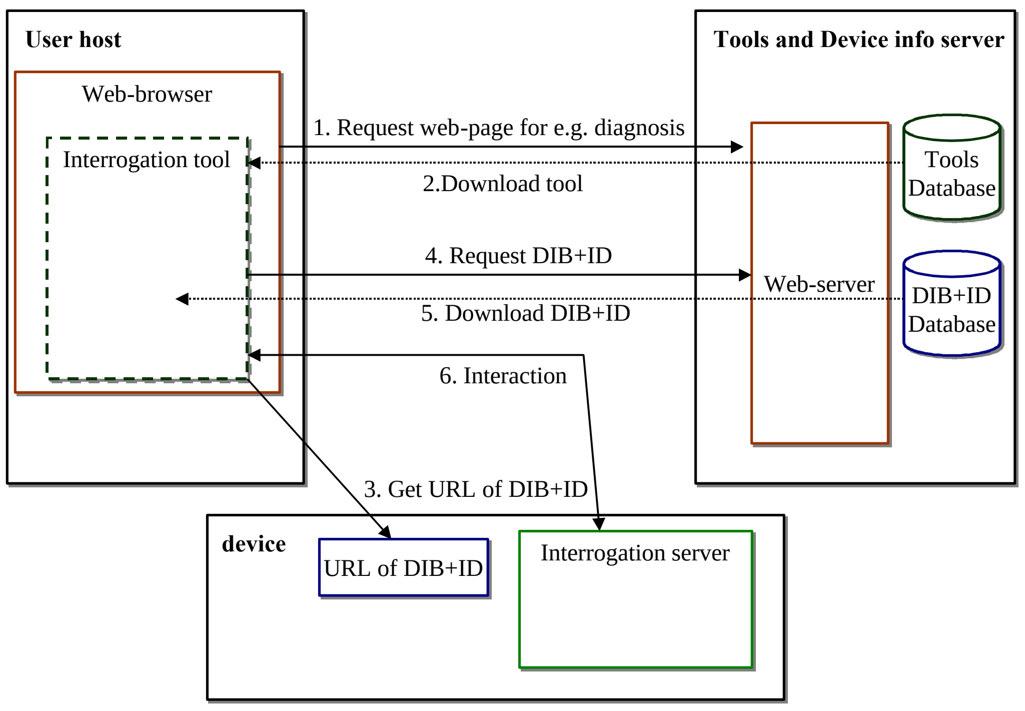


Figure 5-12 Tools and DIB server

The process is defined as follows:

1. The browser connects to the tool and device information server and requests the interrogation software (HTML and Applet). At this time the interrogation tool can not immediately acquire the DIB+ID, because it does not know the location for the specific device.

2. The vendor's tool and DIB+ID server supplies the software.

3. The user specifies the URL of the device to be serviced or maintained and the interrogation software requests the URL of the device's DIB+ID from the device. Therefore the URL must be stored with every device before it is delivered.

4. With the URL the interrogation tool can request the DIB+ID of the devices from the server.

5. The server returns the device information (XML).

6. The generic tool populates its user interface with the information in the DIB. The user sends commands to the device and observes e.g. data and state values.

There are three problems with that approach:

- If applets are used as interrogation software, but the sandbox [FLANNAGAN01] principle will not allow the applet to connect to other servers than the server from where the applet is loaded. This problem can be solved by signing (authenticating) the applet and manually editing the policy file of the browser. The applet becomes trusted and can connect to the device to acquire the DIB+ID URL as described in step 4.
- The tool and DIB server must be always available, otherwise remote access on devices is not possible. To achieve high availability, the service could be replicated. Very important to this consideration here is that changes to the server (adding device information for new device version) must be performed in a way that the overall service is still available.
- The approach inherently requires access to the internet, because tools and DIBs+IDs are downloaded from the server from the internet.

The first problem is not critical because Java provides very simple mechanisms to create authenticated applets. This requires some manual editing of the policy file, which makes it less convenient to the user, but this process might be automated by providing additional software.

The second problem requires management of redundancy which is more critical. A second hot-standby server could installed to provide a high degree of availability, but managing replicated services is a expensive task. Besides, the management of the DIBs+IDs in such a database is very critical, because every device contains a reference to its own DIB+ID on the server. Changes to the structure of the device information database must be performed carefully to avoid that references stored in devices all over the world become invalid. But on the other hand, if changes are made carefully, there is only a single point, where changes have to be performed. A new tool version can be updated very easily on the central server, whereas if tools are stored with the device, every device has to be updated with the new tool. DIBs+Ids are specific to each device, hence a new DIB+ID version is associated with a new device version. If a new device version (software version) is created, it must be uploaded to every device, wherefore a DIB and interface definition could be updated as well. So storing device information centrally will not bring more advantage than storing memory space on the device.

The third problem restricts the possible topologies introduced in 5.1 to those which provide internet access. This is a major drawback because reasonable alternatives like direct modem access are not possible with this approach.

5.5.3 Mixed approach

The download of tools is less critical because changes to generic tools are not very frequent (otherwise they are not generic). Therefore the software (HTML page with embedded Applet) can be downloaded and stored once locally and started from the local disc. This has the advantage that the applet has all rights to connect to the device to acquire the DIB. The server introduced in the last chapter is only used here to provide the tools, either to download on local disc and start them locally, or by loading directly into the local web-browser. In this approach the device information can remain on the device. This has several advantages: A compressed DIB is not very large (10-50Kbyte) but sophisticated tools might need more than 100Kbyte of flash memory space. Therefore, this approach will reduce the memory space requirement compared the approach in chapter 5.5.1. Moreover the availability requirement to the service and maintenance server is defused, because only the tools must be downloaded from it. Local copies (replicas) of tools can be used if the server is not available. Therefore access to the internet is not necessarily required. This scheme supports all topologies suggested in 5.1, even those, which does not support internet access.

5.5.4 Summary

The decision which approach is chosen primarily depends on the free memory space on the device. The approach with the tool and device information server introduced in chapter 5.5.2 is only practicable with topologies which provide internet access The mixed approach combines the advantages of the preceding ones, by storing the tools on a distinct server and storing the DIBs+IDs of devices with the device. In the following we will use the mixed approach but future projects might prefer one of the other alternatives.

5.6 Risk analysis

Remote interrogation of embedded systems, using sophisticated protocols and low quality networks major challenge. Hence, there are potential risks in this project and – although an careful analysis supported by prototypes were carried out – these risks can not be fully excluded. During the analysis here, following risks has been identified and partly mitigated.

Risk	Risk Type	Description
It will be not possible	Technology	Remote access to internal resources (embedded

to access the device.	Business	devices) will not be accepted by the customer.
Device performance will be insufficient	Technology	For real-time interrogation the device performance must be sufficient to produce, process and transmit data at certain rates, especially if serialisation in text is used.
QOS of underlying networks is insufficient	Technology	Low bandwidth and high delays will decrease the capabilities of remote interrogation.
Memory on device will be insufficient to store XML-data.	Technology	Generic tools require that the interfaces and the data objects on a device are described somehow. A description in XML will require large amount of memory.

Table 5-2 Identification of possible risks and their category

Additional to the risks described above, there are typical risks in software projects described in [SOMMER01] or [PRESSMAN01] concerning time and cost of projects, which are not discussed here in context of this dissertation. The risks has been analysed and their probability and effects evaluated. The following table distinguishes between diagnosis type and configuration type interrogation, because requirements to network and device performance is particularly set by the first, the latter will also work in worse conditions.

Risk	**Diagnosis type**		**Configuration type**	
	Probability	**Effects**	**Probability**	**Effects**
It will be not possible to access the device.	High	Catastrophic	High	Catastrophic
Device performance will be insufficient	Moderate	Serious	Low	Serious
QOS of underlying networks is insufficient	High	Serious	Low	Tolerable
Memory on device will be insufficient to store XML-data.	Moderate	Serious	Moderate	Serious

Table 5-3 Identification of possible risks and their category

In the preceding chapter, some of these risks has been mitigated by either careful analysis or prototypes. The following chapter summarises how risk – especially technical risks – are addressed in this project.

Risk	**Mitigation**
It will be not possible to access the device.	Chapter 5.1 provided alternatives to the basic remote access scenario. Direct modem access is a reasonable alternative if security is an issue.
Device performance will be insufficient.	Chapter 5.3 introduced initial prototypes, where the performance of the devices has been evaluated.
QOS of underlying networks is insufficient	Chapter 5.3 has pointed out that some QOS is required to enable reasonable remote interrogation. Chapter 6.2.5 will evaluate compression, buffering and other related techniques, as enhancement for protocols and communication.

Memory on device will be insufficient to store XML-data.	Chapter 5.4 illustrated design issues for generic tools, and pointed out the need for additional information stored on the device. Chapter 6.3.3 will consider compression as technique to save storage space.

Table 5-4 Risk mitigation

Although, the analysis here has shown that the intended solution is practicable in principle, the expectations must be lessoned upfront:

- Remote interrogation of embedded systems will be slower.
- It will be very difficult to design comfortable and sophisticated user interfaces for generic tools.
- The emphasis of this approach is simplicity, some of the interrogation activities must be performed on-site. Therefore remote interrogation with generic tools will not replace existing methods, but it will coexist, in order to provide best flexibility.

6 Architecture

The introduced architecture here should provide a domain-specific and generic model for remote interrogation. A generic model, as defined in [SOMMER01], means that the architectural model is an abstraction of several real-world systems. A generic model represents a common architectural model which can be re-used when developing new systems. Generic models are usually derived bottom-up from existing systems. There is no existing remote interrogation system for embedded devices with generic tools, but the architectural model in this case is derived from the lack of flexibility and usability of existing systems.

6.1 Overview of architecture and major components

This chapter is intended to give an overview of the architecture. This view will be refined in chapter 6.3 where the components will be decomposed to sub-components and their functionality described in more detail. Here, I will give brief description of the major components involved in remote interrogation. The next figure shows the major system components, which are derived during the analysis. The dark rectangles in the figure represent components which must be additionally installed on the device.

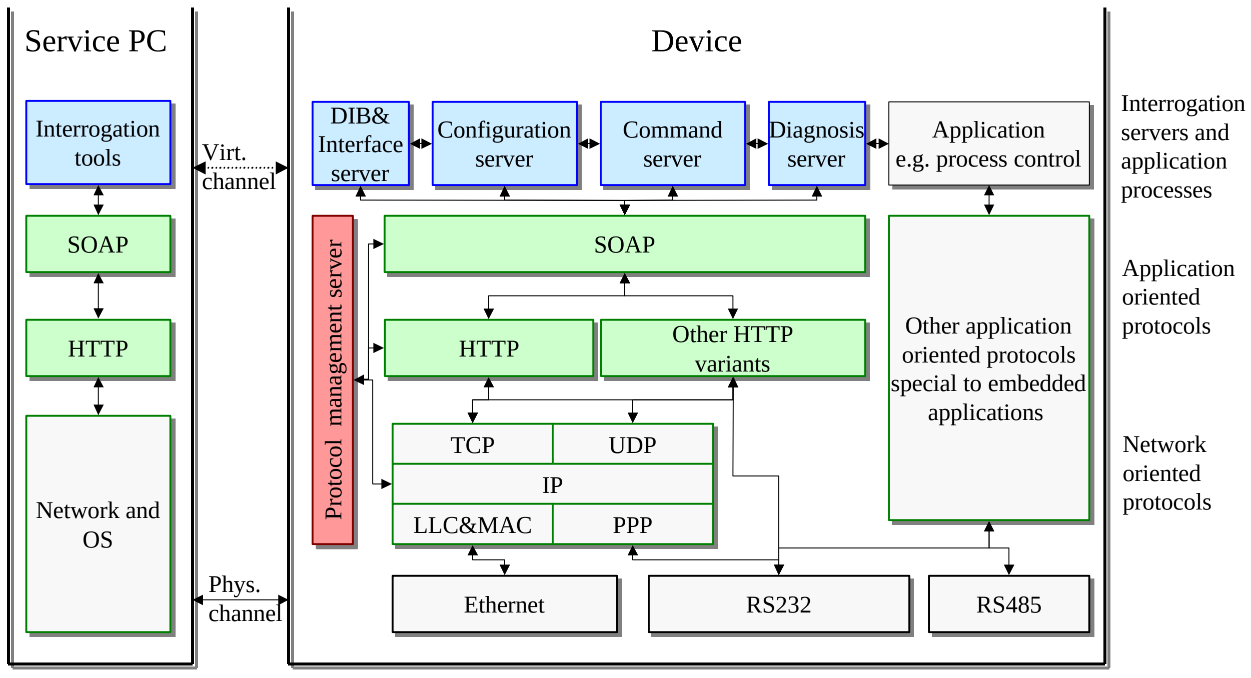

Service PC
Device
Interrogation tools
SOAP
HTTP
Network and OS
Virt. channel
Phys. channel
DIB& Interface server
Configuration server
Command server
Diagnosis server
Application e.g. process control
Protocol management server
SOAP
HTTP
Other HTTP variants
TCP
UDP
IP
LLC&MAC
PPP
Other application oriented protocols special to embedded applications
Ethernet
RS232
RS485
Interrogation servers and application processes
Application oriented protocols
Network oriented protocols

Figure 6-1 Overview of the architecture

The generic tools on the service PC communicates with its peers (servers) on the device. The servers have well-known, common interfaces, which are described with aid of an IDL. The servers provide remote access on devices data objects (parameters, states and process data) residing on the device over common and basic operations. SOAP and HTTP are used as common protocol between device and the interrogation tools. The functionality of the components will be described here briefly:

Interrogation tools: The interrogation tools are generic interrogation software, which can visualised in a standard browser. The interrogation tools will be introduced here for diagnosis, configuration and control. The interrogation tools invoke remote operations on the servers, residing on the device.

Interrogation servers: The interrogation servers represent the peers of the generic tools. Each interrogation tool has a corresponding peer server on the device. The figure shows three interrogation servers: the configuration server provides a common interface to configurable parameter objects; the diagnosis server provides a common interface to diagnosticable process data or state objects; the command server provides a common interface to the device application software (e.g. move robot arms, reset device etc.). The interfaces of the interrogation servers are described with aid of an IDL, the data objects which they manage are described in the DIB. Note, that the servers affect each other. They communicate over shared objects or messages.

DIB&Interface server: The DIB&Interface server provides common interface for remote interrogation tool, which acquire the DIB and the interfaces of a device. The interface for this server is fixed, because it represents a common entry point to a device. The remote and generic interrogation tools and the device are low-coupled, because all other interfaces (to other servers) can be acquired and used dynamically. The one and only interface, which must be fixed and known to all interrogation software is the interface of the DIB&Interface server.

SOAP module: Both SOAP modules here – the one at the service PC and the one on the device – are self-written and not generic. Although, there is a SOAP serialisation library available from the Apache [APACHESOAP] project, which could be used for at least the Java interrogation software, we decided to implement our own special SOAP library, because the library is only in an experimental stage and very awkward to use. Hence, the SOAP modules do not fully implement the SOAP 1.1 specification, but only implements the parts required for this project. It is very recommended that the SOAP modules are replaced by generic and full implemented SOAP modules, as soon as they are available. There is only little prospective that SOAP modules for embedded devices written in ANSI-C will be available in the near future. Therefore the SOAP module implemented for the device in C will probably remain.

HTTP module: The figure above indicates that there are several HTTP modules, providing various possibilities to use HTTP. The basic HTTP module provides a simple request/reply protocol which is suitable for configuration or sending commands to a device. As chapter 6.2.4 will show, that diagnosis requires a persistent communication channel to be established. Therefore another HTTP module could additionally provide persistent channels. Another possibility is that HTTP messages are transmitted over SSL (Secure Socket Layer) to provide confidential communication. These additional features can be implemented in a single module, which can be configured to the specific needs, or several exchangeable modules could be used. As with the SOAP module, the HTTP module in this initial project is a self-implemented module, which only understands the methods POST and GET. The decision is taken because porting a full implementation of HTTP on a device requires more time than implementing a lean, special HTTP module. It is recommended that the self-implemented module is exchanged in future projects.

Protocol management module: This module allows the management of the protocol stack. For instance, the protocol stack might provide compression and security mechanisms at different layers (e.g. IPsec or SSL), which can enabled or disabled. It is intended to provide similar generic and remote interrogation tools also for protocol stack functionalities in future projects, but this project will neglect this issue.

Before the components are specified in more detail in chapter 6.3, the design of application oriented protocols will be examined in the next chapters.

6.2 Protocols and communication

The TCP/IP suite application protocols interact directly with the transport layer protocols (TCP and UDP). Each application protocol (e.g. FTP, HTTP etc.) has to define suitable data formats and data exchange protocols on top of TCP or UDP or upon the socket layer which provides a stream (TCP) or a message passing (UDP) abstraction. This has the advantage that these protocols can be tailored to the specific needs of the application. The protocol designer can choose the most basic functionality for this application and optimise these for performance or reliability, specific to the requirements of the application. This allows the definition of lean protocols with little protocol overhead. On the other hand, there is a basic set of functionality which has to be re-implemented each time and which is common to many applications. Therefore, the OSI reference model has addressed this point, by defining extra layers, which basically provide more flexibility, openness and enable better maintenance and reusability of components. In OSI, application layer protocols interact through the protocol entities associated with the intermediate presentation and session layers. These application support layers are relevant to the protocols developed here for interrogation of remote embedded system. The intention here is not to provide a full mapping between the protocols here and the OSI reference model, but, in order to support the discussion about the protocols, we will refer to the OSI protocol stack, to illustrate analogies in structuring and functionalities of components. The following paragraphs provide an overview of the protocol functionalities, which we are seeking for. These functionality typically resides at the session, presentation and application layers of the OSI stack:

Application layer: The application layer in OSI define common application service elements(CASE) and application specific service elements(SASE). Relevant to this discussion here are CASE services such as ACSE(application control service element) which is used to set up a logical channel between application processes, and ROSE(remote operation service element), which basically provides remote procedure call facility for OSI.

Presentation layer: The presentation layer is concerned with the representation (syntax) of the data. The aim of the layer is to ensure that messages exchanged between two application processes have a common meaning – known as shared semantics – to both processes. Although syntax conversion is the major role of the presentation layer, OSI considers also encryption and compression in this layer[HALSALL96]. OSI suggests to perform first the encryption and then the compression on data. We will see that this has several drawbacks and because we use TCP/IP networks, there are alternatives for at least encryption, which suites better to the requirements of this project.

Session layer: The session layer and the presentation layer in OSI are treated separately because of specification purposes, but they operate in close cooperation with the various application layer protocols to provide a particular application support function. The session layer primarily included to the OSI-stack to minimise effects of network failures. The basic functionality of the session layer is the synchronisation of activities and dialog units(data exchange) and to handle network errors in various ways, so that the sessions can be resumed after network failures. Additional to the half duplex and duplex data transmission the session layer in OSI provides a activity management and synchronisation through synchronisation points[HALSALL96].

The following chapters show the successive development of a protocol based on SOAP for interrogation of remote embedded systems. The protocol will be developed from an intuitive basis XML-protocol by adding fields and functionality to form a generic protocol, which considers all aspects of remote interrogation. Because SOAP only addresses common attributes and structure, but does not itself define any application semantics such as a programming model or implementation specific semantics, we will extend SOAP to the specific need of the application here. Basically, SOAP is used here as a mechanism for expressing application semantics by using the modular packaging model and encoding mechanisms for encoding data within modules. Hence, special protocol functionality will be defined to enhance the basic protocol specification of SOAP.

The design issues for defining the application oriented protocols here has been listed in chapter 5.3.2 The following chapters will discuss these points briefly if SOAP specification already addresses them and in more detail when extensions to SOAP must be defined.

6.2.1 An intuitive XML-Protocol

This section represents the road to SOAP, which also illustrates the history of protocols used in this and former projects. The configuration and diagnosis were formerly performed by using the RS232[4] serial interface and connecting the device directly to a PC. The data was transmitted by using a text-based protocol with pre-defined commands and data structures. Long before SOAP was discovered as a suitable protocol for this project, an XML-based protocol was designed to replace the preliminary protocol for RS232. The amazing point in this is that the defined protocol was very similar to SOAP. I call it intuitive XML-protocol because it represents an logical approach in defining an ASCII-text based protocol.

Basically, this protocol provides a very simple remote procedure call (RPC) mechanism by defining the most basic elements for this purpose. We can intuitively define information needed for a basic RPC:

- We need to specify the method, which has to be invoked.
- We need to specify the parameters required for the specific method.

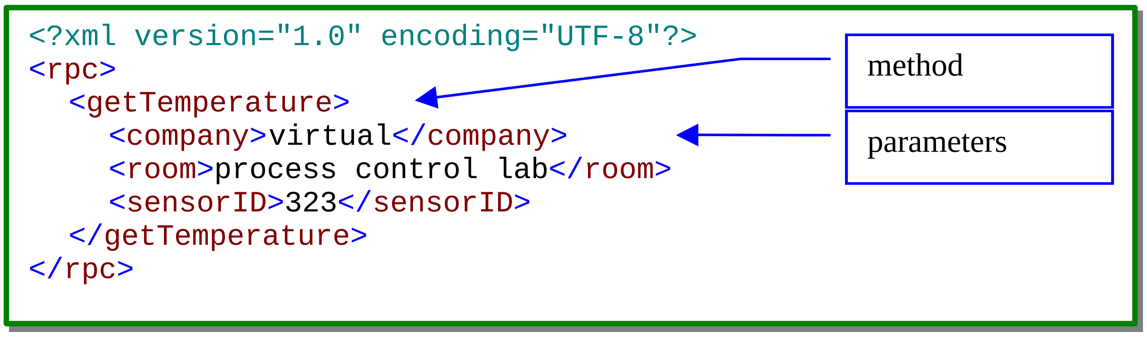
This can be simply expressed in XML by using specific tags for the information carried in the message. The following example shows a remote procedure call for the `getTemperature`

[4] The industry standard for transmission of data serial (one bit at a time) devices. The RS stands for Recommended Standard.

method of a HVAC device located in the process control lab of a virtual company. We assume here that the method requires the location of the sensor and the sensor ID as parameters. The procedure call is given here as pseudo-code. (the data types are not defined here):

```
Temperature = getTemperature(company, location, sensorID);
```

The next listing shows the corresponding XML request message.



```
<?xml version="1.0" encoding="UTF-8"?>
<rpc>
  <getTemperature>
    <company>virtual</company>
    <room>process control lab</room>
    <sensorID>323</sensorID>
  </getTemperature>
</rpc>
```

Listing 6-1 Intuitive protocol in XML for a basic RPC

The protocol defined here satisfies the most basic needs of a RPC. This is a intuitive basis for further enhancements discussed in the next chapters.

6.2.2 SOAP - The generic approach

Even though the preceding example is sufficient for a simple procedure call on a HVAC device, the implementation becomes more difficult to manage as the scope of the application increases for following reasons.

- The method name is not scoped to a particular (object's) interface. Hence, several interfaces on a host can not share operation names.
- Serialisation of complex data types is not defined.

The SOAP specification considers both points already. The first point is addressed by using namespaces to scope method name to interfaces. The second is addressed by the recommendation to use XML-Schemas to type data and define complex data types. Therefore these points will not further examined. Interested readers should refer to the SOAP specification or to [SCRIVNER00] who provides an excellent introduction to SOAP.

6.2.3 Extended information

In this chapter we define extended, application specific information to tailor the protocol to the requirements of remote access on embedded systems.

The defined functionality here is dependent on the needs of the application, for instance remote diagnosis will set other requirements to a protocols as remote configuration, but also on the underlying network protocols and how these mask and signal errors, which occur during data transmission.

If TCP is used as transport protocol to carry the SOAP payload (e.g. XML-data and HTTP-header) reliable communication can be assumed, hence there is no need to care about packet sizes, sequencing, segmentation, flow control and lost or corrupted packets. But as mentioned before the provided solution here should be widely independent of the underlying network oriented protocols, so that communication can not be assumed to be reliable. As an example, UDP can be used as transport layer protocol, or as shown in 5.1.3 direct modem access over telephony infrastructure can be considered. In both cases, our application could suffer from various failure modes in communication channels. So special care must be taken to detect and mask errors.

Providing fault tolerance is one aspect, where extended information is needed in the protocol. Furthermore, extended information can be attached to a SOAP packet regarding security, debugging, causality, or transactional processing. Some of these aspects will also be considered in the following chapters.

SOAP suggests to include application specific, extended information in the header of the SOAP message, as indicated in the next figure.

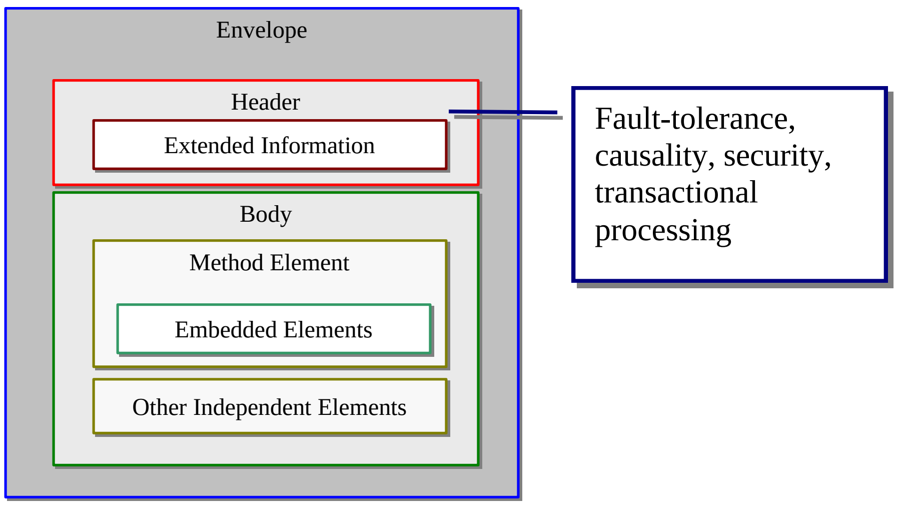

Envelope
Header
Extended Information
Body
Method Element
Embedded Elements
Other Independent Elements
Fault-tolerance, causality, security, transactional processing

Figure 6-2 SOAP message structure

In the following chapters we will identify useful information relevant to this project which can be included in the SOAP header.

6.2.3.1 Fault tolerance

Remote access to embedded systems is associated with a set of additional failure modes. Additional to failures in processes, there will be a set of communication failures introduced by the underlying networks. The following sections will identify failures and show how to mask them.

Omission failures: The faults classified as omission failures refer to cases when a process or communication channel fails to perform actions that it is supposed to do [CDK00], [BACON98]. The chief omission failure of a process is to crash. In this case the process will not respond to any request message. The communication channel produces an omission failure if it does not transport a message from the sender’s buffer to the receiver’s buffer [CDK00]. This is known as “dropping messages” and is generally caused by lack of buffer space at the receiver, or at a intervening gateway.

Arbitrary failures: The term arbitrary or Byzantine failure is used to describe the worst possible failure semantics, in which any type of error may occur. Communication channels can suffer from arbitrary failure, for example messages contents may be corrupted, non-existing messages could be delivered or real message could be delivered more than once or out of order.

The following chapters illustrate approaches that allow reliable communication between two processes. Some of the issues above will be addressed by including additional fields in the SOAP-message, others will be addressed by the design of the communication modules. To complete the discussion here about failures and reliable communication, we will define the term reliable communication and refer to this definition in later discussions.

Timeouts and retransmission

The basic scheme to mask omission failures are timeouts and retransmission. Appropriate timers could be set in the communication software modules which perform the HTTP protocol or other related protocols. After a timeout is detected the message is retransmitted, and if the maximum number of retransmissions are reached, the failure is indicated to the user. This only affects the design of the communication modules but not the SOAP message itself. The following figure shows two peer communication modules which carry out the request-reply protocol. The request message is lost, which causes the sender(requester) to retransmit the message after a timeout.

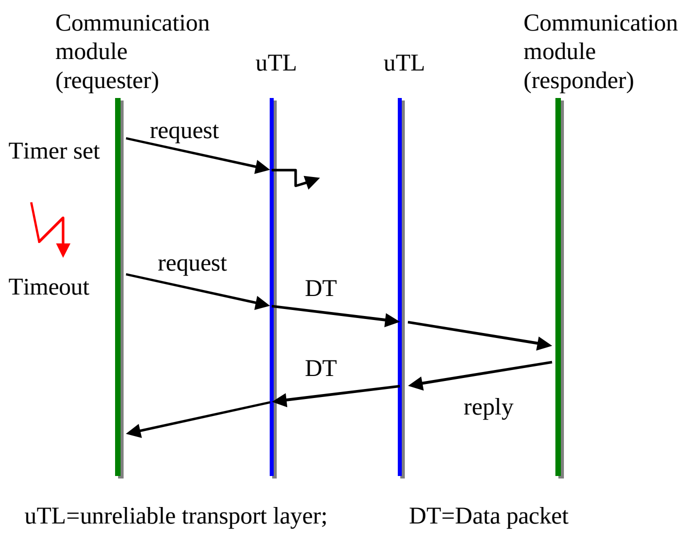

Communication module (requester)
uTL
uTL
Communication module (responder)
Timer set
request
Timeout
request
DT
DT
reply
uTL=unreliable transport layer;
DT=Data packet

Figure 6-3 Timeout and retransmission

It is not possible to find out the cause of the omission failure, e.g. it could be possible that the responder process has crashed or the reply message has been lost, or even the reply could only be late. In the two latter cases the second request message would cause that the method is invoked twice. If the device provides operations which are not idempotent then executing the method twice would lead to serious inconsistencies. The next chapter deals with these kind of problems.

Sequencing and time stamping

Numbering requests with a unique request ID, will enable detection of out of order messages as well as filtering duplicates. The first is important if a persistent connection is used to carry several request/reply pairs, the latter is important if the operations provided by a device are not idempotent. Sequencing can be achieved by including a request ID, lets say an integer value, into the SOAP-Header. Each time a request message is generated the request ID is uniformly incremented. The next listing illustrates a SOAP message with a request ID as extended information.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schmemas.xmlsoap.org/soap/envelope/"
SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header>
    <r:RequestNumber xmln:r="Request-URI">
      2
    </r:RequestNumber>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    …
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 6-2 SOAP request with request ID

The reply message will also include the same request ID, so that the requester can match replies to requests. The request number is scoped to the requester's namespace. The embedded systems on which the remote method is invoked can maintain a history table where it basically maintains the most recent request ID. If the request message's number is detected to be duplicate and the operation is not idempotend, the reply is sent a second time without executing the method twice. This requires also that the most recent result is stored in a table.

If a persistent channel over an unreliable transport layer is used for e.g. diagnosis purposes, the replies are sent continuously and other requests can be sent independent from the replies. In this conversation type communication the requests and replies are none blocking, unless the receive buffer of the communicating processes is empty or the send buffer is full. In such type of communications a request message can be responded with more than one reply messages, all having the same request ID. For the requester it is not possible to detect the order of the arriving replies unless reliable stream communication is used. In such a communication scheme, a reply ID is attached in addition to the request ID to detect out of order replies at the requester process. On the other hand diagnosis mostly involves the transmission of some real-time data, so that some time information should be included to a message. So instead of using a reply ID to mark multiple replies over time, timestamps could be attached to the SOAP reply message, which could give additional information for diagnosis or other purposes. Basically, only using timestamps is not sufficient because of two reasons: firstly, the clock resolution might be not high enough to mark every message with a unique timestamp, and secondly it is hard to detect unordered messages without the knowledge of the exact transmission times and the clock accuracy between the communicating processes. The first reason is self-explaining, because events on a processor can be generated very frequently (e.g. every 10ns) wherefore the clock resolution on the device might not be sufficient to timestamp every event. The second reason is because the requester process(e.g. diagnosis tool) can not know how long it has to hold back the message to ensure causality (that no earlier message can arrive). The following formula calculates the local time of the receiver (requester) process when it can be sure that no earlier reply message will arrive (message becomes stable):

$$\boxed{T_{local\_when\_message\_becomes\_stable} = T_{when\_message\_is\_received} + \Delta T_{receiver-sender} + T_{Transmission}}$$

Formula 6-1

Unfortunately, the only time which can be measured accurately is the receivers time when it receives the message. The transmission time is subject to variations especially for communication over the internet. The estimate on the clock skew between the two processes depends on the transmission time, therefore it will also be inaccurate. For these reasons it will be required to attach both the reply ID and the timestamp information, for this kind of communication where a request can be responded with multiple replies. The next reply message example is 7th reply message to the request numbered with 2.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schmemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle=
                  "http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header>
    <r:RequestID xmln:r="Request-URI">
      2
    </r:RequestID>
    <q:ReplyID xmlns:q="Reply-URI">
      7
    </q:ReplyID>
    <q:ReplyTimeStamp xmlns:q="Reply-URI">
      12:21:12:123
    </q:ReplyTimeStamp>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <get xmlns=" HVACHeatControllerInterface ">
      <result> 37.5</result>
    </get>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 6-3 Timestamps and reply number in multiple reply messages

Detecting arbitrary failures – Checksum

As shown in the last chapter, sequence number can be used to detect arbitrary failures like delivery non-existent messages or duplicated messages. In addition to that checksums can be attached to a message to detect corrupted messages. Basically, using a checksum converts an arbitrary failure of corrupted messages into a omission failure, because the message is dropped at the receiver. A simple single character checksum can be used, but special care has to be taken with special characters which are not allowed in XML.

Fault tolerance summary

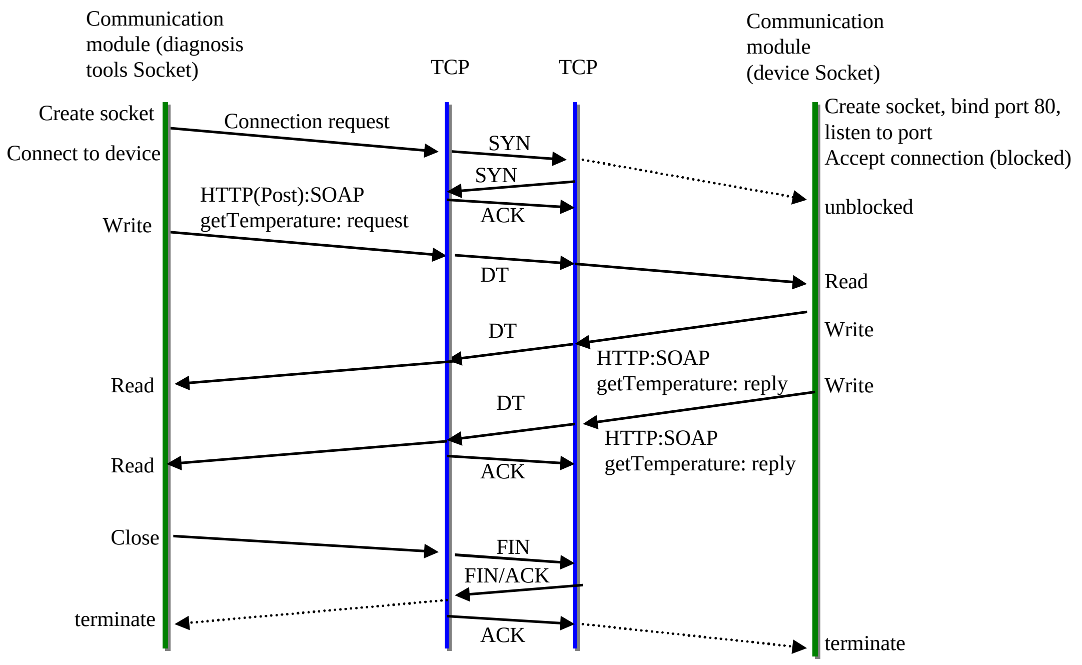
It is very obvious that a reliable transport protocol as TCP simplifies the considerations on failures in communication channels. Implementing fault tolerance above TCP streams would mean doubling the overhead, unnecessarily. Hence, the discussion above illustrated general principals to detect and recover failures, which can occur, if an unreliable transport service is used. Not all devices may afford to run a full TCP/IP stack implementation, because of memory and CPU-performance requirements. Therefore devices may be accessed by the using the simpler UDP implementation or point to point modem access with basic simple protocols.

6.2.4 HTTP and SOAP

Version 1.0 of the SOAP specification defined an XML-based protocol that can travel in HTTP requests and responses. Although version 1.1 of the SOAP specification placed less emphasis on HTTP, there are many advantages in using HTTP. Because the current internet infrastructure is heavily based on HTTP as a standardised method of communication, it only makes sense for SOAP to exploit this feature. HTTP facilitates real inter-operability because it is simple and is based on ASCII-text. An important point for this project is that most firewalls will only allow HTTP port 80 access. Therefore, if the internet infrastructure is used and devices are protected by a company firewall, HTTP is the only possibility to access these devices.

At the first glance, the simple request and reply model defined in HTTP seemed not to be appropriate especially for remote diagnosis, where a request needs to be responded with multiple replies over a certain time. Fortunately, the 1.1. version of the HTTP specification defines a Keep-Alive feature for persistent communication channels. This feature is defined in HTTP later on because re-establishing connections becomes extremely expensive, in terms of system resources, when clients attempt to request multiple resources in very short period of time. Besides, by opening and closing fewer TCP connections, CPU time is saved in routers and hosts (clients, servers, proxies, gateways, tunnels, or caches), and also memory used for TCP protocol control blocks can be saved in hosts. Moreover, allowing multiple request/reply pairs sent over a persistent channel will reduce the amount of messages and the latency between subsequent because connection establishment, management and release is needed only once.

Hence, it is possible to design a communication module which is able to support single and multiple request/reply messages, in order to support a wider range of applications. The following diagram shows a time-space diagram for the HVAC device example shown above, where a persistent communication channel is used over TCP to observe the room temperature over certain time.


Communication module (diagnosis tools Socket)
TCP
TCP
Communication module (device Socket)
Create socket
Connect to device
Write
Read
Read
Close
terminate
Connection request
SYN
SYN
ACK
HTTP(Post):SOAP getTemperature: request
DT
DT
DT
ACK
FIN
FIN/ACK
ACK
HTTP:SOAP getTemperature: reply
HTTP:SOAP getTemperature: reply
Create socket, bind port 80, listen to port
Accept connection (blocked)
unblocked
Read
Write
Write
terminate

SYN=Packets for connection establishment; ACK=Packets containing acknowledgements;
FIN =Packets for connection release; DT=Data packets

Figure 6-4 Diagnosis example over persistent HTTP communication channel with TCP

As we will see later in chapter 6.1 self-implemented HTTP modules will extend the HTTP communication model in various ways, e.g. allowing the device to keep states over multiple requests, or using UDP as transport layer protocol. Because SOAP 1.1 does not insist on standard HTTP as underlying protocol, this approach is feasible. The single most important feature of HTTP which will be kept is the Header information and the message formats.

6.2.5 Enhancing communication efficiency

6.2.5.1 Compression

Although, the OSI-model defines syntax conversion as the major role of the presentation layer, since it performs operations on all data prior to transfer and on all data that is received, it suggest that it is also the best place to perform such functions as data encryption and data compression [HALSALL96]. It suggests that the source presentation entity (layer), after encoding the data in each message from its local syntax into the corresponding transfer syntax, first encrypts the data – using previously negotiated and agreed encryption algorithm/key – and then compresses the encrypted data using a appropriate (agreed) algorithm.

Generally, this scheme can be adopted here with little modifications. Firstly, it is more beneficial to perform the compression prior the encryption, because compression will remove redundancies in the data. An attacker which seeks for these redundancies will have harder time to find them in the compressed and encrypted data. Secondly, the TCP/IP networks provide standard security mechanisms as SSL and IPsec which work underneath the application oriented protocols. Hence, encryption will not be considered here but in a parallel work [OEZASLAN01].

Before going into details how compression can reduce communication cost, we first have to examine the topologies where compression is sensible. The packet sizes for configuration and diagnosis, using SOAP messages will vary between 0.5-10 Kbytes. Compression will reduce these packet sizes and therefore will reduce transmission times which is basically given by packet size divided by bandwidth. The aim is to reduce total latency which include also various overheads for sender, receiver and routing. In the internet the delays introduced by routing and the overheads at the sender and receiver for TCP/IP protocol implementation will be significant and largely independent of the message sizes. Hence, the transmission time for message sizes as given here are insignificant, so that compression using internet infrastructure is not reasonable. But if we consider direct modem access, where sender and receiver overhead is low and routing delays are absent the transmission time will become significant and hence compression is a reasonable mechanism to reduce communication cost. Therefore, the discussion here must be seen in the context of direct modem access, which simplifies the considerations.

There is great potential in compressing XML based data, because it observes very high regularity. Initial tests performed on regular XML-documents with dictionary based compression methods like *gzip* or *compress* have shown that the compression ratio achieved is significant. The next figure shows measurements of compression ratio for regular XML-documents of different sizes compressed with the free available compression tools *gzip* (LZ77) and *compress* (LZC) [SALOMON00]. To make the measurements more realistic for embedded systems the buffer (LZ77) and the dictionary (LZC) size are reduced to 4Kbyte.

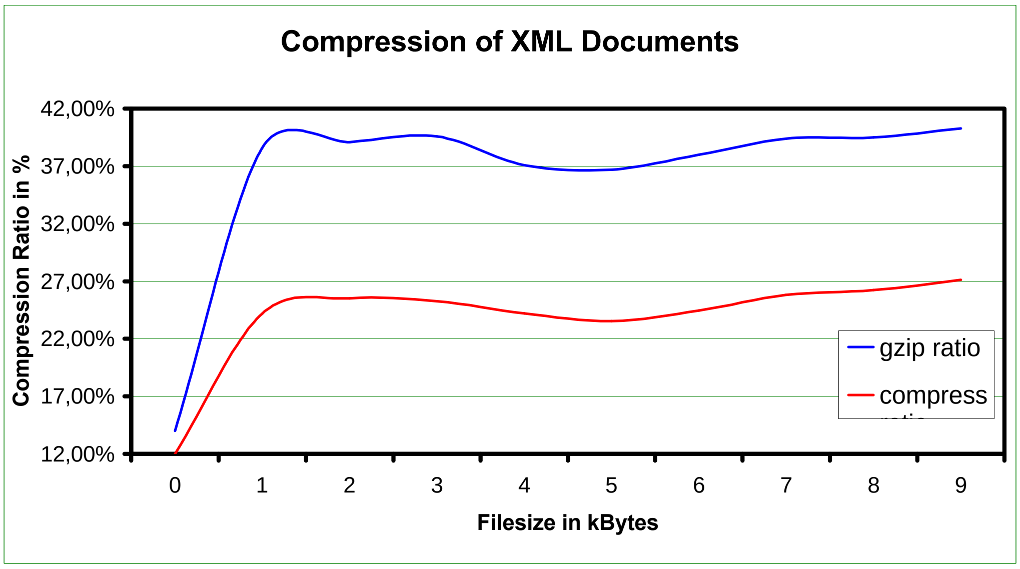


Figure 6-5 Compression ratio of ordinary XML-files

The measurements above were performed for 10 different XML-files and the average has been taken for the diagram. The diagram shows that compression ratio is better for large files, hence compression will give performance enhancement for SOAP messages larger than 1KB.

On the other hand compression will need extra resources on the embedded system, hence bandwidth is saved, but the overall transport latency might be increased, because the compression algorithm needs to much time to perform. The interesting point will be to figure out the bandwidth and the file sizes, where compression is reasonable. The total latency of a transmitted packet is the sum of sender and receiver overhead, the time of flight, and the transmission time, as shown in the next figure.

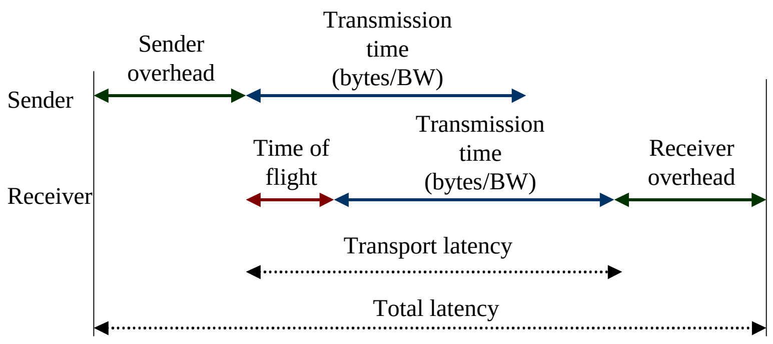


Figure 6-6 Total latency

Compression as used here reduces the transmission time but increases sender and receiver overhead. The sender and receiver overhead will be increased with a constant time, dependent on the computational complexity of the compression algorithm and the systems performance, whereas the transmission time will be reduced in dependence on the available bandwidth. To find the threshold bandwidth, where compression reduces overall latency, first measurements have to be taken for compression speed of the algorithms. Because the compression software discussed here is not available for the target embedded systems a reasonable alternative must be found. To make the speed measurements more realistic, they were carried out on a old UNIX(Solaris) workstation performing less than 80 MIPS. Therefore, the time needed for compression on the target embedded system will at least twice as long as the values given here. Nevertheless, they give a reasonable basis to find out if compression techniques are suitable for the given applications or not.

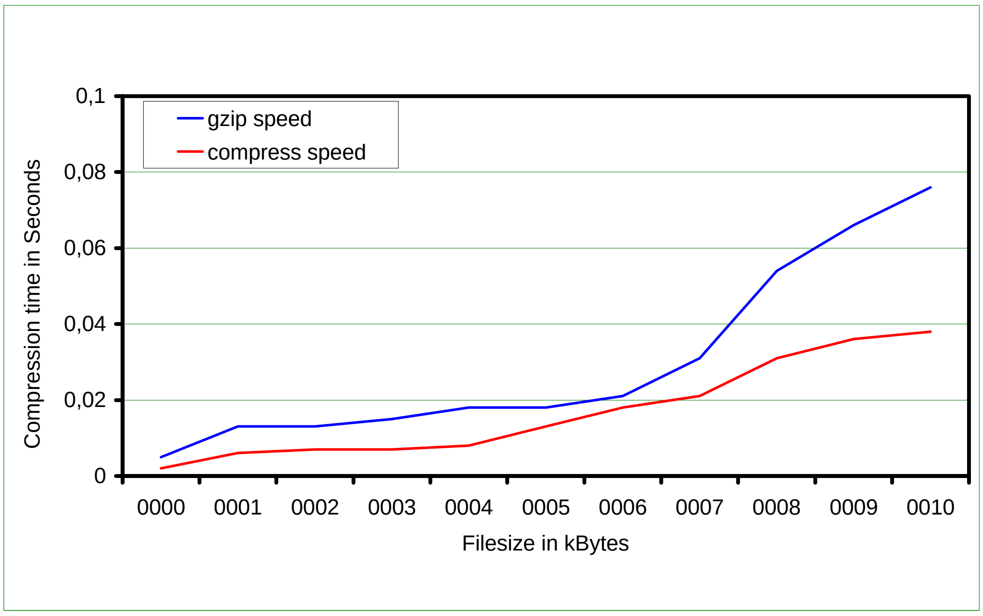


Figure 6-7 Compression time for different file sizes

To compare the overhead introduced by compression and the benefit of smaller packets we make the following assumptions:

- The overall packet size is reduced by 30%. This is an optimistic assumption because the compression affect only the SOAP messages itself, but the overheads of lower layer protocols remain.
- We multiply the time for compression with a factor three to take in account, that the test machine has higher performance than our target embedded system.
- GZIP(LZ77) is used for evaluation, providing better compression ratios for small messages compared to compress(LZC).
- We consider only sender overhead at the embedded system, because compared to this time, the receiver overhead (e.g. diagnosis tool) will be insignificant, if we assume that a powerful computer is used to run the software.

The following figure relates the overhead introduced by compression to the reduction of transmission time, using these assumptions.

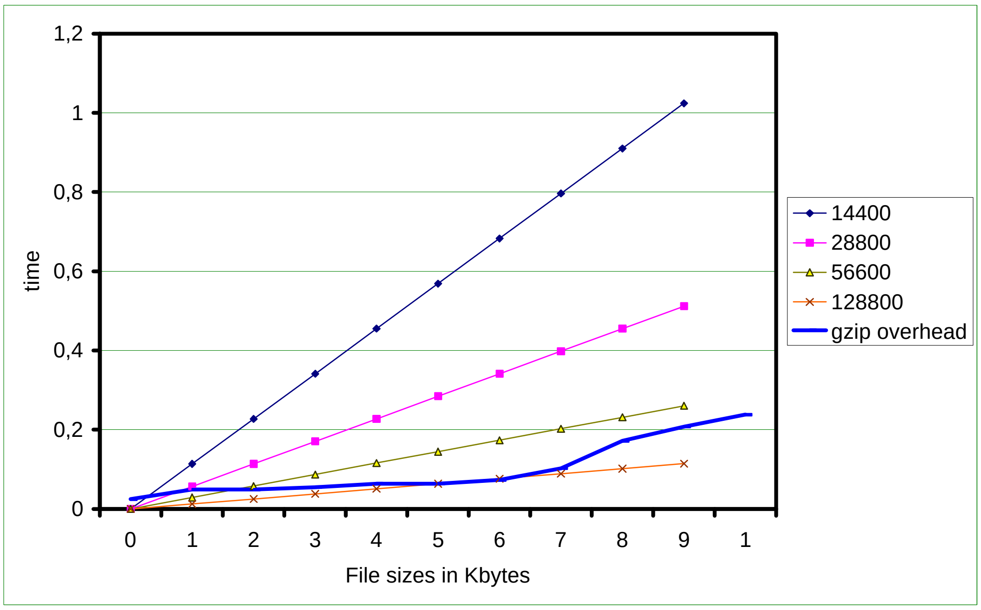


Figure 6-8 Evaluation of compression methods for XML based protocols

At the first glance, the time needed for compression seems to be less than the gained speed up due to smaller packet sized for all bandwidth options except the 128,8Kbit/s. But for small packet sizes less than 2Kbytes there is no reduction in total latency, or even for high bandwidth the total latency will increase.

For diagnosis purposes SOAP-messages (replies) have typically less than 1Kbyte, because a single value or a set of values are transmitted in a message from the embedded device to the diagnosis tool. For this kind of messages there is no benefit in using compression techniques. Moreover, if we consider that the compression algorithm will require roughly 10Kbyte flash memory for the program, and at least 4 Kbytes RAM for the buffer or dictionary, compression before transmission of messages becomes even less attractive.

6.2.5.2 Buffering

In the last chapter, we examined compression as a technique to reduce the required bandwidth and to reduce the delay between the embedded system and the interrogation software. Here, a technique is introduced which adds delay between sender and receiver by buffering at the receiver, but is titled as a method for enhancement. This is partly true, because it is only suitable where delays are acceptable but it is more important to provide a continuous stream of a data or state value over a certain period of time. A service technician can observe the real time behaviour of a data value, but respecting the fact that the signal is delayed for may be more than several seconds. Buffering of messages is important if the internet is used to access the devices remotely, because even though there is enough bandwidth available for reasonable diagnosis, congestion at routers or other network resources can lead to significant variation in delays (Jitter). Buffering compensates variation in delays but requires that the overall delay becomes larger.

In the buffering scheme introduced here, the buffering occurs above the SOAP-layer in application specific service elements (protocol entities). For instance, the diagnosis tool could provide a buffering option to the user, who can choose a appropriate buffer size for the specific value, according to the rates it is transmitted and the quality of the communication link. A second alternative could be to dynamically calculate the optimal buffer size. The timestamp included in the SOAP messages plays a major role in buffering, because the diagnosis tool could represent the values over a real-time basis.

Buffering compensates temporary variation in delays due to congested network resources as routers. An important and obvious requirement for this scheme is that the bandwidth over time must be sufficient for the message sizes and data rates required for diagnosis. The following example simulation illustrates the capabilities of buffering in context of diagnosis, where the temperature values supplied by a HVAC device are buffered at the diagnosis tool. The temperature supplied by the device is given in a range of 0.0-99.9 requiring 3 bytes in ASCII for representation. The associated SOAP message is given in Listing 6-3, including timestamp, request ID and reply ID, making up a SOAP message size of 500 ASCII-character. The overhead is enormous if only a little amount of data is transmitted. The UDP/IP and MAC overhead requires another 46 bytes (8 bytes UDP-Header + 20 bytes IP-Header + 18 bytes MAC header and trailer). With that, the overall packet size is 546 bytes. We assume that the temperature value is required every 100ms to make a reasonable diagnosis of the controller. The required bandwidth for the described scenario here is:

$$\boxed{BW_{\min} = \frac{packet..size \times 8}{sample..time} = \frac{546 bytes...per...message \times 8}{100 m\sec...per...message} = 43.68 Kbit/\sec}$$

Formula 6-2

We assume that most of the time 56Kbit/sec is available along the path between diagnosis tool and HVAC device but that the bandwidth drops because of network congestion from time to time. The buffer size at the diagnosis tool is chosen to 50 messages. The simulation illustrates an extreme where the bandwidth on the path from the device to the diagnosis tool drops to zero. The simulation shows that buffering can survive the situation over 5 seconds until the message queue at the receiver is empty.

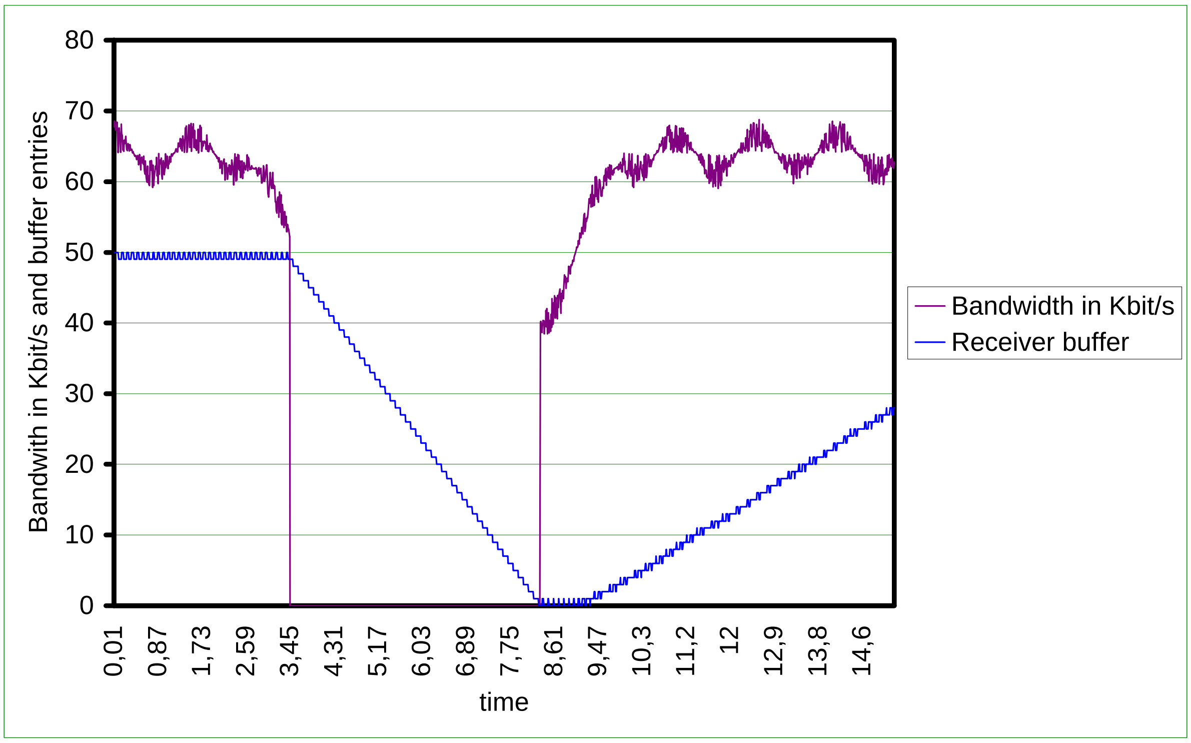


Figure 6-9 Simulation of the effect of buffering at the receiver

The diagnosis tool can show a continuous data stream of values and can survive large delays or variations of bandwidth over a time of 5 seconds, by buffering only 50 messages. On the other hand, the simulation depicts that effects of commands sent by the diagnosis tool can be observed after 5 seconds, so the overall delay is enormous. Hence, it must be decided for every special process parameter or state, whether a large delay is acceptable in order to be able to indicate the values continuously or whether it is more important that effects of commands (user interaction) for the device must be observed as soon as possible. Therefore, the intention here is to provide various modes of communication, where the user (service technician or customer) can decide due to the values she wants to observe which mode suits best.

The example above assumes buffering at the receiver (interrogation tool) but does not consider buffering at the sender (the device). Buffering at the sender (device) is critical because it requires additional RAM memory, and therefore extensive buffering might not be possible. But the example shows a worst case scenario where each temperature value (three ASCII) characters is packaged in SOAP message requiring 546bytes on the wire. An Ethernet frame can have 1500 bytes payload, making up 1472 bytes application data on top of the transport layer UDP, if we require that IP-fragmentation is to be avoided for efficient communication. Therefore, one scheme to reduce the protocol overhead per packet would be to package multiple replies within one single SOAP message to increase efficiency. In Listing 6-3 the reply message would contain more than a result. The difference of both approaches is depict in the next listing:

```
...
  <SOAP-ENV:Body>
    <get xmlns=
"HVACHeatControllerInterface">
      <result> 37.5</result>
    </get>
  </SOAP-ENV:Body>
```

```
...
  <SOAP-ENV:Body>
    <get xmlns=
"HVACHeatControllerInterface">
      <result1>37.5</result>
      <result2>37.3</result>
      <result3>37.3</result>
      <result4>37.1</result>
      <result5>37.1</result>
      <result6>37.8</result>
      ...
    </get>
  </SOAP-ENV:Body>
```

Listing 6-4 Multiple replies with a single SOAP message

In this example, one single SOAP message could carry 40 results making up packet of 1432 bytes on the wire, instead of 40x546=21,84Kbytes when one result is carried by one message. This is an enormous enhancement but associated with the same problems as identified for buffering at the receiver side: The results must be held back until the packet is filled, which produces additional delays. Besides, the timestamp information is lost because timestamping, as introduced here, is for a message but not for every single result. Hence, if timestamp information is required, it must be explicitly added to each result in the message, which introduces additional overheads. And again it depends on the intention of the user whether buffering is acceptable or not. Hence, the protocol provided must be customisable to various needs.

6.2.5.3 Other important enhancement methods for remote diagnosis

The methods introduced here are not relevant to SOAP or HTTP, but with the diagnosis application running on the embedded device, which transmits the acquired values to the diagnosis tool. The methods are concerned with saving bandwidth and reducing latency for diagnosis. The last chapter assumed a cyclic transmission of data at a certain rate(100ms). But this is not suitable for data and states, which are not produced at fixed rates.

Process control systems gather data from the controlled system, analyse it and initiate some action [BACON98]. This may involve simply taking a temperature or pressure reading at an appropriate time interval and checking against a save level, or more complex data gathering across the process followed by mathematical analysis and feedback to fine-tune the system. The data gathering and analysis are predictable and periodic, where the period depends on the process to be controlled. The cyclic transmission of this kind of data as shown in the last chapter is appropriate, but besides this periodic activity, there might be unpredictable events, for which this cyclic transmission of data is not suitable, wasting bandwidth unnecessarily.

Hence, an optimisation to save bandwidth will be to transmit the data or state information, when a new value is produced on the device. The timestamp included in the message can be used by the diagnosis tool, to represent the value appropriately, so that the real time behaviour of the value can be analysed. With this strategy the required bandwidth will be reduced significantly, for values, which are not produced periodically. For instance, a sensor in traffic controller system, which counts cars to optimise its traffic light control strategy, will produce a new car count value, only if a car is detected. In this case, it would be unsuitable to transmit the car count value periodically.

A second optimisation is, if we are not interested in the time the data is produced, but only in the value itself, then it could be possible to transmit the data or state value only if it is changed. As shown in the last example, the temperature of the room measured by the HVAC device is transmitted every 100 milliseconds to the diagnosis tool. Typically, the room temperature will not change dramatically every 100m (The sample rate is chosen to illustrate the network communication cost for a simple example). It might even possible, if the controller is doing its job well, that temperature will not change over many seconds or minutes. Therefore, to observe the temperature value reasonably, it is sufficient to transmit it only if it changes and not at the sample rate the device uses to control the temperature. If the value changes very infrequently, this scheme will save a huge amount of bandwidth.

6.2.6 Advanced transport protocols

Data transfer requirements for diagnosis type interrogation are very similar to those for audio/video streaming applications, except the fact the amount of data is far higher for audio and video streams. Lately, several new protocols have been defined to support the transfer of real-time video and standardised under RFC 1889, 1890, which could be also important for diagnosis type interrogation:

- Real-time Stream Transfer Protocol (RSTP):
  The application-level Real Time Streaming Protocol, RTSP, aims to provide a robust protocol for streaming multimedia in one-to-many applications over unicast and multicast, and to support interoperability between clients and servers from different vendors.
- Real-time Transport Protocol (RTP) and the Real-time Transport Control Protocol (RTCP):
  RTP is used to carry real-time multimedia traffic. RTCP is used in conjunction with RTP. RTCP provides additional services not catered by RTP, like transmission of out-of-band control data.
- Resource ReSerVation Protocol (RSVP)
  The Reservation Protocol (RSVP) is a resource reservation setup protocol designed for an integrated services internetwork. An application invokes RSVP to request a specific end-to-end QoS for a data stream.

The next figure shows the resulting protocol architecture. The details of these standards are discussed in [COMER01].

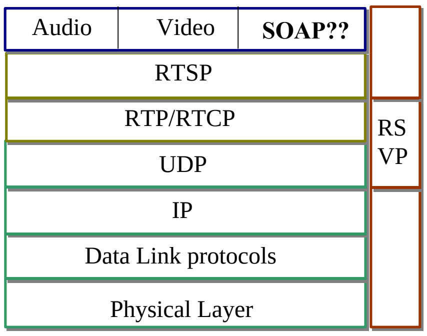


Figure 6-10 Protocol stack for RTP[COMER01]

Very important to the discussion here is the RSVP, which allows resources to be allocated at routers and other network devices, in order to achieve a guaranteed QoS between sender and receiver. Unfortunately, at the time this work takes place, most routers does not support RSVP, so that it will not considered further. But to allow reasonable diagnosis, future project must consider such mechanisms, which allow allocation of resources in network devices.

6.2.7 Summary

A solution for protocols for remote interrogation of embedded devices must support a wide range of applications, which have different requirements on communication and must be independent of the underlying network oriented layers. The preceding chapters introduced concepts and techniques for enhancing reliability and efficiency of communication. Because this work considers remote interrogation of embedded systems in general, there is no single solution which covers all possible applications and communication links. The aim here is rather to provide various options which can be chosen dynamically (by negotiation between device and interrogation tool) or manually (by the user) in order to enhance the communication for each specific case.

6.3 Refinement of architecture

The overview given in chapter 6.1 will now be refined by considering aspects of protocols and communication in the last chapters and by analysing the needs of generic interrogation tools in more detail. The focus of this chapter are the interfaces of the major components and their internal structure. Additional components will be defined which provide transparency and flexibility.

6.3.1 The communication software

6.3.1.1 The SOAP module

The SOAP modules – the one on the device and the one for the interrogation tool – are very similar in structure and functionality. The implementation is slightly different, because the one for the interrogation PC is written in Java, and therefore can use more enhanced programming mechanisms. In the following, the components for communication introduced chapter 6.1 are discussed in more detail. Besides, implementation specific aspects will be shown to illustrate the approach taken in the reference implementation here.

SOAP module on device

SOAP module on the device consists of platform specific part and a general part. The general part basically consists of a simple XML-SAX (Simple API for XML)Parser [MEANS00], which allows parsing of incoming request messages and a SOAP message assembler which marshals outgoing replies. The platform specific part provides a handler functionality which is invoked by the XML-parser, after the SAX parser convention (start of document, start of tag, end of tag etc.). The handler interprets the tags and the content of tags and builds pre-defined C-Structures. Because several interrogation servers use a single SOAP module, multiplexing and de-multiplexing functionality is required. This is addressed by introducing an additional dispatcher module which accepts messages from the SOAP module by exchanging system specific C-Structures and dispatches them to the corresponding interrogation server.

The dispatcher here assumes mailboxes provided by the operating system and maintains a table of mappings between the interface name (specified by the interface-uri namespace in the SOAP message) of the interrogation server and the mailbox which defines the communication entry point of the interrogation servers. The implementation emphasis is on a platform-independent approach, hence the system specific calls are reduced to a minimum and centralised in few modules. Hence, to port the software to other devices (with operating systems which e.g. not support mailboxes) only the system specific modules or functions have to replaced (see 8.2 for further discussion). The next figure shows the structure of the SOAP module and the interfaces to other components.

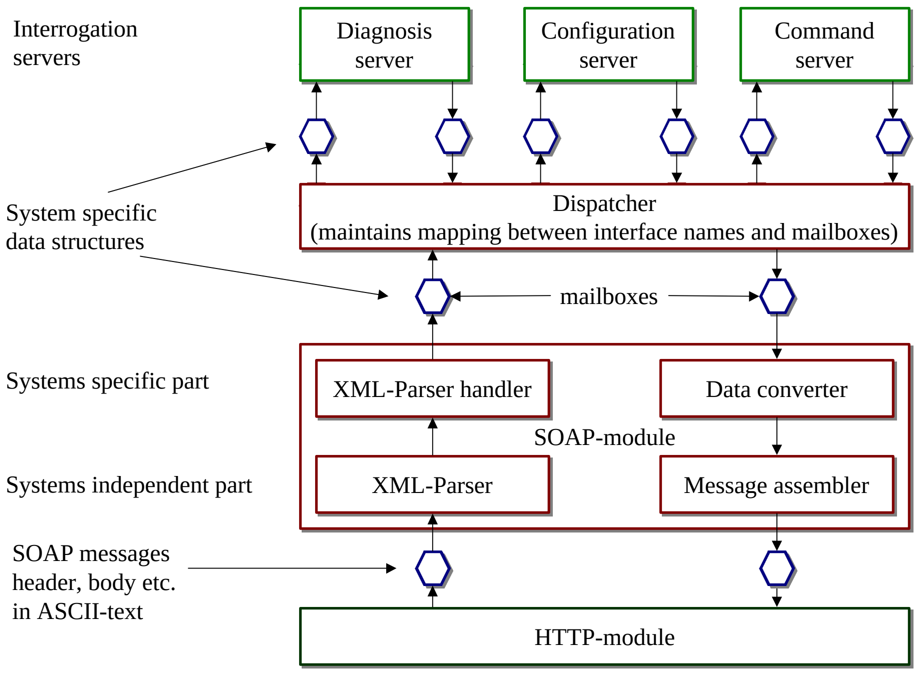


Figure 6-11 SOAP module and dispatcher

The XML-parser handler and the data converter, convert system specific data representation in a XML-data representation and vice versa. Because XML-schema is not used, the reference model here suggests application and platform specific conversion. The interesting point here is the mapping between SOAP messages into system specific message structures. Chapter 5.3 has shown that there are required and optional fields in the SOAP message, especially the header of a SOAP message is fully optional. In the following a pseudo code (similar to C) data representation of a SOAP message is shown, which allows to model optional fields by using unions as defined in C. The representation chosen here is specific to used programming language on the device. Later in this chapter a better alternative will be shown, which can be used for the SOAP module on the interrogation tool site. The following pseudo-code (very similar to C) listing shows a data representation of a SOAP message as defined in chapter 5.3. Unions are used for optional fields. A message type field is used to distinguish between various messages which differ in their optional fields:

```
Struct SOAPMessage{
        String interfaceURI
        String methodName;
        int messageType;
        Union extendedInfoHeader{
                Struct FaultTolerance{
                        Int requestID;
                        Int replyID;
                        String timestamp;
                }
                Struct FaultToleranceAndSecure{
                        Int requestID;
                        Int replyID;
                        String timestamp;
                        String hashValue;
                }
        …
        }
        String [2][] arguments;
}
```

Extended optional fileds

Arguments as key value pairs

Listing 6-5 Example representation of a SOAP message in pseudo code

The SOAP layer does not interpret any extended information but leaves the interpretation to the interrogation servers. Besides it has no knowledge about the method and the required arguments, hence it can not map the given arguments in text form into a system specific representation. It is possible that a generic SOAP-module builds the system specific data structures using XML-schema and dynamically downloads data definitions from a pre-defined location (as it is possible in Java) but these mechanisms are absent for the implementation on an embedded system. If the SOAP-module here was made responsible for converting system specific data structures in XML-data it would be required that every possible argument of any method (in each interrogation server) is known by the SOAP module. Therefore additional modules will be defined later with the interrogation servers which are responsible for the conversion of arguments.

The major design decision made here was that the SOAP-module is represented as self-contained task which communicates asynchronously over mailboxes. An alternative to this is to link the code of the SOAP module to the interrogation servers and communicate over pure synchronous method calls. This is faster but provides less modularity and flexibility. The reference implementation here is only an example and is specific to device platform in this project, but since the model should be generic, it is left to the developer how the components and their interactions are implemented.

SOAP module on the service PC

The SOAP implementation for the interrogation tool, which is preferably implemented in Java, can use more enhanced techniques. For instance, an inheritance tree can be used to represent various SOAP messages: The SOAP message object at the top of the inheritance tree defines the most basic and required fields (attributes). Each level in the tree defines more special and extended information. Because objects within a inheritance tree are polymorph, the most general object reference can be use to pass and object within the tree.

The second difference to the SOAP module defined for the device is that the SOAP module and the interrogation tools are within the same process – the Java Virtual Machine process. Hence, pure synchronous communication between SOAP module and interrogation software (tool) can be easily implemented, but also asynchronous communication is possible by using Threads and shared objects. Here, in this project the latter alternative is preferred because it allows that computation (and also user interaction) runs in parallel with the communication.

The HTTP module

The HTTP module is widely according to the specification 1.1, as it also defines persistence channels. The HTTP implementation here on the device and interrogation tool is only capable to create and interpret the HTTP-GET and HTTP-POST methods. The devices is provided with an extra dispatcher, which distinguishes between the GET and the POST method, by transferring GET-requests to the standard embedded web-server and POST-requests to the SOAP module. This is because the both the web-service implementation and the SOAP-module uses HTTP as common protocol, and a port (here 80) can be only associated with one process.

6.3.2 Interrogation servers

One requirement for interrogation with generic tools is that a peer interrogation server runs on the device. The interrogation software communicates with the corresponding server peer-to-peer. The tool-server pair can be viewed as distributed application (after the Client-Server model) which communicate over common protocols. A major design goal for every distributed application is that networking functionality should not reside at the application itself. Chapter 6.3.1.1 has shown that converting system specific data structures to text-based XML-data can not be done by the communication software (as defined here) because the SOAP module is not generic. To avoid that data conversion must be performed in the interrogation servers additional modules will be defined which are called – in analogy to RMI – stubs and skeletons.

The skeletons reside on the device and exist for every interrogation server. They receive the messages as defined in Listing 6-5 and convert the arguments which exist in key-value pairs (Strings) to a system specific representation, by considering the method (methodname) which is to be called. They invoke the method which is specified by the method name with the given arguments, convert the result in key-value string representation, package the result in a appropriate data (message) structure similar to that in Listing 6-5 and send the message to the dispatcher. For this purpose, the skeletons maintain a table of mappings between argument/result names/keys (strings which specify the path in the DIB) and the data structures (addresses) on the device. The dispatcher in this case simple transfers the message representation to the SOAP module, which converts it to a SOAP message. The following figure exemplifies this approach for the configuration server:

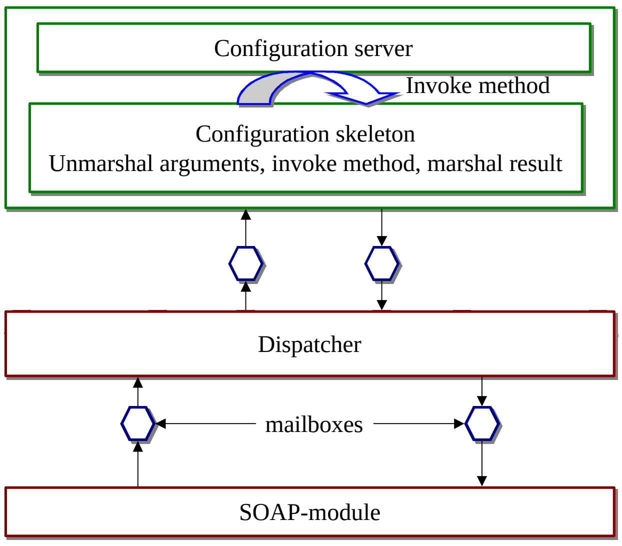


Figure 6-12 Marshalling and unmarshalling with skeleton

The stub residing on the service PC is very important to the goal that (generic) interrogation tools are independent of the device. The stub as defined here assembles the SOAP messages dynamically, by using the information from the DIB of a device and the interface definition of the servers residing on the device. This means that the stub adapts to the device (skeleton) dynamically). The support of generic tools and the role of the stub will be discussed in the next chapter and in chapter 7, where the implementation of a adaptive stub is outlined.

The major problem here in this approach where the skeleton is not generic (means that it can not produce system specific data types from tagged XML) is that the skeleton must be adapted each time by hand. Compared to CORBA where a similar scheme is used with IDL, stubs and skeletons, the missing link here is a IDL-compiler, which converts the definition of data types and operations to the corresponding target language. A similar approach is taken here by defining XSL (eXtensible Stylesheet Language) script which can generate C-Code. A XSL-stylsheet can be used to define a conversion from a XML based document into an other type (HTML, PDF etc.) documents. Here the target syntax is C-code. Once the DIB and the interfaces are defined (both in XML), two separate XSL-scripts can be used to generate .h files and .c files. The .h file defines type definition in C derived from the DIB and the .c file defines operations which are derived from the XML interface definition. A full implemented IDL compiler could also create the implementation of the skeleton function, in order to convert XML-data into system specific data, but in this initial project with tight time constraints the implementation of such an IDL-compiler is not possible. Hence, the conversion function are only defined and then implemented by hand. Future project must consider full automated skeleton generation, in order to increase maintainability and reduce development time.

6.3.3 Support for generic tools

There are three major requirements set by generic interrogation tools as defined here:

- The common interfaces for diagnosis, configuration and control must be described in such a way, that the interrogation tool can interpret them.
- The data objects which can be accessed from outside must be described in a device information base, which can acquired by a interrogation tool.
- The device must be able to understand SOAP messages. The message must be interpreted, the arguments extracted and the specified method invoked.
- Interrogation tools and servers must understand the data object types in a DIB. The data exchange is reduced to these pre-defined types, so they must be very general and cover (model) all possible data objects on a device.

If these requirements are fulfilled by the device, the interrogation tool and the communication software, then configuration, diagnosis and control is independent of the device. The next chapters describe the major innovations for generic tools – namely the DIB, the interface definition and the dynamic assembly of messages.

6.3.3.1 The device information base - DIB

This chapter describes an example definition of a DIB. Figure 5-9 has shown an example DIB representation in a tree. The information required to model and describe data objects is described here. The data objects are grouped in data object groups. The overall grouping represents the device information tree DIT. The nodes following the root node are named parameterprofile, processdataprofile and diagnosisprofile and specify the interface through which they are preferably accessible. The next listing depicts the grouping of objects.

```
<?xml version="1.0" encoding="UTF-8"?>
<DIB DIBversion="1.4" devicetype="HAVAC device"
deviceversion="1.4">
  <parameterprofile interfaceID="ConfigurationInterface">
    <parameterobjectgroup ID="HumunitySetting">
      ...
    </parameterobjectgroup >
    <parameterobjectgroup ID="TemperatureSetting">
      <parameterobject ID="ReferenceValue">
        ...
      </parameterobject>
        ...
      ...
    </parameterobjectgroup>
  </parameterprofile>
  <processdataprofile interfaceID="DiagnosisInterface">
    <dataobjectgroup ID="Humunity">
      ...
    </dataobjectgroup >
    <dataobjectgroup ID="Temperature">
      <dataobjectgroup ID="Sensor1">
        <dataobject ID="TemperatureValue">
          ...
        </dataobject>
      ...
    </dataobjectgroup >
  </processdataprofile>
  <stateprofile interfaceID="DiagnosisInterface">
    <stateobjectgroup ID="devicestates">
      ...
    </stateobjectgroup >
    <stateobjectgroup ID="processes">
        <stateobject ID="IOProcess">
          ...
        </stateobject>
      ...
    </stateobjectgroup>
  </stateprofile >
</DIB>
```

Listing 6-6 Grouping of objects in DIB

The object groups can contain several objects or further groups. Here, the DIB is divided into three root nodes, names with e.g. parameterprofile. Another structuring is of course possible as long as the interrogation tools and servers can interpret the grouping. There is a DTD defined for the DIB which allows validation of the structure of the DIB.

Important to the discussion here is the definition of the device data objects types, which must be understood by all components and especially by the interrogation tool which must represent them in a generic manner. The definition of the types of data objects is the most critical part here. The DIB can be changed in various ways for different device, each of them requiring another set of accessible objects. But if additional data object types are required then nearly all components must be modified, in order to support the additional type of objects. Hence, the types defined here must cover all possible data object types. Therefore, the most general and basic data types will be defined here. Especially, specific, compound types are totally avoided because they will differ from to device to device. The following data types are defined for parameters, (process) data and states.

Data type	**Description**
Integer	Defines a general integer type but does not specify how it is represented locally. It rather defines fields which define minimum, maximum, increment and default values. The interrogation tools might use these value to constrain user inputs.
Boolean	This type can be used for flags, bits or events. It has a default value which is specific to the device.
Selection	This type is appropriate for enumeration type variables. It specifies a set of selectable options. The options are specified in human readable form. This suites also very good for the representation of states, because state objects are also defined with a finite set of options.
Char	A single character could be also modelled by an integer with appropriate min and max values. But an explicit type is defined to allow the interrogation tool designer to represent characters different to normal integer objects.
String	Defines a string of characters with unknown or specified length.
Float	Defines floating point numbers. Most of the embedded system applications do not use floating point numbers, therefore this type is used very infrequently.

Table 6-1 Data object types for remote interrogation of embedded systems

The following listing illustrates example representations of data objects of float and selection type in XML. Other types are represented very similar.

```
<parameterobject ID="TemperatureUnit">
  <selection min="0" max="2" standard="0">
    <item value="0">Celsius</item>
    <item value="1">Fahrenheit</item>
  </selection>
</parameterobject>
...
<dataobject ID="TemperatureValue">
  <float max="100" min="0" precision="1" standard="20"/>
</dataobject>
```

Listing 6-7 Float and selection type

The DIB is static and does not contain the actual values. The actual values are dynamic and must be acquired explicitly. The example here also illustrates the limitations of the generic approach here and a static DIB: As shown in the listing above the float object has a min, max, precision and standard value. This values are fixed and defined by the device software developer upfront, therefore, a change in the temperature unit as provided above can not affect them. Hence, with the definitions here it is not possible to model associations of data objects. This limitation arise, because DTDs are used to define the DIB and the data objects within it. With more advanced approaches like XML-Schemas or dedicated XML-description languages, it is possible to model more complex constructs like associations or inheritance. Here, it is suggested to replace the DTD based definition by more appropriate techniques as soon as they are fully specified and validating parsers are available.

It is possible to include extended information for a data object concerning the representation on the interrogation tool's user interface. This extended information is also specified by the DIB's DTD, but enhanced techniques as PGML (Precise Graphical Mark-up Language) or UIML (User Interface Mark-up Language) could be used to specify the representation of the data objects in generic tools in future projects, if more sophisticated user interfaces are desired. Here, we only define a label, a brief description and a hint, which can be used to populate the user interface of the generic tools. Another extension to the DIB definition here would be to specify the access rights on the device data object level. This information can be used by a generic tool to control access for every specific data object. The next example listing illustrates extended information for objects and object groups as they are defined here.

```
<parameterobject ID="temperatureunit" readlevel="customer"
writelevel="administrator">
  <label>
    <english>temperature unit</english>
  </label>
  <hint>
    <english>Specify the unit of the temperature here</english>
  </hint>
  <briefdescription>
    <english>The temperature sensed by the HVAC device can be
provided in different units</english>
  </briefdescription>
  <selection max="1" min="0" standard="1">
    <item value="0">
      <english>Celsius</english>
    </item>
    <item value="1">
      <english>Fahrenheit</english>
    </item>
  </selection>
</parameterobject>
```

Listing 6-8 Extended information for data objects in DIB

The parameter object defined here requires at least a access rights of a customer for read access and administrator rights for write access. Here this information is included for demonstration purposes. Interested readers should refer to[OEZASLAN01], where access rights and security issues are discussed in detail. The other information defined here represent the minimum information required to visualise the object on a appropriate user interface. This issue will be further examined in chapter 7.3.2, where a generic tool is developed which extracts this information here to populate its user interface.

6.3.3.2 The interface definition

The interfaces of the services (interrogation servers) are described with interface definitions, which are used by a generic interrogation tools to invoke methods on the corresponding servers. The interface definition suggested here uses the object definitions of the DIB to specify arguments and result types. The diagnosis and configuration servers should provide the most general operations on data objects defined in the DIB, otherwise it is not possible to provide a generic approach. This is because only those data objects can be interpreted by generic tools. In the following, an example interface definition will be presented for the diagnosis server:

```
<component servername="diagnosisserver">
  <interfacename> diagnosisinterface</interfacename>
  <methodlist>
    <method methodID="get" comsmode="requestreply">
      <methoddescription>Get object</methoddescription>
      <arguments> <DIBobject/></arguments>
      <result><DIBobject/></result>
    </method>
    <method methodID="observe" comsmode="conversation">
      <methoddescription>Observe object</methoddescription>
      <arguments>
        <DIBobject/>
        <mode>
          <label>
            <english>Select mode</english>
          </label>
          <hint>
            <english>Specify the mode.</english>
          </hint>
          <briefdescription>
            <english>Specify the mode of observation.</english>
          </briefdescription>
          <selection max="2" min="0" standard="0">
            <item value="0">
              <english>Cyclic transmission</english>
            </item>
            <item value="1">
              <english>Transmit when produced</english>
            </item>
            <item value="2">
              <english>Transmit on change</english>
            </item>
          </selection>
        </mode>
        ...sample rate, other arguments
      </arguments>
      <result>
        <DIBobject/>
      </result>
    </method>
  </methodlist>
</component>
```

Listing 6-9 Interface definition of diagnosis server

The listing defines, that a diagnosis server on the device has an interface (might have more) which is specified by one or more methods. A method has arguments (zero or more) and may have a result. The interesting point here is that methods may have objects as arguments, which are not specified by the DIB. DIB objects are simply referred with a DIBobject tag to indicate that the method can accept objects defined by the DIB as arguments. The mode object specifies the mode of observation (selection) and is not contained in the DIB, because it does not represent a data object on the device. Hence, the interrogation tool can not know how to represent the mode object. Therefore it is explicitly specified in the interface. The important thing here is that the mode object is of a type (here selection) which is known by all and which has been already defined in the DIB, otherwise the object can not be interpreted.

The interface of the configuration server looks very similar, except that the observe method is replaced by a set method, because observing settings of a device over a time does not make sense. The command server interface is specific to every device, because it specifies the (remote) interface of the application running on the embedded system. For instance, the command interface for a HAVAC device could specify a reset, stand-by, activate etc. operations, which could be invoked remotely. The object required as arguments for those methods are typically not in the DIB, so that they must be explicitly specified in the interface. Chapter 7 will show further examples for commands.

7 Case study – A generic interrogation tool

This chapter exemplifies the development of generic and remote interrogation tools using the architectural framework defined in chapter 6. The major components identified there have been implemented prototypically. The implementation is very lean and supports only a basic set of functionalities. The time given for this initial project was too short to implement them fully but future projects will hopefully do it. The design and implementation of the components in chapter 6 will not discussed here explicitly, but important interfaces and functionalities will be illustrates when these are important to the design and implementation of generic tools.

The requirements on the components running on the device and the information which must be provided to a generic tool have been examined in chapter 6.3. The interrogation software here builds upon this architecture and assumes following functionality:

- Interrogation servers run on the device and accept remote method invocations
- SOAP and HTTP modules are implemented in Java and can be used by the interrogation tool
- DIB and interface definitions can be acquired from the DIB&Interface server on the device
- Device can be accessed remotely over the internet or intranet

7.1 Requirements for Interrogation tool software

Requirements for the generic interrogation tool are as follows:

Requirements on software

100 Interrogation software must allow diagnosis, configuration and control of remote devices.

101 Interrogation software must be independent of device

102 Installation of software on a service PC should not be required

103 The software must be run able on a standard browser

Requirements on user interface:

104 The user interface should allow intuitive interaction

105 The look and feel should be according to current web application GUI

7.2 Analysis

7.2.1 Use cases

Use case identify major services of the system which are provided to the system's user. The services provided by the device has been already identified in chapter 6.1. The services of the interrogation tool are closely related to them by the means that the service are presented to the user and service requests are forwarded to the device. The next diagram shows the use cases for the interrogation tool.

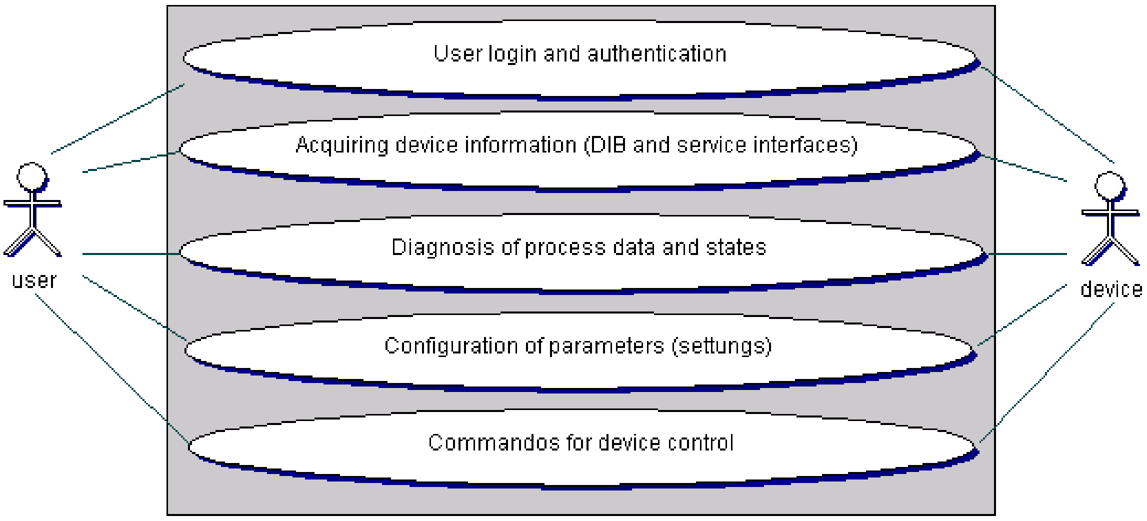

User login and authentication
Acquiring device information (DIB and service interfaces)
Diagnosis of process data and states
Configuration of parameters (settungs)
Commandos for device control
user
device

Figure 7-1 Use cases for interrogation tool

It follows a brief description of the use cases:

User login and authentication:

As mentioned before, this work here does not consider security issues. It is assumed here that perfectly secure access to devices is possible. Nevertheless, the initial prototype here provides a very simple login facility to illustrate how login and authentication can be deployed in the overall (user interface) design. Details about user login and administration will be ignored.

Acquiring device information (DIB and service interfaces):

The DIB and the service interfaces has identified as required information for generic tools in chapter 5.4 and the DIB&Interface service has been specified which provides uniform access to these information. The DIB&Interface service is invoked by the interrogation tool during the initialisation phase in order to populate the graphical user interface and to build a model of the device. The user can then choose services to manipulate data objects residing on the device.

Diagnosis of process data and states:

The user selects process data or state objects and invokes methods on diagnosis services of the device. The diagnosis service supplies the value either once or many times in a certain mode (cyclic, when changed, when produced) as specified in chapter 6.2.5.3. The requests and replies and packaged in SOAP(+HTTP) messages, where the messages are assembled dynamically according to the information in the DIB and interface definition.

Configuration of parameters (setting)

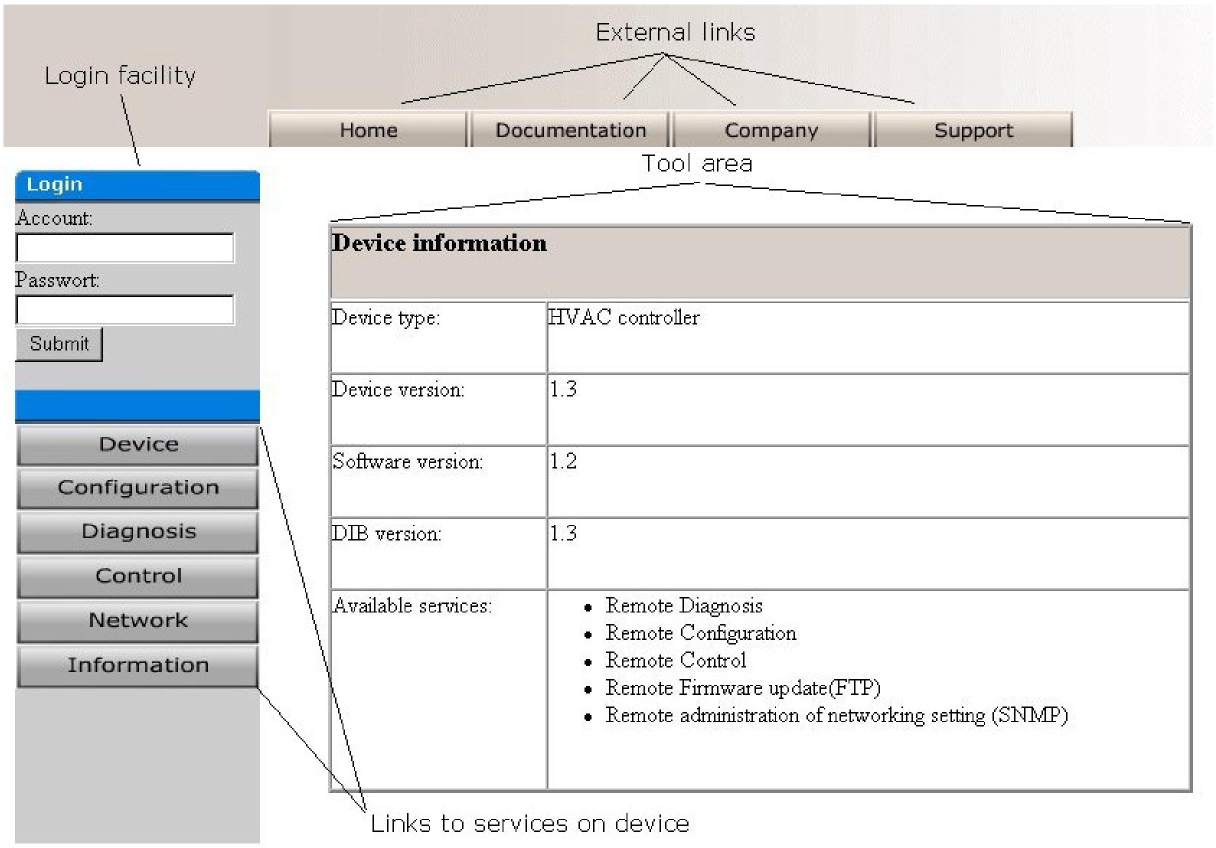
The user selects parameter objects and invokes configuration services of the device in order to modify or read settings. The configuration service on device either supplies the acquired setting (get) or modifies the parameter object according to the new settings (set). The requests and replies are packaged in SOAP(+HTTP) messages, where the messages are assembled dynamically according to the information in the DIB and interface definition.

Commandos for device control:

Diagnosis and configuration services are concerned with device data objects described in the DIB. Commandos which can be sent remotely to the embedded application (e.g. for process control) are not covered by these services. Hence, the commando service provides additional operations (commandos) for miscellaneous purposes. Such commandos can be: reset of device, set stand-by mode, or initiate an action (e.g. move robot arm). The user sends commands to the device by invoking the corresponding operation on the command server.

7.2.2 Basic concepts

The basic concept adopted here is that each device can represent itself, as described in chapter 5.5.1, and that the overall presentation is according to the web-presentation of the vendor company. This is done by providing a web-service on the device which supplies HTML pages (overall layout and representation) and Java applets for remote interrogation. The HTML pages allow intuitive navigation through services on the device, provide a login facility and could provide external links to e.g. product documentation, to the company’s web-page or a online support centre. The following figure shows an example layout. The area in the middle of the page is for the interrogation tools, which are loaded dynamically on demand.



Device information

Device type:	HVAC controller
Device version:	1.3
Software version:	1.2
DIB version:	1.3
Available services:	• Remote Diagnosis • Remote Configuration • Remote Control • Remote Firmware update(FTP) • Remote administration of networking setting (SNMP)

Figure 7-2 Overall presentation of device

The figure above represents the entry page. The links on the left hand site are used to load tools which provide access to the services within the device. The tools may reside on the device (in form of Java applets) or may be loaded from a additional server, as discussed in 5.5.3. The figure suggests that there are distinct tools for each service (multiple links), but in the following one single remote interrogation tool will be developed which contains diagnosis, configuration and control.

7.2.3 Classes and objects

The interrogation tool objects and classes can be identified by parsing the requirements and use cases for the prototype here. The next figure shows the problem-domain objects and their associations as identified here.

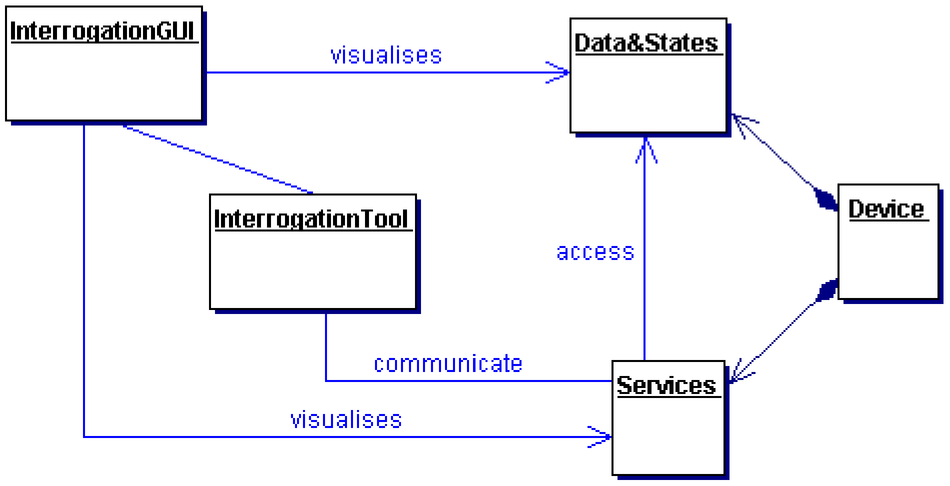


Figure 7-3 Objects and associations

The diagram shows that a device has data and states and provides services which allow access to them. The interrogation tool communicates only with the services on the device. The interrogation tool is associated with a appropriate graphical user interface which visualises data, states and operations available on the device. The diagram neither shows how this information is acquired nor how communication takes place, which is left to the object oriented design later on.

7.2.4 *Objects collaborations – Sequence diagrams*

The following sequence diagrams show interactions between objects which are required to fulfil the overall functionality specified by the use cases. In the following, it is assumed that the user has been already logged in.

Acquiring device information (initialisation):

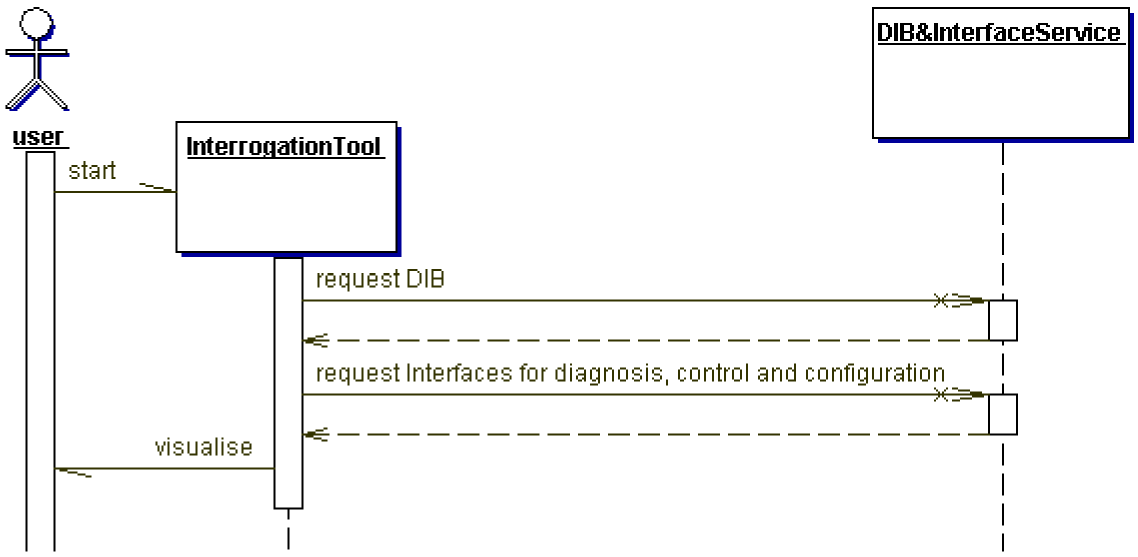


Figure 7-4 Initialisation

The initialisation of the interrogation tool requires that DIB and the interface definition has been received from the device. The sequence diagram above omits important details of initialisation concerning GUI and communication components. The details will be discussed in the design stage.

The use cases for diagnosis, configuration and control are very similar and will be considered in a single sequence diagram for remote interrogation.

Remote interrogation:

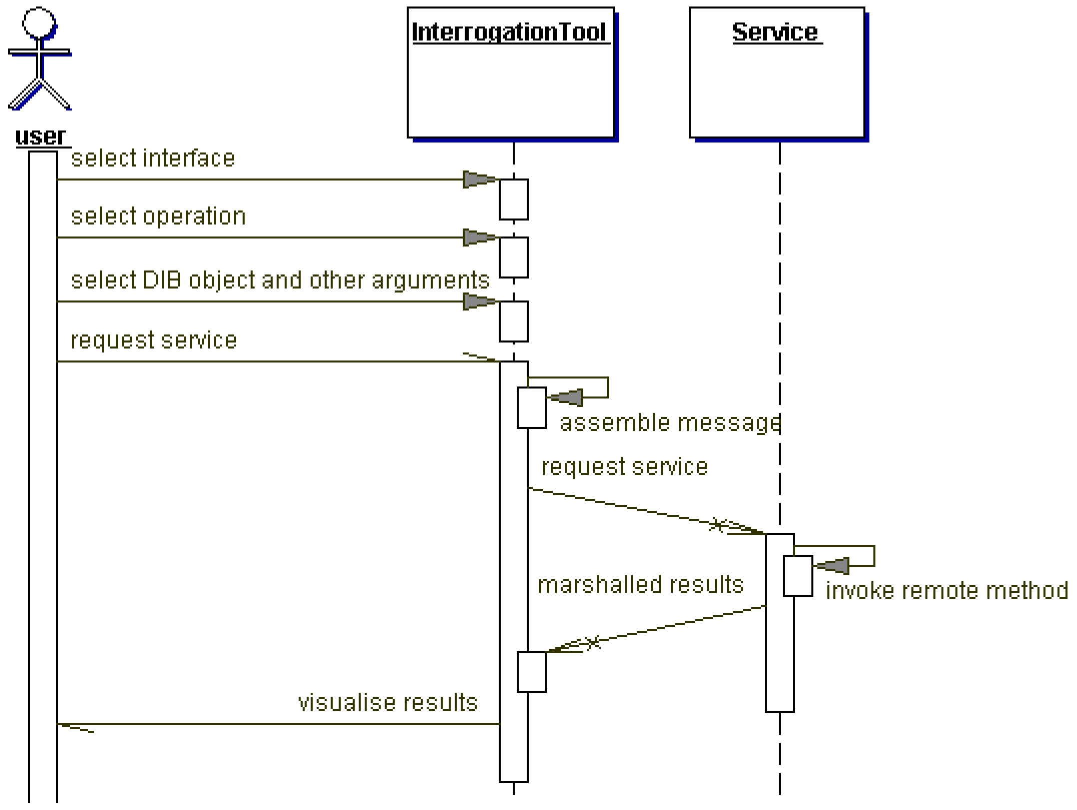


Figure 7-5 Remote interrogation sequence diagram

The user selects the interface, the desired operation and the arguments required and submits its request. The interrogation tool assembles the corresponding message and send it to the corresponding service. The basic communication paradigm is remote method invocation, by the means that the remote service invokes the specified operation on the service. The figure shows a sequence diagram appropriate for analysis which omits the fact that interaction with the user is over the user interface and that the assembly of messages and the communication is not done by the interrogation tool itself but by appropriate communication components.

7.3 Design

7.3.1 The device model

The device model represents the device with its data states and operation and provides an abstraction for the interrogation tool. It is used by the GUI to map the device model in an appropriate user interface representation and by the adaptive stub to assemble messages. The DIB and the interface (service) descriptions represent an abstract device model for diagnosis, configuration and control in XML. This XML descriptions are mapped to appropriate objects which allow convenient access to other objects of the interrogation software.

The mapping between XML-descriptions (data or interface) and objects (classes) are performed as follows:

- Top level elements in XML which contain further elements are modelled as single objects.
- Attributes in XML become also attributes of objects.
- Sub-elements, without descendants become attributes.
- Sub-elements with descendants are modelled as objects, which are aggregated by top level object

Listing 6-9 shows a service interface description of the diagnosis server. It illustrates that a service component has a name and a defines a list of methods which can be invoked remotely. It further specifies that each method has a method description and arguments which must be objects known by the device (device object: e.g. states, process data and states). The class diagram shows an appropriate modelling of a service:

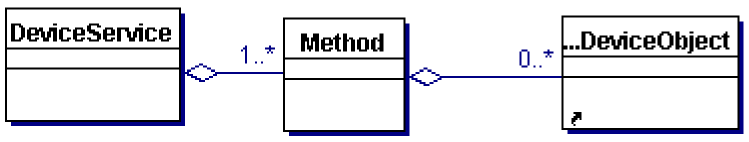


Figure 7-6 Device service model

The device name is an attribute of the `DeviceService` class and the method name and description are attributes of the Method class. As shown here, the cardinalities specified in the DTD and implemented in XML can be mapped to cardinalities marking the associations between objects.

The `DeviceObject` class specifies DIB and other objects which can be used as arguments. The device objects can be of different types (integer, float, string etc.) but also have common attributes like a name (ID) or attributes specifying access rights or GUI information. Hence, device objects are refined by specialisation (inheritance). The next figure shows how type specific classes specialise the `DeviceObject` class and the resulting overall device model.

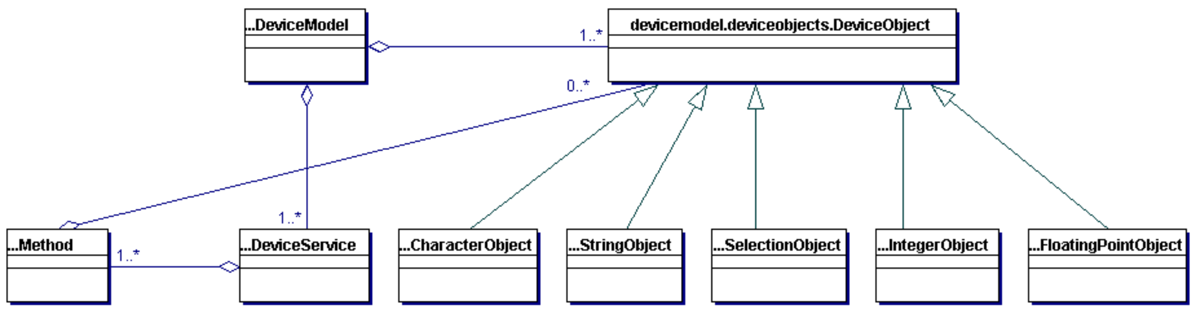


Figure 7-7 The Device model

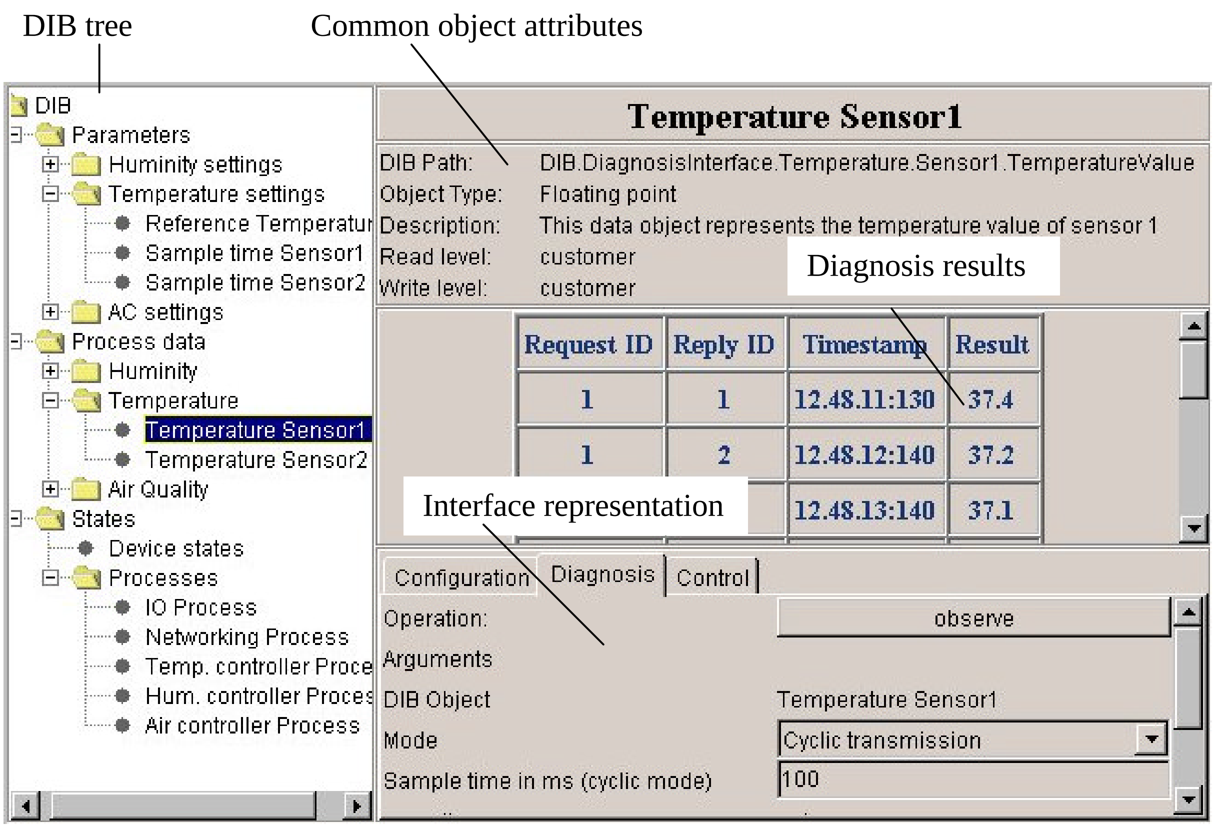
The `DeviceObject` class contains common attributes and the descendant classes contain attributes special to their type. The distinction between state, process data or parameter objects is not made by inheritance because this leads to more complexity. The distinction is rather made by a single type attribute in the `DeviceObject` class as suggested by the prototype pattern described in [GAMMA et. al. 96]. The device model comprises (aggregates) both the device data objects and services.

7.3.2 User interface design of generic tools

The design of the user interface of generic tools is probably the most challenging part, because user interface elements must fully adapt to the device peculiarities at run-time. The approach taken here is to extract the information for the user interface from the DIB. Although, there are some standard available (most of them based on XML) like UIML (User Interface Mark-up Language) and PGML (Precision Graphic Mark-up Language), we constrained the initial prototype to a simple representation (1 to 1 mapping) of the DIB for two reasons:

- Sophisticated user interfaces require complex interpreters and extensive description of user interface components. Hence, the memory space on device to store these description will increase.
- Implementations (Interpreters) of such standards are not available yet.

The approach taken here is very intuitive and provides a *natural presentation* of operations and data objects on the device. As illustrated in 6.3.3.1 the DIB is a simple XML-document structuring and describing data objects on the device. A XML-document has always a tree structure, so that the DIB can be easily represented with a tree component. The data objects and methods in the interfaces have pre-defined attributes which can be represented with standard user interface components. Hence, the whole GUI can be dynamically generated with the information in the DIB and interface definition. The next figure shows a dynamically generated GUI, where the user has selected a data object for diagnosis.



Temperature Sensor1

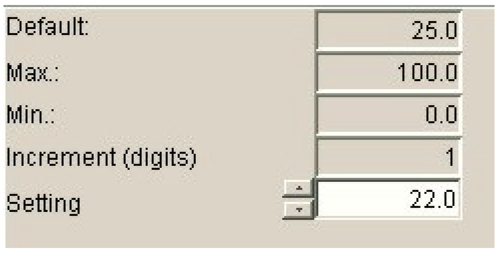
DIB Path:	DIB.DiagnosisInterface.Temperature.Sensor1.TemperatureValue
Object Type:	Floating point
Description:	This data object represents the temperature value of sensor 1
Read level:	customer
Write level:	customer

Request ID	Reply ID	Timestamp	Result
1	1	12.48.11:130	37.4
1	2	12.48.12:140	37.2
		12.48.13:140	37.1

Operation:	observe
Arguments	
DIB Object	Temperature Sensor1
Mode	Cyclic transmission
Sample time in ms (cyclic mode)	100

Figure 7-8 Dynamically generated graphical user interface

The tree on the left hand site shows the structure of the DIB objects and the grouping of objects on the device. At the top-right the information about the object is represented which is also taken from the DIB. The interfaces are represented as a set of method which require a set of arguments at the bottom-right. The arguments are either objects which are described in the DIB or other attributes (like mode above) which must be represented explicitly. The representation of data objects for diagnosis here is always a table (or a more sophisticated a diagram) where the results are represented with additional (extended) information. For configuration this representation is unsuitable because the user must be able to input new setting. For configurable parameters (settings), each object type is given another view, allowing appropriate interaction for this type. The following figure illustrates such a representation for float type objects.



Default:	25.0
Max.:	100.0
Min.:	0.0
Increment (digits)	1
Setting	22.0

Figure 7-9 GUI component for float type parameters for configuration

Each time the user selects another DIB object, the object specific representation in the middle-right area of Figure 7-8 is changed. The representation is dependent on the data object type and the service through which is accessible. The two figures above illustrates the representation for float type data. In case of diagnosis all data types can be represented in a table (or diagram) no matter which type. Components for configuration parameters are more complex and must allow convenient interaction. In the example representation above the user gets details about the parameter (like its range, possible increments etc.) and can set the actual setting conveniently by the mouse.

An user interface component which represents a device object is composed of several *atomic* GUI components as they are defined in Java. To achieve a modular and reusable user interface design the composite components can be implemented with aid of Java-Beans. Each device object type is given a corresponding representation in form of a Java-Bean, which encapsulates other GUI components and provides uniform access to them. Chapter 7.3 describes the overall software design of the interrogation tool with particular emphasis on the interaction between representation (Java-Beans) and the device data object models.

7.3.3 Representation of the device model

Chapter 7.3.1 has shown an appropriate device data model and last chapter introduced user interface components (Java-Beans) which can represent the information in the model to the user. This chapter illustrates how the representation can be managed and how a proper design can reduce coupling between objects. There are two design pattern described in [GAMMA et. al.96] which are relevant to the design here: The observer pattern which is used to design interactions between model classes and user interface classes, and the variants pattern which is used to map user interface components (Beans) to models. The next figure shows the design used to represent the device model.

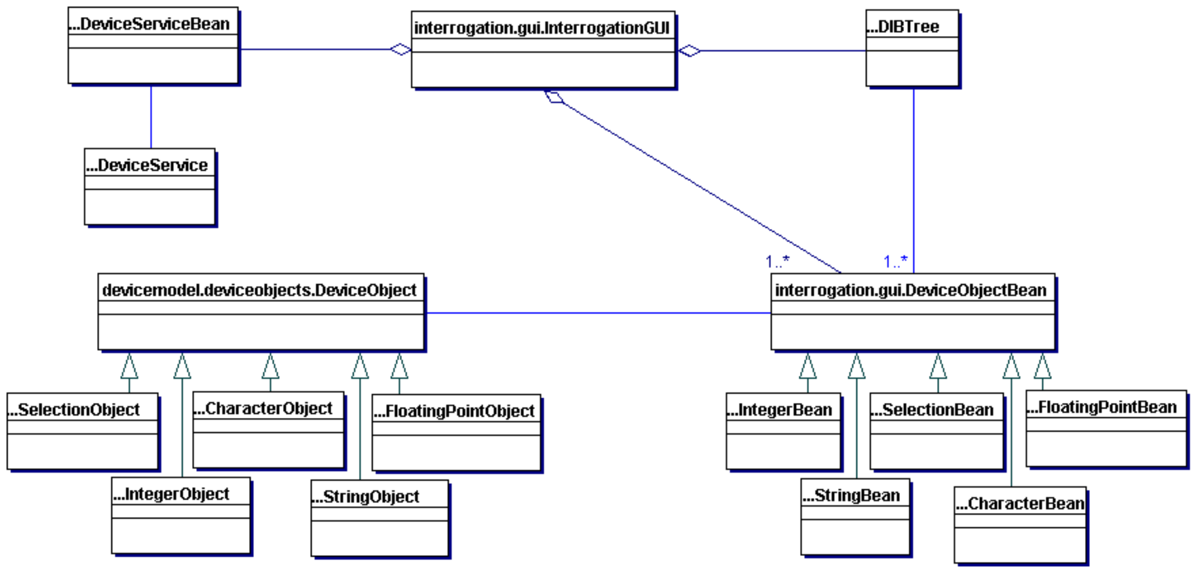


Figure 7-10 Representation of the device model

Each class in the device model discussed in chapter 7.3.1 is associated with a corresponding user interface component (Bean). User interface components and model object implement the observer pattern similar to the model-view-controller model, whereby the control and view is merged in a single user interface component. The `DIBTree` class facilitates browsing through the DIB. The user interface is then modified according to the selected DIB object and the device service (the selected DIB object is presented to the user as shown in Figure 7-8) .

The variant pattern – which basically makes use of polymorphism of object within a inheritance tree – allows that each data object variant gets it corresponding user interface component variant at run time (late binding). Factory patterns [GAMMA et. al.96] are used to reduce complexity of generating these pairs of models and user interface components.

7.3.4 Communication classes

The sequence diagram shown in Figure 7-5 in the analysis has shown how overall communication between interrogation tool takes place. It pointed out that communication is implemented as remote method invocation on a service using pre-defined device data objects as arguments, but it omitted how communication components specified in chapter 6.1 implement the communication mechanism. The approach taken here for design and implementation is according to that given chapter 6.3. The next sequence diagram depicts the objects (communication components) involved in communication at the tools site.

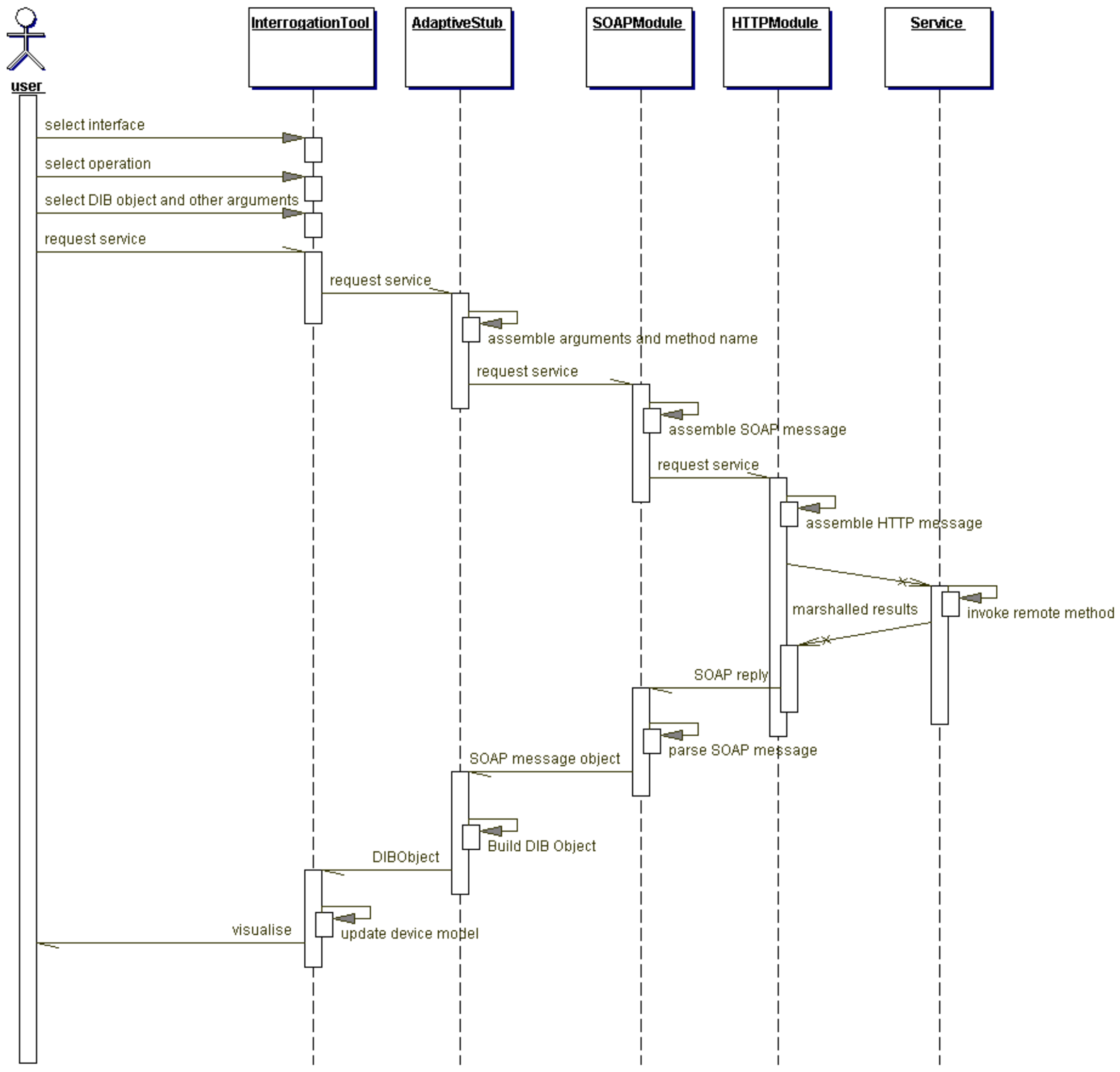


Figure 7-11 Interaction between interrogation tools and device service

The user selects the DIB objects, the service and a method and submits the request. The adaptive stub assembles the message according to the selections of the user, by the means of that it collects the information needed for a SOAP message from the device model and assembles SOAP message object which represents a SOAP message. The `SOAPModule` object serialises the message object into a SOAP (ASCII text) message and passes it to the `HTTPModule`. The `HTTPModule` object carries out the communication with the corresponding communication (HTTP) module at the device. Finally the device service receives the message and invokes the specified method as discussed in chapter 6.3.2. The results are serialised in a SOAP+(HTTP) message and sent back to `HTTPModule` object at the interrogation tools site. The `SOAPModule` receives the message from the `HTTPModule` parses the SOAP message and assembles an appropriate SOAP message objects which is passed to the adaptive stub. Actually, the parsing and assembling of the SOAP message object is done in cooperation between adaptive stub and SOAP module. As suggested in chapter

6.3.1.1 the SOAP module (or class) includes a simple XML-parser and the stub object implements a handler functionality which is invoked by the XML-parser while parsing the SOAP message. The SOAP message object is built by the stub during successive invocations of the parser. Finally, the adaptive stub generates a appropriate DIB object which represents the result of the method invocation and passes it to the interrogation tool, which updates the corresponding DIB object value in the device model.

The important design decision taken here is that whole communication occurs asynchronously as indicated by the half arrowheads in the sequence diagram. Each communication class contains Threads which allow non-blocking request allowing other Threads to continue as soon as the service is requested. This has two consequences: Firstly, the user interface does not freeze after a request and hence the user can initiate further interaction and secondly Threads must be synchronized in their access to shared objects using monitors [FLANNAGAN01] in Java.

7.4 Implementation, integration and testing of prototype

The implementation and integration can not discussed in detail because of the lack of space given here, but the overall approach should be outlined briefly. The implementation and integration were performed bottom up, by the means of that first the communication modules at the device and the interrogation tool has been implemented and tested fully, because communication was the most critical part here. Performance and memory requirements has been assessed by using small, simple and pre-defined messages in the communication modules. After, testing these sub-components the classes required for device model has been implemented and the system has been tested again using a dummy interrogation tool with simple GUI. The interrogation tool (class) logic has been implemented and integrated but using a static and simple GUI. Parsing the DIB and interface definitions has been included and the stub has been made adaptive. At the end, the static GUI is replaced by a dynamic one as described in chapter 7.3.2, which was able to adapt to the information in the DIB and interface definitions.

Class testing was performed with JUnit, a test suite for Java which enables effective automatic testing. Modules on the device has been tested similar while implementation. Overall functional tests has been performed by a service technician in the system. Most important to the discussion here are performance and load tests, because it was unclear whether this approach illustrated here works with embedded systems. The overall results of these tests are discussed in the following chapter.

8 Assessment of architecture and protocols

8.1 Requirements on device, network and the software design of tools

Initially in the analysis, prototypes has been build that evaluated the requirements for device and network for reasonable communication using ASCII serialisation (SOAP). The tests were made with a dummy application using the target hardware but running dummy software of a HVAC controller. These tests were appropriate to make estimates whether the approach taken here is feasible or not. But real-world application require more computation time and memory, so that the results given in the analysis part are not very representative. Hence, the solution given here for architectures, protocols and interrogation tools has been ported into a real-world device performing complex barcode reading. Barcode reading is complex because it requires complicated I/O peripherals (lasers, sensors) and very computational intensive algorithms for acquiring and decoding barcode information. Hence, tests on such a device will be representative for complex real-world embedded devices.

8.1.1 Memory requirements

Chapter 5.2 has shown the target embedded system and its capabilities. The target embedded system has very large Flash memory (256 Kbytes internal + 256 Kbytes external) for program code and a relatively large RAM memory (256 Kbytes) for temporary data. The software components running on the device and their required memory space are listed in the following table.

Component/Information	Flash (internal) in Kbytes	Flash (external) in Kbytes	RAM in Kbytes	Description
Application (Barcode)	120	-	60	Programs are preferably stored in internal Flash, because of access times.
TCP/IP stack	80	-	80 (using large buffers)	Using large buffers and packet sizes will enhance communication performance.
HTTP+SOAP	10	-	40	Very lean implementation.
Interrogation servers	20	-	20	RAM for shared memory between servers.
DIB and interface definition	-	380+40 (over 220 data objects)	-	XML-documents can be stored in external flash.
Overall	230	420	200	

Table 8-1 Memory requirements on device.

Internal Flash and RAM memory is close-fitting, proving the statement in the analysis part that this approach is only applicable to large-scale embedded systems. The external Flash is to small in this configuration where the XML-files are simple stored as ASCII-text. Chapter 6.2.5.1 pointed out that compression of XML-files can halve the message sizes for messages in the range of 1-10 Kbytes but that compression before transmission introduces additional sender and receiver latency. But compression to reduce the required memory size on the device does not require that the files (DIB and interface definitions) are compressed/decompressed on the device, by the means of that files are compressed once on a PC, loaded to the device and supplied to the interrogation by the device. Compressing the large DIB and interface definitions gives excellent results: The overall file sizes of 420 Kbytes is reduced to 40 Kbytes using JAR (Java Archive, which uses gzip as compression algorithm). The reason why these compression ratio here (0.1) is not reached in the experiments in chapter 6.2.5.1 is because that the file size here is larger, the DIB has more regularity than SOAP messages, and the LZ77 buffer is not bounded to 4 Kbytes (this was made to become representative results for compression on embedded systems). So the storage problem is solve by compression. For this reason the interrogation tool (128Kbytes in JAR) could be also stored in the external Flash of the device (as suggested in chapter 5.5.1).

8.1.2 Performance requirements

The prototype given in chapter 5.3.3 evaluated the device performance requirements running a dummy process control task on the device which required minimum computation time. Besides the SOAP and HTTP implementations were not fully implemented and therefore very fast. The measurements have shown that a virtual temperature value can be produced, processed and transferred every 2-3 milliseconds. Initially the barcode reader device was able to produce barcode information every 1.5 milliseconds, requiring the processor for about 0.8 ms. Tests have been performed using high-speed LAN (100Mbit/s) to evaluate the maximum transfer rates for barcode information (15 characters). The measurements have shown that the device is able to transfer barcode information every 3-4msec to the interrogation tool using UDP as transport layer protocol and 6msec using TCP (with large buffer sizes). In the case of the barcode reader application this rates are more than sufficient, because in normal use they will never reached. Another phenomenon which is preserved here is that there is a latency (delay) between production of the result at the device and the indication by the interrogation GUI for about 5 messages using UDP(delay of roughly 25msec). This effect is even increased when TCP is used as transport protocol (because of sliding window buffers). The channel memory and delays in LANs (using single HUB for interconnection) is negligible (about

3msec), therefore this delay is most probably produced by Thread communication in the interrogation tool that use buffers (producer/consumer with shared objects) to transfer information. Further tests have shown that improper Thread priorities will increase this effect further. Stating other computational intensive programs on the service PC lead to further delays. Therefore, the maximum rates at which data can be transferred for reasonable diagnosis using LANs has been identified as below than 10msec per message for this kind of application, which is appropriate for most diagnosis purposes.

8.1.3 Requirements on bandwidth and delays

Initial prototypes for communication in chapter 5.3.3 have shown that bandwidth and delays are more critical than the device performance. The tests performed for the interrogation tool prototype have confirmed these results and shown that data transfer (barcode information) can be performed only every 200msec, but that delays vary between 100mesec-1sec, using internet infrastructure and modems (56Kbit/s) as suggested in chapter 5.1.2. Packaging multiple results in single SOAP messages and buffering did not bring enhancements because the delays are primarily introduced by network resources but not by transmission times due to low bandwidth. In contrast to that, these mechanisms have brought significant enhancements for direct modem access (see chapter 5.1.3), which provided reasonable diagnosis at rates below 100msec per messages and insignificant delays. The topology suggested in chapter 5.1.1 could not be tested because the problem of reconfiguring firewalls at the customer network could not be solved in the short time given for this project. But considering today’s company networks, which provide high speed access to the internet, this approach will probably achieve better results than at least the one which uses modems to access the Internet over an ISPs and Internet infrastructure.

8.2 Portability to other devices

Because the given solution here for interrogation software is generic, the portability to other devices is a very important issue. The interrogation tool itself is written in Java (Applet), and therefore can run on every service PC without modifications, requiring only a modern standard browser (Netscape or IE). Old browsers require a Java plug-in because they are not able to support Java-Swing GUI components, but a plug-in can be downloaded and installed at run-time.

The software running on the device – namely the interrogation servers, DIB&Interface server and the communication software (SOAP+HTTP+Skeleton) – is more critical, because the components have to compiled for each embedded system platform. This point is addressed in the initial prototypes by using only ANSI-C for implementation. System calls (memory allocation, mailboxes) has been concentrated to a few C-functions, which can be modified for other platforms. The solution has been ported to other barcode reader devices using different hardware and using other operating systems. The port has been performed in 2-3 days by a single person, requiring little modification to the software and the implementation of DIB and interface definitions. This is very short compared to the effort that is required to develop specific solutions each time from scratch.

8.3 Maintainability

Maintainability means here the effort which has to be taken to maintain additional software and information (DIB etc.) but not the device maintenance for which the software is intended for. In terms of maintainability the solution provided here very efficient, because once the generic interrogation tool has been developed and the software required on the device has been ported, the maintenance is limited to the DIB and interface definitions. In order to illustrate the enhanced maintainability here, we will show a simple maintenance example:

A new device software has been created which provides additional parameters, process data and commands. The following steps has to be performed to adapt the interrogation software solution to this new version:

1. Add additional parameters, process data and states to DIB
2. Add additional methods to the command server interface definition according to the new commands required.
3. Modify tables in skeletons which map DIB pathnames to real data objects on device or in case of the command server skeleton the mapping between method names to method addresses must be modified. Hopefully, this step can be performed automatically in future projects with an interface compiler.
4. Upload new software, DIB and interface definitions to the device, preferable remote by using FTP.

Except step three, the maintenance is limited to modification to XML-documents which requires little effort. Step three is done by hand but requires only little modifications to table entries. The most markable advantage here is that the interrogation software has not to be changed as long as additional data types in the DIB are note required. That means for instance if new data types are introduced in the DIB, the adaptive stub at the interrogation tool has to be changed, too. As soon as the types of data objects in the DIB are stable interface compilers become available, the maintenance of interrogation tools is reduced to modifications to XML-documents.

8.4 Problems identified in the prototype

The main problems identified in the prototype concern the usability of the interrogation tool prototype. Service technicians and customers which have tested this solution have criticised the following points:

- The representation over the DIB tree allows only diagnosis of one data object. Sometimes it is beneficial to observe a set of data objects on the device in parallel, which is not possible with the prototype given here.
- Some settings put constraints to others. As an simple example if the user selects a barcode type to be decoded then only a set of barcode length are allowed. In the implementation here all settings are configured independently, so that improper pairs of settings can be configured.

The first point requires more complex algorithms for dynamically creation of the interrogation tool's GUI. XML-Schemas, which allow the definition of more complex data types as well constructs like associations and inheritance could be used in future to address point two. Alternatively, the embedded application could check settings whether they are valid or not and can return an error message if improper settings are detected. Adding these checks at the configuration server is not reasonable because this would require that the configuration server is modified each time when new parameters are added.

The initial aim was to provide a solution which reduces cost for service and maintenance by facilitating remote access on embedded devices but also to reduce cost for development of interrogation tools by providing support for generic tools. The latter aim conflicts with the two points criticised here because enhancing usability and representation (GUI) – especially for generic tools and GUIs – will increase costs for development of generic tools significantly. Hence, it is up to the device vendor company who develops the tools whether a more complex solution which provides better usability and enhanced GUI outweighs the disadvantages of more development costs.

8.5 Summary

This chapter summarises the advantages and disadvantages of the approach here for remote and generic interrogation tools identified in the analysis in chapter 5 and the assessment in the preceding chapter.

The advantages are as follows:

- Using ASCII serialisation provides real interoperability between devices and interrogation tools. Especially using SOAP and HTTP seems to be the most proper approach to overcome firewall barriers.
- SOAP is simple and extensible and can be used over various transport protocols. As the requirements for communication vary by application (e.g. diagnosis and configuration) this is an important advantage to support a wide range of interrogation applications.
- With the DIB and interface definitions in XML, we identified a simple way to support generic interrogation tools. Together with the adaptive stub and dynamic GUI at the interrogation tool, this approach provides real independency from the device.
- The development of generic interrogation tools is limited to the development of dynamic GUIs. Other classes for communication, device model and interrogation logic can be reused. Communication details are hidden from the tool developer.
- The approach gives enhanced portability and best maintainability for interrogation tools provided that software components for the device are largely written in ANSI-C.

Especially the assessment has identified disadvantages, which are listed below.

- SOAP (ASCII serialisation) messages contain large overhead, that increases the bandwidth required.
- SOAP serialisation is computationally complex.
- Diagnosis type interrogation applications require certain QOS of networks. Diagnosis facilities over internet are limited.

- Dynamic generation of GUI is very complex. Therefore better usability and special feature in GUI are very difficult to implement and require high development cost.
- The approach shown here is only suitable for large-scale embedded system.

The approach for protocols and architectures provided here can satisfy the initial requirements for remote and generic interrogation tools for embedded system, but this is not for free. Enhanced maintainability and reduced development cost are at the expense of usability and fancy GUIs. Remote access over internet reduces costs for device service and maintenance but real-time diagnosis is limited. Using ASCII serialisation provides best interoperability and is the best solution for crossing firewall boundaries, but also requires high protocol overheads and computation time on device. The applicability of enhancements like compression and buffering is limited, so that the disadvantages could not be mitigated. Future projects might consider reservation of network resources as suggested in chapter 6.2.6 with RSVP that is essential to enhance diagnosis facilities.

9 Conclusions and future work

The distributed object model as applied here for embedded devices and interrogation tools and the associated communication mechanism RMI has been identified as a reasonable basis for remote interrogation of device with generic tools. The concepts associated with that model as abstraction, encapsulation and communication over well defined interfaces has been a reasonable basis to access the devices uniformly. A very flexible architecture has been defined with aid of the distributed object model and RMI as basic concept for communication which considers remote access and supports generic tools. As in principle, the approach taken here was promising and feasible from the research and design point of view, but the realisation of these ideas in context of performance and memory constrained embedded systems and public inter-networks was faced with major problems and therefore very challenging.

A major hold-up for implementing the initial ideas of a architecture for remote and generic interrogation tools based on internet standards was that these standards – namely SOAP and XML – are specified recently and that fully implemented off-the-shelf components are not yet available. This was not critical for this feasibility study here, besides the fact that it increased the work significantly and the modules have been implemented only partly, but standard of-the-shelf components will be very important for future developments. The effort taken here for research, design and realisation for a prototypical implementation was immense. Therefore, a full implementation is only reasonable as soon as off-the-shelf components are available. Nevertheless, the predicted effort here in this feasibility study is only acceptable if the long term advantages of the solution concerning enhanced maintainability, portability and remote access to devices are considered. These long term advantages will become significant if the given approach is widely-applied to all devices of a vendor, or even better if the development costs are shared between several vendors in a joint project.

Contemplating the results here for remote access on devices, SOAP and the related architecture defined here was the only feasible approach for internet infrastructure. The problem associated in using public inter-network are the same as for every distributed application, which involves some real-time communication. Several enhancement methods has been shown here for efficient communication but they could not solve the problem that quality-of-service in the internet is absent. This area is subject to extensive research, because of the high commercial interest in real-time data transfer over public inter-networks as voice-over-IP, video on-demand and related applications. Breakthroughs concerning QoS in inter-network will also support future work here.

Therefore the recommendations for future work made here in the following are concerning the realisation of the architecture defined here:

- Concepts like VPN and NAT should be considered as soon as they are widely used by companies.
- The communication modules here should be replaced with off-the-shelf components as soon as implementations are available. If off-the-shelf components are not available for the communication modules on the device, the standards (HTTP and SOAP) must be implemented according to the specifications to replace preliminary versions here.
- DTDs used for data type and interface definition should be replaced by XML-schemas in order to allow a more generic approach to serialisation with SOAP. This approach will probably make the stubs and skeletons superfluous. If stubs and skeletons are still required an interface compiler should be developed which can create those automatically.
- Standards PGML and UIML concerning the description of the graphical user interface for generic interrogation tools should be included to enable the development of sophisticated and dynamic user interfaces.

Future work will also require further research in areas above to assess the applicability to embedded systems. But as the these technologies and standards advance device technology will advance, too, and more complex and extensive approaches will become feasible. Therefore single most important recommendation here is that future work must base on internet standards in order to profit from the innovations and rapid growth in this area

References and Bibliography

References

Books:

[CDK00]	George Coulouris, Jean Dollimore, Tim Kindberg, Distributed Systems: Concept and Design, 3rd edition (2000)
[SOMMER01]	Ian Sommerville, Software engineering, (2001)
[BACON98]	Bacon Jean Bacon, Concurrent Systems: Operating Systems, Database and Distributed Systems (1998)
[COMER00]	Douglas E. Comer, Internetworking with TCP/IP (2000)
[HALSALL96]	Fred Halsall, Data communications, Computer networks and Open systems(1996)
[PRESSMAN01]	Roger S. Pressman, Software engineering – A practitioners approach, (2001)
[SALOMON00]	David Salomon, Data compression – A complete reference(2000)
[SCRIBSTIV00]	Kennard Scribner and Mark C. Stiver, Understanding SOAP (2000)
[MEANS00]	Elliotte Rusty Harold, W. Scott Means, XML in a Nutshell, O'Reilly (2000)
[FLANNAGAN00]	Flannagan, Java in a Nutshell, O'Reilly 2000
[GAMMA et. al.96]	Gamma, Helm, Johnson, Vlissides, Design patterns 1996
[POHLMANN00]	Firewall systems, 3rd edition (2000)

Journal articles, papers and specifications:

[DOBBS98]	A real time weather station – Merging embedded devices with Internet Technology, Dr Dobb's journal, October 1998, Page 40-46
[OEZASLAN00]	Hakan Oezaslan, A framework for scalable security architecture for internet based mobile (multimedia) systems, with role dependent access, 2000
[BINSTOCK00]	Andrew Binstock, The Middleware Layer Model (2000)
[EMBEDCORBA00]	CORBA for Embedded Systems Yarisa Jaroch Stephan Schulz (2000)
[W3CSOAP1.0]	W3C SOAP specification 1.0
[W3CSOAP1.1]	W3C SOAP specification 1.0

[OMGCORBA01] The Common Object Request Broker: Architecture and Specification (2001)

Bibliography

Internet ressources:

http://www.esconline.com/, white papers, presentations and embedded system conferences

http://msdn.microsoft.com/soap/, SOAP and related topics

http://www.sdtimes.com/cols/middlewatch_017.htm, Andrew Binstock, The Middleware Layer Model

http://www.xml-rpc.com, XML-RPC specification

http://engr.arizona.edu/~sschulz/ece678/ProjectProposal.html, embedded CORBA

http://www.omg.org, OMG web-side

http://www.omg.org/technology/documents/formal, CORBA specifications

http://www.w3.org/TR/SOAP, SOAP specifications

http://www.apache.org; SOAP/1.0 Serialization Library Version 0.3

Glossary

Device	Used here to refer to embedded systems in general
RMI	Remote Method Invocation
	Remote method invocation in distributed object model
RPC	Remote Procedure Call
	Remote procedure call in procedural languages.
HTTP	HyperText Transfer Protocol:
	Protocol used in the WWW to transfer text based information like HTML
HTML	HyperText Markup Language:
	Markup language using Tags to format strings. Used in the WWW
XML	eXtended Markup Language:
	Markup language, which allows defining own tags by a DTD or Schema.
DTD	Document Type Definition:
	Declaration for an XML file.
XML-Schema	Declaration of data types in XML.
SOAP	Simple Object Access Protocol
	Protocol based on XML
CORBA	Common Object Request Broker Architecture
	Middleware for distributed systems.
SNMP	Simple Network Management Protocol
	Protocol for management of network devices like routers, switches etc.
MIB	Management Information Base
	Information base of network device, which can be configured with SNMP
OSI-Model	Open System Interconnection:
	Reference Model for networking.