TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Engineering

Denis Konstantinov 111615 IASM

Embedded service oriented microcontroller architecture.

Extensible client-server communication architecture for small devices

Master thesis

DRAFT

Supervisor: Peeter Elervee
Associate Professor at the Department of Computer Engineering / Ph.D., Dipl.Eng.

Tallinn 2013

Author's Declaration

This work is composed by myself independently. All other authors' works, essential states from literary sources and facts from other origins, which were used during the composition of this work, are referenced.

Signature of candidate: Date:

Annotation

Current work introduces conceptual approaches for implementing an extensible service oriented client-server application on a small microcontroller. This is a general-purpose transport and hardware independent embedded server that uses remote procedure calls as primary communication protocol. This server looks like remote service that could provide defined functions to the client.

This paper has a research about available service oriented architecture technologies and reviews them in the context of embedded resource constrained hardware systems.

There is also an implementation of a small embedded service and corresponding client. Service part is implemented on STM32F1 family ARM Cortex-M3 microcontroller and runs FreeRTOS operating system. Client part is realized as Java library RPC stub and Android demo application.

Client and server are communicating using lightweight JSON-RPC protocol over Bluetooth wireless communication line. Some few RPC call functions are implemented to show the capabilities of such client-server architecture.

Annotatsioon

Käesolevas töös kirjeldatakse kuidas rakendada teenusorienteritud klient server arhitektuuri sardmikrokontrolleril. See on üldotstarbeline transport ja riistvara sõltumatu rakendusserver, mis kasutab kaugprotseduuri väljakutseid kommunikatsiooni pidamiseks. Mikrokontrolleris jooksev programm näeb välja nagu serveri teenus, mis pakkub kasutajale ettenähtud funktsionaalsust.

Antud töö eesmärk on teha uuring teenus orienteeritud arhitektuuride ja tehnoloogiate kohta ning analüsida nende kasutamist piiratud ressursidega sardsüsteemide realiseerimises.

All on toodud realiseeritud teenusorienteritud sardsüsteemi kirjeldus ja sellele vastav kliendi rakendus.

Serveri pool on tehtud STM32F1 pere ARM Cortex-M3 mikrokontrolleri baasil, kus jookseb FreeRTOS reaalaja operatsioonisüsteem. Kliendi rakendus on mobiilsel platformil kasutatav programm, mis kasutab Javas kirjutatud teegi kaugprotseduuride väljakutsemiseks.

Klindi ja serveri vaheline kommunikatsioon toimub labi traadita Bluetooth kanali kasutades JSON-RPC protokolli. Süsteemis on tehtud mõned funktsioonid, selleks et näidata antud klient-serveri arhitektuurilisi omadusi.

Contents

Acronyms

API Application programming interface. 9

CAN Controller Area Network. 47

CRUD Create, read, update and delete. 17, 30

DMA Direct Memory Access. 43

FTP File Transfer Protocol. 11, 12

HMI Human Machine Interaction. 7

HTTP Hypertext Transfer Protocol. 6, 12, 14, 15

IPC Inter-process communication. 19, 24

ISR Interrupt service routine. 46

JMS Java Message Service. 14

JSON JavaScript Object Notation. 22, 25

JVM Java Virtual Machine. 53

REST Representational state transfer. 15

RPC Remote procedure call. 10, 30

SMTP Simple Mail Transfer Protocol. 12, 14

SOA Service-oriented architecture. 6, 9, 10, 13, 15, 28, 60

SOAP Simple Object Access Protocol. 6, 12–15

TCP Transmission Control Protocol. 14

UDDI Universal Description, Discovery and Integration. 11–14

URI Uniform Resource Identifier. 16

WSDL Web Services Description Language. 6, 12–15, 51

XML Extensible Markup Language. 6, 12, 13, 15, 25

YAML YAML Ain't Markup Language. 25

1 Introduction

Computers are very essential in our life. Computer is an electronic device that is used in almost every field. It is very accurate, fast and can accomplish many tasks easily. In early days computers were only used by the government and army to solve different high computational tasks. After invention of low-cost microprocessors, computers became available to every person. Nowadays there are billions of personal computers and they are almost at every home.

Present day computers may be divided into two groups: very big and very small systems. In one group are mainly servers and server farms, and in the other are mainly embedded systems. The gap between these groups becomes more wider, because of the availability of new small and low-power devices, which computational power raises constantly. Lot of people prefer now to buy a tiny laptop instead of traditional workstations with a monitor and computer case under the table. There is also a more smaller group of devices, that are implemented for a particular purpose - embedded computers. Every home has several examples of embedded computers. Any appliance that has a digital clock, for instance, has a small embedded microcontroller that performs no other task than to display the clock. Modern cars have embedded computers onboard that control such things as ignition timing and anti-lock brakes using input from a number of different sensors.

Today, there is very little or no communication between embedded devices and large servers in the web. The problem is not only in the communication infrastructure, because the current communication technologies are able to provide different wired and/or wireless connections. The problem is how we design and implement embedded systems. While we try to keep big systems as open as possible (since it is their primary role), we tend to seclude and isolate embedded systems without providing easy ways to add a custom interface to them. Embedded systems are still mainly seen as vendor-specific and task-oriented products, and not as components that can be easily manipulated and reused.

If all classes of devices could speak the same language, they could talk directly to each other in ways natural to the application without artificial technical barriers. This would allow easily creating seamless applications that aggregate the capabilities of all the electronics. The interoperation adds value to all the devices.

This idea comes from conception of Internet of Things (IoT). The Internet of Things is the network of physical objects that can communicate to each other using Internet and embedded technologies. This connections compose an complex system where each member can send information about its state and acquire data about other parties without any intervention of human being. For example sensors at your home could communicate to heater and ventilation system and control temperature and humidity or your alarm can tell all other devices that you are going to wake up soon. This technologies could help to track and count everything and improve the quality of our lives by removing the unnecessary waste and additional cost.

The Internet of Things is quite popular topic of research nowadays. It possibly can change the world like the Internet did. Many companies and universities are trying to find and invent reasonable technologies for implementing this approach.

One of the methods how this communication can be performed is the concept of remote services and service oriented architecture (SOA). World Wide Web Consortium (W3C) defines a "Web service" as:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL[1]). Other systems interact with the Web service in a manner prescribed by its description using SOAP[2]-messages, typically conveyed using HTTP[3] with an XML[4] serialization in conjunction with other Web-related standards.

Services are unassociated, loosely coupled units of functionality. Not only large server system are capable of providing this functionality. Services can also be applicable in the resource-constrained embedded devices.

This work would introduce the concepts how SOA[5] can be in the context of embedded systems. This contains some research of already available technologies for machine-to-machine communications and the implementation of small system prototype, which contains two connected devices and uses service approach.

[1] Web Services Description Language
[2] Simple Object Access Protocol
[3] Hypertext Transfer Protocol
[4] Extensible Markup Language
[5] Service-oriented architecture

1.1 Impact

The impact of the research in this thesis has been started during the accomplishment of internship at the university. I was worked for some company and my task was to develop HMI[6] interface to some embedded system. We were using wireless communication between the control unit and the machine it was controlling. Control unit was a smartphone that was sending commands through Bluetooth protocol. On the other side there was a coffee machine that was receiving and executing that commands.

At the same time i was studying how large enterprise systems communicate to each other. I was reading about web services and related technologies. Then was born an idea that there could also be a "small" device network, where devices communicate to each other.

This was a research project and developed prototype could potentially become a real product. In that case it needs to be connected to existing infrastructure. Coffee machine could provide different remote services: remote coffee product ordering, coffee machine maintenance and acquisition of statistical data, remote payment. This could look like traditional coffee automatic machines at the streets that accept cash, but with a remote wireless control.

I stated to mine the information about different control possibilities. This is how this research became a topic of my master thesis.

1.2 The goal

The purpose of this work is to create a prototype system which has remote service capabilities and make a research of available machine-to-machine communication possibilities. This should be an universal and platform independent architecture, which can be easily ported to any suitable hardware and connected into existing infrastructure.

Already existing hardware are two STM32f103xx family microcontrollers which have 20 and 64 Kbytes of RAM, 128 and 512 Kbytes of flash memory. They are also equipped with UART communication and whole communication need to work using serial line. Remote server with service capabilities should work on that hardware. General requirement for the hardware is low-cost microcontroller with some connectivity, that does not make the already existing system more expensive and in the same time fulfil all required functionality.

Embedded server will be connected to a target device, which is actually a coffee machine, and handle requests from clients by executing various functions on target device. Server should provide a functional interface to the client and know about available functions inside coffee machine. That interface need to be verbose and easy to connect.

Client will be executed on mobile phone and will communicate with remote server using Bluetooth wireless technology. It might be any mobile platform, but the organization decided to try Google Android smartphones first. Android device need to have running program with a graphical user interface, which is able connect to remote service and accomplish functional needs.

We need to be able to switch existing client hardware architecture and the to choose various clients for this embedded service. It might be a Android mobile application, regular desktop computer program or even web application. It is good to have one common client code that will be used in these different environments. We should write a client library, which have bindings to the remote service and handles all the communication. It should provide a convenient interface to a library client.

Coffee machine application is only the example of such architecture. We need a real world problem and device to show all capabilities of such system. Controlled device and the client application could vary. It might be a remote light control at home or any data acquisition and control system at the production plant. During this work such universal and extensible system will be built.

To summarize and make our goal more complete we should make a list and follow it:

1. Make a research about available device-to-device connectivity techniques, software standards and communication protocols. Get current state of the art and find a suitable technology for the existing environment

2. Get familiar with already existing hardware tools, setup programming environment and write some sample programs to test the capabilities.

3. Implement service oriented architecture on a microcontroller hardware.

4. Write some test functionality to prove chosen approach.

[6] Human Machine Interaction

5. Implement a client library software module.
6. Create a test application that will use this client module inside the system and show how it can be applied.

   This work will cover every step in this list and author will try to complete it.

1.3 Outline

The first section will introduce available technologies of implementing machine-to-machine communications. The main research is about the Internet technologies and concept of remote web services, this is because the Internet is already an interconnected network with lots of machines are doing distributed computing and interacting with each other. Lots of problems are already solved there and there are available different technologies and tools. Although, all these already implemented features are not limited only with the Internet and related technologies like TCP/IP and HTTP. Some essential features may be extracted from there and ported to resource-constrained devices. The first section will also cover some connection and interaction possibilities that embedded systems have.

Implementation section covers embedded system prototype, that uses concepts from different machine-to-machine communication technologies. It contains description of architecture and software and hardware was used. Embedded service was implemented on a microcontroller device. Implementation of the embedded server section covers the hardware and software parts of the system.

General purpose client library implementation section describes the details of Java client stub library. There is also implementation of a demo client application in the section next to it. This is the last section here.

2 Background

Internet technology is the environment in which billions of people and trillions of devices are interconnected in various ways. As part of this evolution, Internet becomes the basic carrier for interconnecting electronic devices – used in industrial automation, automotive electronics, telecommunications equipment, building controls, home automation, medical instrumentation, etc. – mostly in the same way as the Internet came to the desktops before. More and more devices getting connected to World Wide Web. Variety of factors have incluenced this evolutions [1]:

- The availability of affordable, high-performance, low-power electronic components for the consumer devices. Improved technology can assist building advanced functionality into embedded devices and enabling new ways of coupling between them.
- Even low cost embedded devices have some wired or wireless interface to local area networks of the Ethernet type. TCP/IP family protocols are becoming the standard vehicle for exchanging information between networked devices.
- The emergence of platform independent data interchange mechanisms based on Extensible Markup Language (XML) data formatting gives lots of opportunities for developing high-level data interchange and communication standards at the device level.
- The paradigm of Web Services helps to connect various independent applications using lightweight communications. Clients that are connected to the service and the service itself may be written using different programming languages and be executed on different platforms.
- Presence of Internet allows existing of small embedded controllers and large production servers in the same network, with a possibility to change information.

The integration of different classes of devices, which employ different networking technologies, is still an open research area. One of the possible solutions is the use of SOA software architecture design pattern.

2.1 Service oriented architecture

> *Service-oriented computing is a computing paradigm that uses services as basic building blocks for application development.* [2]

The purpose of SOA is to allow easy cooperation of a large number of computers that are connected over a network. Every computer can run one or more services, each of them implements one separate action. This may be a simple business task. Clients can make calls and receive required data or post some event messages.

Services are self-describing and open components. There is a service interface, that is based on the exchange of messages and documents using standard formats. Interface internals (operating system, hardware platform, programming language) are hidden from client. Client uses only a service specification scheme, also called contract. Consumers can get required piece of functionality by mapping problem solution steps to a service calls. This scheme provides quick access and easy integration of software components.

Service architecture have been successfully adopted in business environments. Different information systems, that were created inside companies for automation of business processes, are now turned into services which may easily interact with each other. For example, Estonian government uses services to transmit data between information systems of different departments. There are also some free services available. Some Internet search companies like Google, Bing, Yandex provide lots of alternatives how to retrieve data without using regular browser(search , geolocation and maps, spell check API[7]s)

There are available many technologies which can be used to implement SOA [3]:

- Web Services
- SOAP Simple Object Access Protocol, is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks.

[7] Application programming interface

- RPC Remote procedure call is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.
- REST Representational state transfer is a style of software architecture for distributed systems such as the World Wide Web. REST has emerged as a predominant web API design model.
- DCOM Distributed Component Object Model is a proprietary Microsoft technology for communication among software components distributed across networked computers.
- CORBA Common Object Request Broker Architecture enables separate pieces of software written in different languages and running on different computers to work with each other like a single application or set of services. Web services
- DDS Data Distribution Service for Real-Time Systems (DDS) is an Object Management Group (OMG) machine-to-machine middleware standard that aims to enable scalable, real-time, dependable, high performance and interoperable data exchanges between publishers and subscribers.
- Java RMI Java Remote Method Invocation is a Java API that performs the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java objects and distributed garbage collection.
- Jini also called Apache River, is a network architecture for the construction of distributed systems in the form of modular co-operating services.
- WCF The Windows Communication Foundation (or WCF), previously known as "Indigo", is a runtime and a set of APIs (application programming interface) in the .NET Framework for building connected, service-oriented applications.
- Apache Thrift is used as a remote procedure call (RPC) framework and was developed at Facebook for "scalable cross-language services development".
- . . .

This list can be continued. Most of these technologies are inspired by idea of RPC[8]. An RPC is initiated by the client, which sends a request message to a known remote server to execute a specified procedure with specified parameters. The remote server sends a response to the client, and the application continues its process. This idea is described in details in subsection 2.4.

Web Services are the most popular technology for implementing service-oriented software nowadays. Next section will focus on this framework and on the main features that any SOA implementation should have.

2.2 Web Services architecture

2.2.1 Web Services Model

The Web Services architecture is based on the interactions between three roles [4]: service provider, service registry and service requestor. This integration has of three operations: publish, find and bind. The service provider has an implementation of service. Provider defines a service description and publishes it to a service requestor or service registry. The service requestor uses a find operation to retrieve the service description locally or from the service registry and uses the service description to bind with the service provider and invoke or interact with the Web service implementation. Figure 1 illustrates these service roles and their operations.

[8] Remote procedure call

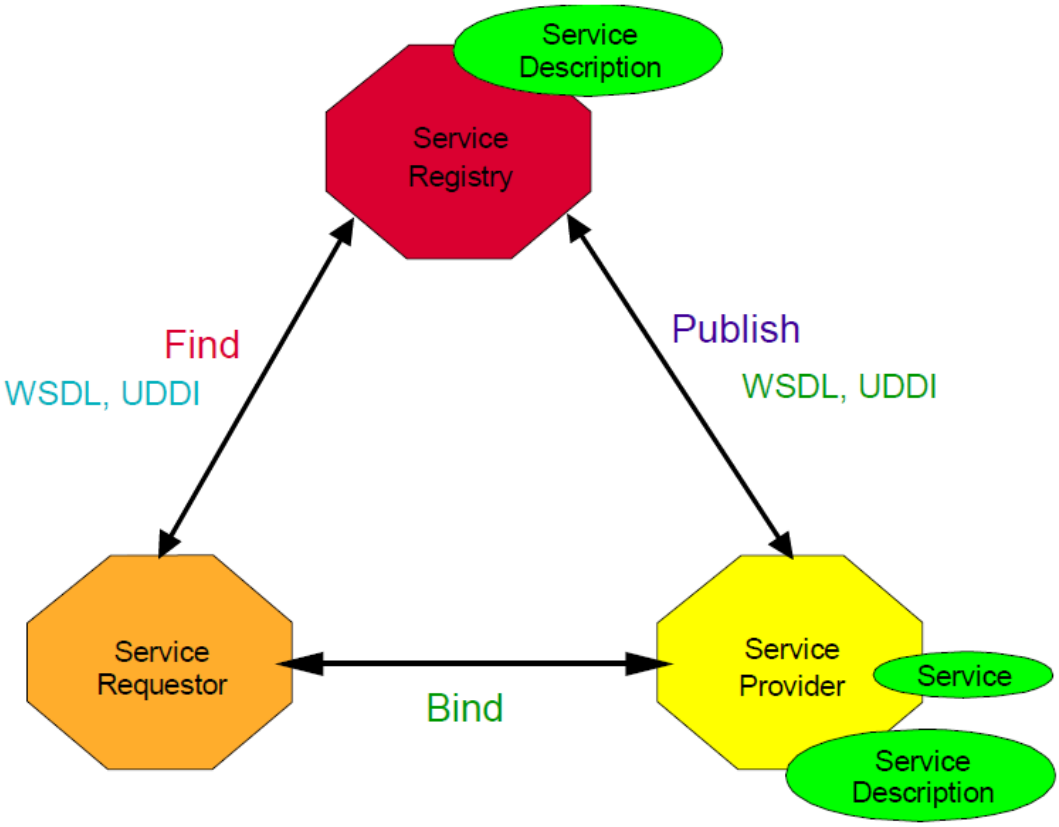


Figure 1: Web Services roles, operations and artifacts [4]

Service registry is a place where service providers can publish descriptions of their services. Service requestors can find service descriptions and get binding information from them. Binding can be static and dynamic. Registry is needed more for dynamic binding where client can get service info at the runtime, extract necessary functional methods and execute them on the server. During static binding service description may be directly delivered to the client at the development phase, for example using usual file, FTP[9] server, Web site, email or any other file transfer protocol. There are also available special protocols, named Service discovery protocols (SDP), that allow automatic detection of devices and services on a network. One of them is the UDDI[10] protocol, which is also was mentioned on Figure 1. UDDI is shortly described in Web Services Protocol Stack section.

Artifacts of a Web Service Web service consists of two parts [4]:

- **Service Description** The service description contains the details of the interface and implementation of the service. This includes its data types, operations, binding information and network location. There could also be a categorization and other metadata about service discovery and utilization. It may contain some Quality of service (QoS) requirements.
- **Service** This is the implementation of a service - a software module deployed on network accessible platforms provided by the service provider. Service may also be a client of other services. Implementation details are encapsulated inside a service, and client does not know the details how server processes his request.

2.2.2 Web Services Protocol Stack

WS architecture uses many layered and interrelated technologies. Figure 2 provides one illustration of some of these technology families.

[9] File Transfer Protocol
[10] Universal Description, Discovery and Integration

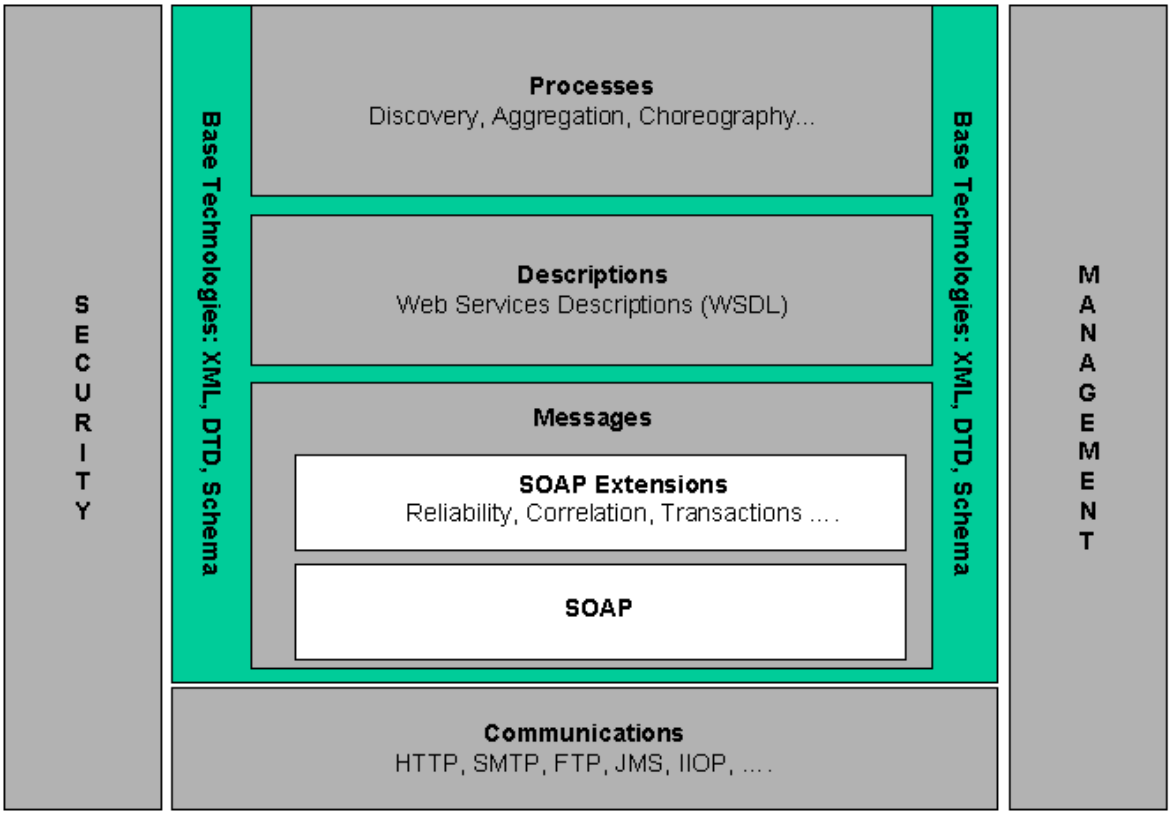


Figure 2: Web Services Architecture Stack [5]

We can describe these different layers as follows:

- **Communications** - This layer represents a transport between communication parties(service provider, client, service registry). This layer can be any network protocol like: HTTP, FTP, SMTP[11] or any other suitable transport protocol. If Web service is used in the Internet, the transport protocol in most cases will be HTTP. In internal networks there is the opportunity to agree upon the use of alternative network technologies.
- **Messages** - In order to communicate with a service, client should send a message. Messages are XML documents with different structure. SOAP protocol defines how these messages should be structured. SOAP is implementation independent and may be composed using any programming language. Protocol specification and message descriptions can be found in document SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)[6].
- **Descriptions** - This layer contains the definition of service interface (see also section 2.2.1). Web Services use WSDL language for describing the functionality offered by a service. WSDL file is the contract of service, which contains information about how the service can be called, what parameters it expects, and what data structures it returns. It is similar to method signatures in different programming languages.
- **Processes** - This part contains specifications and protocols about how service could be published and discovered. Web services are meaningful only if potential users may find information sufficient to permit their execution. Service as a software module has its own lifecycle, it needs to be deployed and deleted somehow. Traditional Web Services use UDDI mechanism to register and locate web service applications. UDDI was originally proposed as a core Web service standard.
- **Security** - Threats to Web services include threats to the host system, the application and the entire network infrastructure. To solve problems like authentication, role-based access control, message level security there is need for a range of XML-based security mechanisms.

[11] Simple Mail Transfer Protocol

Web services architecture uses WS-Security[12] protocol to solve security problems. This protocol specifies how SOAP messages may be secured.

- **Management** - Web service management tasks are [5]: monitoring, controlling, reporting of service state information.

  Web services architecture uses WS-Management [8] protocol for management of entities such as PCs, servers, devices, Web services and other applications. WS-Management has ability to discover the presence of management resources, control of individual management resources, subsrcibe to events on resources, execute specific management methods.

  WS-Management was created by DMTF(Distributed Management Task Forse, Inc., http://www.dmtf.org/) organization, which is creating stantards for managing the enterprise level systems. Organisation members are largest hardware and sortware corporations like Broadcom, Cisco, Fujitsu, Hewlett-Packard, IBM, Intel, Microsoft, Oracle. DTMF standards promote multi-vendor interoperability, which is great for the integration between different IT systems.

Most of mentioned protocols are recommended by W3C Consortium and are production standards. Lots of SOA information systems use WS-* protocols for enterprise level services. Mostly these protocols are based on XML and SOAP. One example of such protocol is the WSDL service description.

2.2.3 Service Description and Service Contract

WSDL file is an XML document which has specification of service contract. As it was mentioned earlier(section 2.2.1) contract should be shipped with a component and should tell the client what input does service expect, and what output it will produce if specified input conditions are met. Contract may be a primary specification and it should be enough for a client to start using a service. This is similar to library header file in C language. You have a ready and compiled library shipped with a header file, where are all method declarations and definitions of data structures. If header file is verbose enough, there is no need to use the documentation. You can place this component into your system very easily.

Regular WSDL document contains some necessary elements [9, 10]: Service, Endpoint, Binding, Interface and Message Types. Table 1 describes them in details.

Document may also contain optional element named *documentation*. There may be human readable service documentation, with purpose and use of the service, the meanings of all messages, constraints their use, and the sequence in which operations should be invoked. To be documentation more complete, you may specify an external link to any additional documentation.

You can see the example of WSDL service contract in the appendix Appendix A. This example is from the official WSDL standard [9]. It describes a hotel reservation service, where you can book you a room in a fictional hotel named GreatH. For simplicity it describes only one method - the *opCheckAvailability* operation. This description is quite verbose to understand what it is about. There is input and output object type declaration. It also has an output error response declaration and if some error occures during client request processing, server should send a message of specified kind. These XML object types are declared in different *namespaces* (see xmlns:* declarations at the start of WSDL document). This gives an ability to group domain types into one separate file and your main description file would not be overcrowded.

WSDL definition of the service does not contain any additional information about service hosting company and its products. At the moment when you get a service contract, you already know what company provides this service and what for this service was made. There is assumption that service provider somehow gave you this service contract. Another possibility to get the service description is to use a special *directory* or catalog, where you can find all information about company you are dealing with. Web Services architecture include UDDI mechanism for that particular purpose.

Official UDDI Version 3.0.2 Technical Specification draft [11] defines UDDI as follows:

> *The focus of Universal Description Discovery Integration (UDDI) is the definition of a set of services supporting the description and discovery of (1) businesses, organizations, and other Web services providers, (2) the Web services they make available, and (3) the*

[12]WS-Security (Web Services Security, short WSS) is an extension to SOAP to apply security to web services. It is a member of the WS-* family of web service specifications and was published by OASIS(Organization for the Advancement of Structured Information Standards, https://www.oasis-open.org/). [7]

[13]XML schema is the XML document, that specifies structure of other XML document and describes data types and constraints, that other document might have. You can create a schema for necessary XML data structure and verify if processed message corresponds to schema you have already defined

WSDL 2.0 Term	Description
Service	The service element describes *where* to access the service. A WSDL 2.0 service specifies a single interface that the service will support, and a list of endpoint locations where that service can be accessed.
Endpoint	Defines the address or connection point to a Web service. It is typically represented by a simple HTTP URL string. Each endpoint must also reference a previously defined binding to indicate what protocols and transmission formats are to be used at that endpoint.
Binding	Specifies concrete message format and transmission protocol details for an interface, and must supply such details for every operation and fault in the interface.
Interface	Defines a Web service, the operations that can be performed, and the messages that are used to perform the operation. Defines the abstract interface of a Web service as a set of abstract *operations*, each operation representing a simple interaction between the client and the service. Each operation specifies the types of messages that the service can send or receive as part of that operation. Each operation also specifies a message exchange *pattern* that indicates the sequence in which the associated messages are to be transmitted between the parties.
Message Types	The types element describes the kinds of messages that the service will send and receive. The XML Schema[13] language (also known as XSD) is used (inline or referenced) for this purpose.

Table 1: Objects in WSDL 2.0 [9, 10]

technical interfaces which may be used to access those services. Based on a common set of industry standards, including HTTP, XML, XML Schema, and SOAP, UDDI provides an interoperable, foundational infrastructure for a Web services-based software environment for both publicly available services and services only exposed internally within an organization.

UDDI mechanism uses SOAP messages for client-server communication. Service provider publishes the WSDL to UDDI registry and client can find this service by sending messages to the registry (see also Figure 1). UDDI specification defines the communication protocol between UDDI registry and other parties.

UDDI registry is a storage directory for various service contracts, where lots of companies hold their service descriptions. WSDL contracts may also be published on a company website using direct link to the WSDL file, but UDDI contains them all in one place with the ability to search and filter. You have a choice and there is a possibility to find most suitable service from all provided companies and services.

2.2.4 Advantages and disadvantages of WS-* standards

Usage of WS-* technologies gives some benefits [**?**]:

- Reusability
- Interoperability and Portability
- Standardized Protocols
- Automatic Discovery
- Security

WS-* uses the HTTP protocol as transport medium for exchanging messages between web services. SOAP messages can be transfered using another protocol (SMTP, TCP[14], or JMS[15]), which can be more suitable to your system environment than HTTP.

[14] Transmission Control Protocol
[15] Java Message Service

One another fundamental characteristic of web services is the service description and contract design. Contract specification gives you the ability to reuse service functionality in many different and separate applications. Contract in Web Services is general standardized description of a service in universal data format (XML and WSDL), that is platform and programming language independent. Service description is only the interface and the implementation of that interface may be unknown by the service client. There is possibility to transparently change (totally or partially) implementation details of a service.

The main reason why Web Services standards are bad in context of embedded systems is the performance. Web services impose additional overhead on the server since they require the server to parse the XML data in the request. Web Services use SOAP messages, which are structured XML data, for client-server communication. Some experiments [12, 13] show that performance of SOAP transfer is more than 5 times slower compared to others SOA implementations, like CORBA [16] or custom made protocol messages. If you start reading WS-* standards one by one, you will ensure that all they are interconnected using idea of XML, SOAP and HTTP.

Another statement against Web Services is that these standards are too complicated and their documentation is hard to understand. Document [15] contains a use case survey about two most common implementations of SOA: Web Services and REST[17]. Most of people in the survey agreed that WS-* standards is not easy to learn and adapt. WS-* suits better for highly integrated business solutions, but not for simple applications, with atomic functionality.

WS-* standards rely on each other and to implement a small web service with few features you will need to dive into all WS standards. This is at least 783 pages of not just text, but a techical specifications [16]. Surely, Web services have good ideas, but this technology is promoted and developed by large corporations like Microsoft, IBM, Oracle, who are not intrested in simple and lightweight solutions, because they need to utilize their thousands of developers and earn money(document [16] says that most of WS-* specifications are hosted by Microsoft and OASIS organisation. The foundational sponsors of OASIS are IBM and Microsoft). It seems that Web services are trying to solve every business problem and there is a *"WS-problem"* standard for it.

This topic described Web Service architecture features. There are lots of useful principles like portability, interface description and message exchange patterns, but the WS-* implementation is not suitable for resource-constrained hardware.

The REST has some advantages over WS-*. Next section will shortly describe the main principles of REST approach.

2.3 REST and RESTful services

2.3.1 What is the REST?

Representational state transfer (REST) is a software design model for distributed systems [17]. This term was introduced in 2000 in the doctoral dissertation of Roy Fielding, one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification. REST uses a stateless, client-server, cacheable communications over HTTP protocol. Its original feature is to work by using simple HTTP to make calls between machines instead of choosing more complex mechanisms such as CORBA, RPC or SOAP.

REST-style architectures conventionally consist of clients and servers. Clients make requests to servers, servers process requests and return responses. Requests and responses are built around the transfer of *representations* of *resources*. Author defines the *resource* as the key abstraction of information in REST [17]. It can be any information that can be named and addressed: documents, images, non-virtual physical systems and services. A representation of a resource is typically a document that captures the current or intended state of a resource.

Restful applications use HTTP requests to change a state of resource [18]: post data to create and/or update resource, read data (e.g., make queries) to get current state of resource, and delete data to delete existing resource.

REST does not offer security features, encryption, session management, QoS guarantees, etc. But these can be added by building on top of HTTP, for example username/password tokens are often used for encryption, REST can be used on top of HTTPS (secure sockets)[**?**].

[16] The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together (i.e., it supports multiple platforms).[14]

[17] Representational state transfer

[18] all four CRUD(Create/Read/Update/Delete) operations)

2.3.2 Key principles of REST

REST is a set of principles that define how Web standards, such as HTTP and URI[19]s, are supposed to be used.

The five key principles of REST are[18]:

- Give every "thing" an ID
- Link things together
- Use standard methods
- Resources with multiple representations
- Communicate statelessly

Give every "thing" an ID

Every resource need to be reachable and identifiable. You need to access it somehow, therefore you need an identificator for the resource. World Wide Web uses URI identificators for that purpuse. Resource URI could look like:

```
http://example.com/customers/1234
http://example.com/orders/2007/10/776654
http://example.com/products/4554
http://example.com/processes/salary-increase-234
http://example.com/orders/2007/11
http://example.com/products?color=green
```

Listing 1: Resource identificator examples [18]

URIs identify resources in a global namespace. This means that this identificator should be unique and there should not be another same URI. This URI may reflect a defined customer, order or product and it might correspond to database entry. The last two examples of listing 1 identify more than one thing. They identify a collection of objects, which is the object itself and requires an identificator.

Link things together

Previous principle introduced an unigue global identificator for the resource. Resource URI gives possibility to access the resource from different locations and applications. Resources can be also linked to each other. Listing 2 shows such scheme. Representation of an order contains the information about this order and linked product and client resources. This approach gives client an opportunity to change a state of client application by following linked resources. After receiving order information client has two possibilities for choice: to get product information or to fetch customer details.

```
<order self='http://example.com/orders/2007/10/776654' >
   <amount>23</amount>
   <product ref='http://example.com/products/4554' />
   <customer ref='http://example.com/customers/1234' />
</order>
```

Listing 2: Example of linked resources[18]

The idea of links is a core principle of the Web [20]

[19] Uniform Resource Identifier

[20] The World Wide Web (abbreviated as WWW or W3,[3] commonly known as the web), is a system of interlinked hypertext documents accessed via the Internet.[19]

Use standard methods

There should be a standard inteface for accessing the resource object. REST relies on HTTP protocol, which has definitions of some standard request methods: GET, PUT, POST and DELETE. Table 2 describes standard actions on resource.

Resource	GET	PUT	POST	DELETE
Collection URI, such as `http://example.com/resources`	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.	**Delete** the entire collection.
Element URI, such as `http://example.com/resources/item17`	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	**Replace** the addressed member of the collection, or if it doesn't exist, **create** it.	Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry in it.	**Delete** the addressed member of the collection.

Table 2: RESTful web API HTTP methods [20]

All four CRUD[21] operations may be done with these HTTP methods. It is possible to manage a whole lifecycle using only these methods. Client of the service should only implement the default application protocol (HTTP) in correct way.

Resources with multiple representations

When client gets representation of a resource using GET method he does not know which data format will server return. REST as architectural method does not provide any special standard for resource representation. How can client and server could make an agreement about data format they will use?

HTTP protocol helps to solve these problems again. It has a special field in the message header that defines the operating parameters of an HTTP transaction. Field *Accept* in the header of HTTP request message specifies content type that is acceptable for the response [21].

Such request header could look like this:

```
GET /images/567 HTTP/1.1
Host: example.com
Accept: image/jpeg
```

Listing 3: Request for a representation of resource in a particular format

This means that client expects that representation of a resource having identificator `http://www.example.com/images/567` should be in *image/jpeg* format. Both client and server should be aware of such format, whole system may be designed around any special format. There could be also an another representation of same resouce(the same image), for example *image/png* or *image/bmp* formats, that server could send according to received request.

Not only outgoing data format could be specified. Server can also consume data in specific formats (there are different specific header fields for this, for example *Content-Type*).

Using multiple representations of resources helps to connect more possible clients to the system.

[21] Create, read, update and delete

Communicate statelessly

Stateless communication helps to design more scalable systems. RESTful server does not keep any communication context. Each incoming request is new for the server and there should be enough information to necessary to understand the request[17].

Server could contain all the information about each connected client, but it requires a meaningful amount of resources. Server need to keep and control the current state of application for every client, which locks the server resources while client is not active (opened connections, memory, data integrity locks).

There is no need for keeping using all these resources if the application state is on the client side. Client controls the flow of the application and changes its state by making requests to server. Server does not make any work while clients are not sending requests, it starts to work only on demand.

You can easilly switch between different servers if there is no any client context on the server side. Imagine a system with some amount of application servers and load balancer server in it. All application servers run the same application. While some servers are making hard work, another servers may be idle. Load balancer have a posibility to route new incoming requests to a server, which has a smallest workload, because of the absence of application context between client and server. Even parallel request of the same client could be handled by different servers. Such system become more scalable and new server nodes can be easilly added or removed.

The main disadvantage of such approach is the decrease of network performance. Client needs to repeat sending the same session data on every new request, because that context data cannot be stored on the server.

2.3.3 Implementation constraints

The REST architectural style can be described by the following six constraints applied to the elements in this architecture[17]:

1. **Client-Server** Separation of concerns is the principle behind the client-server constraints. User interface is separated from data storage and has improved portability. Server side does not aware of client application logic and server tasks may be more optimized and independent.

2. **Stateless** This constraint reflects a design trade-off of keeping session information about each client on the server. Stateless method does not keep any application context on the server and allows to build more scalable server components. Each new request contains enough information to process it, but such technique increases network traffic by sending repeating session invormation over the network again and again.

3. **Cache** This constraint improves network efficiency. The data in the server response may be marked as cacheable or non-cacheable. Client is able to store cachable data on its side and reuse it, if it needs to send the same request again. Frequently changed data should not be marked as cacheable in order to provide data integrity.

4. **Uniform Interface** System becomes more universal if the interface of all components is the same. You can add new and replace existing componens more easilly. Component implementations are decoupled from the services they provide, system components are more independent.

5. **Layered System** style allows an architecture to be composed of hierarchical layers, that separate knowledge between components from different layers. Components in each layer do not know the structure of a whole system, but can only communicate to each other and with neighbour layers through a specified interface. Layers can be used to encapsulate legacy services and to protect new services from legacy clients. Structures inside a layer may be transparently changed. Figure 3 shows such a complex layered system.

   The primary drawback of layered systems is that they add overhead and latency to the processing of data. Every additional layer requires new amount of resources (which could be shared using common access, but layers are separated and they don't) and reduce communication performance by introducing new bottleneck at the layer boundaries.

6. **Code-On-Demand** This is an optiona constraint. REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. Not all features on

a client side may be implemented. Your system could download execution instructions from the server. This is how JavaScript and Java applets work in the Web browser.

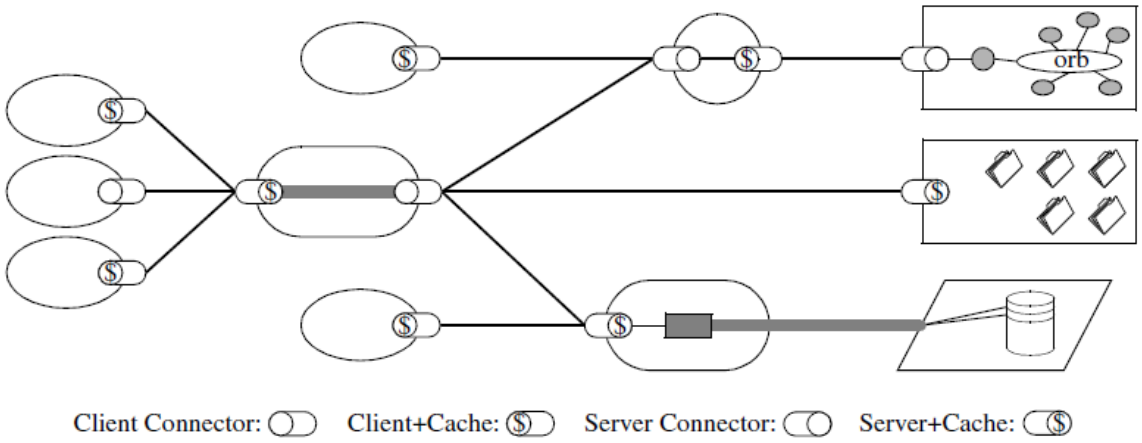


Figure 3: Uniform-Layered-Client-Cache-Stateless-Server [17]

2.3.4 Summary

Notion of independence was mentioned above in this section quite many times. This is because REST itself is a high-level style that could be implemented using any different technology and your favourite programming language. It was initially described in the context of HTTP, but it is not limited to that protocol. You can use all your creativity in a system design and you are only limited with a small amount of abstract constraints. RESTful applications maximize the use of the existing, well-defined interface and other built-in capabilities provided by the chosen network protocol, and minimize the addition of new application-specific features on top of it[18]. Therefore RESTful Web services seamlessly integrate with the Web and are straightforward and simple way of achieving a global network of smart things.

2.4 Remote procedure calls and *-RPC

Many distributed systems are based on explicit message exchange between processes. If you see a list of SOA technologies (provided above, see subsection 2.1) you can find that many of these technologies use RPC within them. Some of them don't, for example REST described in previous section, it uses different resource oriented approach(resources which the client can consume). Other majority of technologies are message oriented and use messages for IPC[22]. In RPC messages are sent between client and server to call mathods and receive results.

2.4.1 RPC in details

Remote procedure calls have become a de facto standard for communication in distributed systems[22]. The popularity of the model is due to its apparent simplicity. This section gives a brief introduction to RPC and the problems in there.

Only one figure is enough to describe RPC(see Figure 4).

When a process on client machine calls a procedure on server machine, the calling process on client is suspended, and execution of the called procedure takes place on server. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing at all is visible to the programmer. Programmer operates only with method calls.

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In other words, we want RPC to be transparent—the calling procedure should not be aware that the called procedure is executing on a different machine or vice versa[22]. To achieve this

[22] Inter-process communication

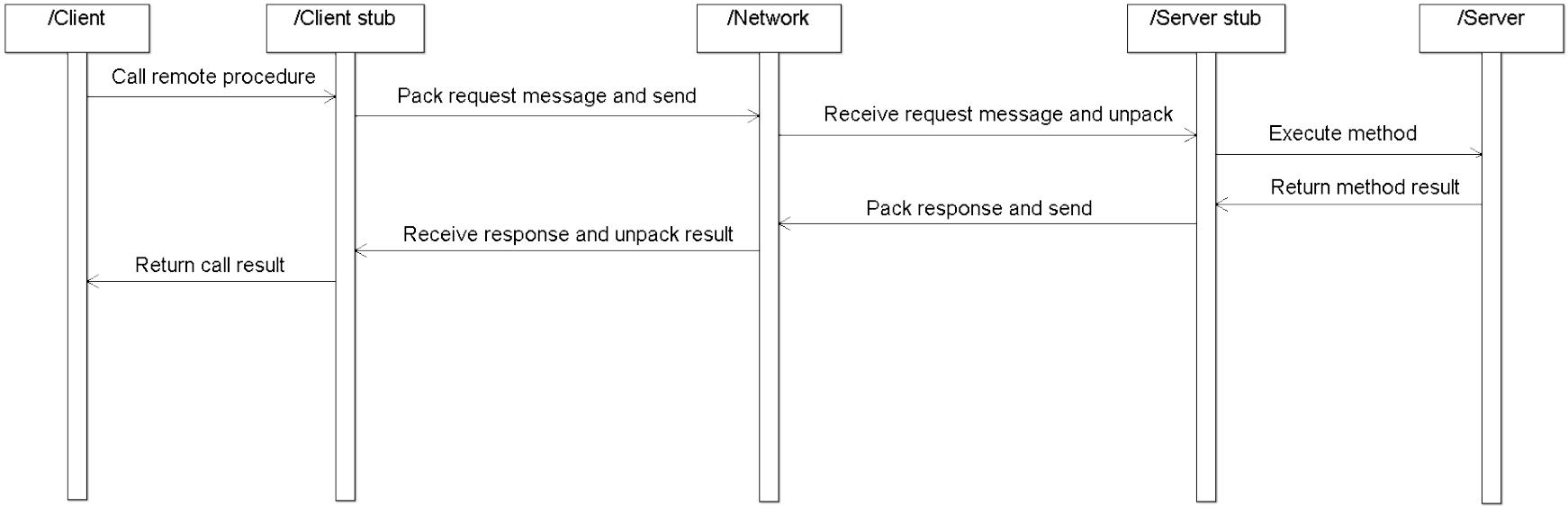


Figure 4: Principle of RPC between a client and server program

transparency special software modules are used. They are called **stubs**. The main purpose of a stub is to handle network messages between client and server.

Whole RPC call process could be described using words like this:

1. The client procedure calls the client stub on the client machine.
2. The client stub builds a message and sends it over the network to remote machine using local operating system(OS).
3. The remote OS receives the message from the network and gives this message to the server stub. Server stub unpacks the parameters and calls the server.
4. The server does the work and returns the result to the server stub.
5. The server stub packs result in a message and sends it to client using network and underlying OS.
6. The client's OS gives the message to the client stub.The stub unpacks the result and returns to the client.
7. Client receives procedure result and continues his processing.

Modern software tools help to make this process very easy. Here is the real world example of the small RPC system(written using Python programming language) that proves this[23]:

```
import xmlrpclib
from SimpleXMLRPCServer import SimpleXMLRPCServer

def is_even(n):
    return n%2 == 0

server = SimpleXMLRPCServer(("example.com", 8000))
print "Listening on port 8000..."
server.register_function(is_even, "is_even")
server.serve_forever()
```

Listing 4: RPC server example (Python and xmlrpclib)

Code is quite verbose[23] and people, who are not familiar with Python, could understand it. You create a server using the special RPC server implementation from *xmlrpclib* library. You specify a remote host and a port number in a object constructor. When server object is created you need to specify remote methods, which may be executed. Example above uses small even check method. Server registers the methods and starts to wait for incoming calls.

Client implementation for the corresponding server looks like:

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://example.com:8000/")
print "3 is even: %s" % str(proxy.is_even(3))
print "100 is even: %s" % str(proxy.is_even(100))
```

Listing 5: RPC client example (Python and xmlrpclib)

Actually, it is more shorter and simpler than server code. You just specify a remote server object and receive a proxy object. Then you can use this proxy like a usual local object. This creates

[23]Python programming language has its own philosophy, called The Zen of Python. It has some statements how software should be designed. Two statements in the Zen of Python that are related to this example are:

Simple is better than complex.

and

Readability counts.

an illusion that you are not going anywhere for the result and working with a usual objects in the code. Proxy object may be passed as a parameter to a function and be used in that function without knowledge where it was came from.

This idea is simple and elegant, but there are exist some problems. First of all, calling and called procedures run on different machines and are executed in different address spaces, which introduce additional complexity in passing parameters and results between client and server. Finally, both machines can independently crash, therefore special error handling mechanism is required. System developers must deal with such failures without knowing about remote procedure was actually invoked or not.

As long as the client and server machines are identical and all the parameters and results are scalar types, such as integers, characters, and Booleans, this model works fine. However, in a large distributed system, it is common that multiple machine types are present. Each machine often has its own representation for numbers, characters, and other data items[22].

There are several representations of character data used in computer systems: one byte characters(ASCII[24], EBCDIC[25]), multibyte characters(UTF-8, UTF-8, UTF-32[26],) and characters in different encodings(all tree character encodings are used for cyrillic symbols: Windows-1251, Code page 855, ISO/IEC 8859-5). Each RPC client and server should agree about character encoding they will use.

In addition to that, problems can occur with the representation of integers (sign-and-magnitude method, one's complement, two's complement) and real data types(floating-point, fixed-point, binary-coded decimal and single precision, half precision, double precision). Some machines have different endianness[27]. If two different machines, one little-endian and other big-endian, are communicating to each other, they should use common endianness, otherwise they will accept bytes in wrong order and data will be invalid.

This was a description of primitive data types in RPC so far, but client and server not always send primitive data types to each other. There could also be a complex data structures, that contains several primitives like characters, numbers or just raw bytes. Client and server should be aware of structure of messages they send and receive. Usually these structures are specified by Interface Definition Language(IDL). IDL is a is a specification language used to describe a software component's interface. IDLs describe an interface in a language-independent way, enabling communication between software components that do not share same language and platform. An interface is firstly specified in an IDL and then compiled into a client stub and a server stub. RPC-based middleware systems offer an IDL to support application development[22].

In most cases communication scheme is well known and some standard message protocol is used. Previous example of RPC system, that was written using Python programming language, used XML-RPC protocol. XML-RPC defines XML data types that are used in RPC messages. There are several alternative common used schemes, that provide similar functionality. They all could be divided into two groups: platform and programming language dependent and systems that can be used in multiplatform and multilanguage environment. XML-RPC belongs to second group, the similar technologies are JSON-RPC, SOAP and CORBA. Actually, most RPC implementations in the first group do not require any special harware. They are programming language dependent and may be used with the assumption, that components in the system are written using that specific language. Some examples are provided below:

- Java Remote Method Invocation
- Pyro object-oriented form of RPC for Python.
- Windows Communication Foundation (.NET framework)
- ...

Most of these programming languages have multiplatform implementations and RPC system may be built using various hardware.

JSON-RPC version 2.0 will be used in this work as primary RPC solution in the implementation of embedded client-server architecture. It is based on JSON[28] object serialization (see 2.5.2

[24] American Standard Code for Information Interchange

[25] Extended Binary Coded Decimal Interchange Code (from IBM)

[26] Universal Character Set Transformation Format

[27] The terms little-endian and big-endian refer to the way in which a word of data is stored into sequential bytes of memory. The first byte of a sequence may store either the least significant byte of a word (little-endian) or the most significant byte of a word (big-endian). Endianness refers to how bytes and 16-bit half-words map to 32-bit words in the system memory. [24]

[28] JavaScript Object Notation

section to see information about JSON or visit http://www.json.org/) and uses JSON format for transferring messages between client and the server.

JSON-RPC is a lightweight RPC solution that is designed to be simple [25]. It defines how client and server should communicate between each other. To make a RPC call client needs to send Request object, server need to response with a Response object.

The specification [25] defines Request object like a stucture with following members:

- **jsonrpc** – A String specifying the version of the JSON-RPC protocol. This must be exactly "2.0".
- **method** – A String containing the name of the method to be invoked.
- **params** – A Structured value that holds the parameter values to be used during the invocation of the method. This member may be omitted.
- **id** – An identifier established by the Client that must contain a String, Number, or NULL value if included. If it is not included the request is assumed to be a notification.

A Notification is a Request object without an "id" member. A Request object that is a Notification means the Client's lack of interest in the corresponding Response object. The Server must not reply to a Notification.

Response object is a JSON object with similar to Request members:

- **jsonrpc** – Has the same meaning as in Request object
- **result** – This member is required on success. It must not exist if there was an error in method invocation. This member returns a method result value. It might be complex JSON object.
- **error** – This member is required on error. This should be a defined object (see below)
- **id** – Identification of a client. This member value should be the same, which was sent previously by the client. It may be NULL in case of error

When there is an error in the RPC call, the Response Object must contain the error member with a value that is a Object with the following members:

- **code** – A Number that indicates the error type that occurred.
- **message** – A String providing a short description of the error.
- **data** – A Primitive or Structured value that contains additional information about the error. This may be omitted.

To give a visual information about these objects and data structures listing 6 contains examples of client server communication.

```
Legend:
--> data sent to Server
<-- data sent to Client

rpc call with positional parameters:
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

--> {"jsonrpc": "2.0", "method": "subtract", "params": [23, 42], "id": 2}
<-- {"jsonrpc": "2.0", "result": -19, "id": 2}

rpc call with named parameters:
--> {"jsonrpc": "2.0", "method": "subtract",
     "params": {"subtrahend": 23, "minuend": 42}, "id": 3}
<-- {"jsonrpc": "2.0", "result": 19, "id": 3}

--> {"jsonrpc": "2.0", "method": "subtract",
     "params": {"minuend": 42, "subtrahend": 23}, "id": 4}
<-- {"jsonrpc": "2.0", "result": 19, "id": 4}

a Notification:
--> {"jsonrpc": "2.0", "method": "update", "params": [1,2,3,4,5]}
--> {"jsonrpc": "2.0", "method": "foobar"}

rpc call of non-existent method:
--> {"jsonrpc": "2.0", "method": "foobar", "id": "1"}
<-- {"jsonrpc": "2.0",
     "error": {"code": -32601, "message": "Method not found"},
     "id": "1"}

rpc call with invalid JSON:
--> {"jsonrpc": "2.0", "method": "foobar, "params": "bar", "baz]
<-- {"jsonrpc": "2.0",
     "error": {"code": -32700, "message": "Parse error"},
     "id": null}

rpc call with invalid Request object:
--> {"jsonrpc": "2.0", "method": 1, "params": "bar"}
<-- {"jsonrpc": "2.0",
     "error": {"code": -32600, "message": "Invalid Request"},
     "id": null}
```

Listing 6: JSON-RPC examples of client-server communication [25]

JSON-RPC is more simple when it is compared with XML-RPC. It is more compact and transport independent [29]. XML vs JSON section provides more detailed comparison of JSON and XML

2.4.2 RPC summary

This section described one another posible way of IPC. RPC is used in many distributed systems in obvious or imlicit way. It provides mechanism for calling subroutines or procedures on another computer or system. Communication in RPC is based on message passing. The Client sends to the server a message containing request for method call. The Server sends to the client a message with procedure results. Modern software tools provide ready RPC libraries and application programmer has no need to explicitly reinvent whole RPC system from scratch. You can build various distributed systems using RPC.

[29] XML-RPC specification at http://xmlrpc.scripting.com/default.html tells that it uses HTTP as the main transport

2.5 Data serialization

Serialization is a process for converting a data structure or object into a format that can be transmitted through a wire, or stored somewhere for later use [26].

Previous sections described some possible implementations of Service Oriented Architecture. These technologies use client-server communication and send information between client and server, that need to be understood at both destinations. No matter how this information is sent, using resource/object representation in case of REST or request/response message in case of RPC, it needs to be converted to format that can be decoded and understood with the user of that information. Common transmission scenario can look like:

1. Client wants to send a some information to a server. It has some data in memory and that data is in application specific format(object structure, text, image, movie file).
2. Client **packs** his information into a message and sends it to server using any possible transport channel(email, paper mail, homing pigeon, tcp socket, etc).
3. Server receives this message, **unpacks** the message and gets a piece of information that client wanted to send.
4. Server reads the information and decides what to do with it.

Process of packing information is called **serialization** (also deflating or marshalling) and process of unpacking is called **deserialization** (inflating or unmarshalling).

There are lots of different ways and formats that can be used. Which method and format to choose depends on the requirements set up on the object or data, and the use for the serialization (sending or storing). The choice may also affect the size of the serialized data as well as serialization/deserialization performance in terms of processing time and memory usage[26]. Next section describes possible serialization solutions.

2.5.1 Serialization technologies

Serialization is supported by many programming languages, which provide tools and libraries for data serialization to different formats. Article [27] provides a summary of well known data serialization formats. Most of them could be divided into two groups: human-readable text based formats and binary formats. Both groups have their own advantages and disadvantages. Table 3 shortly describes them.

Property	**Binary formats**	**Human-readable formats**
Format examples	Protocol Buffers from Google Apache Thrift(TBinaryProtocol) BSON used in MongoDB database MessagePack(http://msgpack.org) and most of native serialization mechanisms in various programming languages(Java, Python, .NET framework, C++ BOOST serialization)	XML JSON YAML
Advantages	The two main reasons why binary formats are usually proposed are for **size** and **speed**. Typically use fewer CPU cycles and require less memory. Binary data is transformed as is, no need to encode data bytes(image, video, etc) Better for larger datasets Random data access.	Do not have to write extra tools to debug the input and output; you can open the serialized output with a text editor to see if it looks right. Self-descriptive and easily recoverable. No need to use programming issues like sizeof and little-endian vs. big-endian. Platform and programming language independent. Broad support by tools and libraries

Continued on next page

Table 3 – continued from previous page

Property	Binary formats	Human-readable formats
Disadvantages	Not verbose. Hard to debug. Fixed data structures. Hard to extend. Not self-descriptive(it is hard to understand for human where actual data starts in the array of bytes,), has no data description(metadata, layout of the data) Require special software and highly customized data access algorithms. Hard to recover data after software version change (remember different MS office formats)	Binary data needs to be converted to text form(Base64). Additional processing overhead (CPU and memory consumption). Lot of redundancy. Representing your data as text is not always easy/possible or reasonable(video/audio streams, large matrices with numbers)

Table 3: Comparison of binary and human-readable serialization formats

Text-based nature makes human-readable format a suitable choice for applications where humans are expected to see the data, such as in document editing or where debugging information is needed. Binary formats are better for high speed and low latency applications.

2.5.2 XML vs JSON

Choosing the right serialization format mostly depends on your data and application. But if there is no any constraint what protocol to use, text protocols are more preferable. They give you advantages like: verbosity, extensibility, portability.

Most popular human and machine readable serialization formats are XML and JSON.

XML

> *XML is hugely important. Dr Charles Goldfarb, who was personally involved in its invention, claims it to be "the holy grail of computing, solving the problem of universal data interchange between dissimilar systems." It is also a handy format for everything from configuration files to data and documents of almost any type. [28]*

The fundamental design considerations of XML include simplicity and human readability[29]. W3C[30] specifies the design goals for XML like [30]:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML[31].
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness in XML markup is of minimal importance.

[30] World Wide Web Consortium

The primary uses for XML are Remote Procedure Calls and object serialization for transfer of data between applications.

Simple data structure in XML looks like:

```
<person>
        <firstname>John</firstname>
        <surname>Smith</surname>
        <email>john.smith@example.com</email>
        <mobile>1234567890</mobile>
</person>
```

Listing 7: XML structure describing abstract person

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format[32].It is easy for humans to read and write and also is easy for machines to parse and generate. JSON is based on JavaScript Programming Language and is directly supported inside JavaScript. It has library bindings for popular programming languages.

JSON is built on two structures[32]:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

The same structure from XML section looks in JSON encoding like:

```
{
        "person" : {
                "firstname" : "John",
                "surname" : "Smith",
                "email" : "john.smith@example.com",
                "mobile" : 1234567890
        }
}
```

Listing 8: JSON structure describing abstract person

JSON value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested.

JSON specification is quite easy and it is described using only one page [32]. This page provides also a list of programming languages and libraries that support JSON.

Research in different documents [33, 29] showed that that JSON is faster and uses fewer resources than its XML counterpart. Transferring data in the form of JSON instead of XML can speed up the server data transfer efficiency.

This is because XML is characterised by its rich verbosity, meaning that it requires a separate start-tag and end-tag for describing the content. As JSON does not use end-tags at all for the description of the content, the resulting number of bytes is smaller. Paper [33] contains performance analysis of mobile device(Apple iPhone), which executes different tests working with XML, JSON and SOAP. Research results prove that SOAP and XML have overhead over JSON.

To make messages more smaller binary formats or compression(for example gzip) should be used. There is also available newly emerged Efficient XML Interchange (EXI)[32] standard for binary XML encoding. It is expected to become an alternative to XML for exchanging data between embedded systems[34].

[31] Standard Generalized Markup Language. SGML is a system for defining markup languages. Authors mark up their documents by representing structural, presentational, and semantic information alongside content. [31]

[32] It was adopted as a Candidate Recommendation by the W3C, for more information see http://www.w3.org/TR/2013/CR-exi-profile-20130416/

Parsers
All JSON and XML parsers can be divided into two groups: stream-based and tree-based. Stream based parsers (also known as Simple API for XML(SAX)) are event-based. They read input message sequentially and signals the application when a new component has been read. They raise notifications on reaching different document parts and programmer needs to decide what to do with this part of document(store or skip). Such parsers require less resources than tree-based, because they do not need to keep whole document in memory.

Tree-based parsers load whole message to memory and then parse it. They create a tree of this document (also known as Document Object Model (DOM)) and return it to application developer. Programmer receives whole document tree and is able to extract needed parts from that.

Although stream-based parsers often have better performance than tree-based parsers, they make application code more complex due to their event-driven nature.

Document serializers work using similar scheme.

2.5.3 Is there a right serialization format?

There is no direct answer for that question. The format and the parser used has to be chosen according your application constraints and requirements. Binary formats give you a small message size and reduce required amount of CPU cycles for data processing, but there is a lack of verbosity and it is hard to extract data without using additional tools. Text based formats reduce development and debugging time, but they have overhead because of their verbosity. There is a hardware resources/ development time tradeoff in this problem. Some calculations should be made before making a decision.

Whatever format you choose, binary or human-readable, this should be a standard and open format. It means that this format should be supported by different software tools and programming languages, you can read its specification and understand its features, your people can study it more quickly, system integration becomes more easy, because other system understand it too. If it is a production standard, there is a chance that it will not essentially change in near future.

Text formats like XML and JSON are user all over the world and are good candidates to be in your system.

2.6 SOA and Embedded Systems

Resent topics had a review of available tools and technologies for implementing a SOA system. Most of them are not directly portable to embedded systems. Related publications [13, 15] claim that SOA tools need to be adapted to a constrained hardware by using more lightweight approaches. Common possibilities are: use of more resource friendly protocols, use of existing service protocols in a contstrained manner(low request per second ratio or smaller payload/packet size), use of special constrained protocols, which are designed specially for interaction between small devices.

This section will describe two possibilities of implementing SOA on an embedded device, that are based on different research papers[35, 1]. Devices Profile for Web Services section will introduce a Web Services based device mmcounication framework. Constrained Application Protocol and Constrained RESTful Environments section will cover RESTful device interactions.

2.6.1 Devices Profile for Web Services

The Devices Profile for Web Services (DPWS) was developed to enable secure Web service capabilities on resource-constrained devices[36]. DPWS was mainly developed by Microsoft and some printer device manufacturers. DPWS allows sending secure messages to and from Web services, discovering a Web service dynamically, describing a Web service, subscribing to, and receiving events from a Web service.

Figure 5 shows DPWS protocol stack. It is based on several Web Services specifications[36]:

- WS-Addressing for advanced endpoint and message addressing
- WS-Policy for policy exchange
- WS-Security for managing security

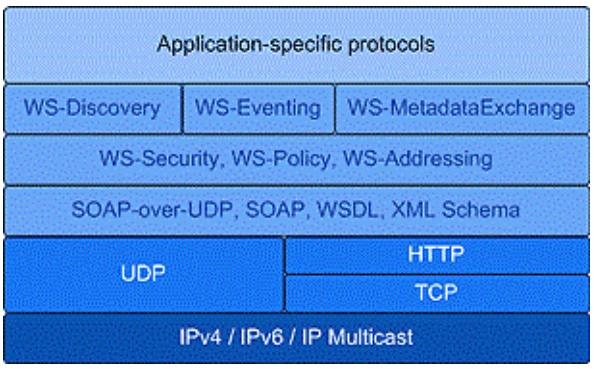


Figure 5: DPWS protocol stack [36]

- WS-Discovery and SOAP-over-UDP for device discovery
- WS-Transfer / WS- Metadataexchange for device and service description
- WS-Eventing for managing subscriptions for event channels

Like Web Services, DPWS uses SOAP, WSDL, XML-Schema.

DPWS has been ported to several target software platforms (Linux, Microsoft's Windows and Windows CE, ExpressLogic's ThreadX and Quadros Systems' Quadros). It was tested on a processor board comprising a 44-MHz ARM7 TDMI and associated memory (but no cache memory), running ThreadX. The static memory footprint of the device software including the OS, the TCP/IP protocol stack and the DPWS software is less than 500 KB, while the dynamic memory requirements are below 100 KB.[1]. Another research [13] reports that system disk space requirements are between 61 and 478 Kbytes.

2.6.2 Constrained Application Protocol and Constrained RESTful Environments

Constrained Application Protocol (CoAP) is a software protocol is used in very simple electronics devices that allows them to communicate interactively over the Internet. It is particularly targeted for small low power sensors, switches, valves and similar components that need to be controlled or supervised remotely, through standard Internet networks.

Lots of applications over the Interent use REST architecture. The Constrained RESTful Environments (CoRE) research in IETF organization aims to realize the REST architecture in a suitable form for the most constrained nodes (e.g. 8-bit microcontrollers with limited RAM and ROM) and networks (e.g. 6LoWPAN, IPv6 over Low power Wireless Personal Area Networks. RFC4944) [37].

One of the main goals of CoAP is to design a generic web protocol for the special requirements of this constrained environment, especially considering energy, building automation and other machine-to-machine (M2M) applications. The purpose of CoAP is to implement a subset of REST coupled with HTTP, but optimized for M2M applications. Although CoAP could be used instead of HTTP interfaces because of more compact protocol, it also offers features for M2M such as built-in discovery, multicast support and asynchronous message exchanges.

CoAP has the following main features:

- Constrained web protocol fulfilling M2M requirements.
- UDP [RFC0768] binding with optional reliability supporting unicast and multicast requests.
- Asynchronous message exchanges.
- Low header overhead and parsing complexity.
- URI and Content-type support.
- Simple proxy and caching capabilities.
- A stateless HTTP mapping, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way or for HTTP simple interfaces to be realized alternatively over CoAP.
- Security binding to Datagram Transport Layer Security (DTLS) [RFC6347]

CoAP messages of two types: requests and responses. CoAP uses a short fixed-length binary header (4 bytes) that may be followed by compact binary options and a payload. CoAP is by default bound to UDP and optionally to DTLS, providing a high level of communications security [38].

2.6.3 Performance issues

Web services implementations on embedded devices have quite big overhead. Web service messages are at least 5 times larger than conventional messages(messages using usual data stuctures, not XML), messages take at least 2.5 to 2 times longer than conventional messages, consume more 2 times more energy[13]. Therefore this technology can be only used on hardware with high computational and power posibilities, not for deeply embedded devices like low-cost microcontrollers for wireless sensors and actuators.

The second techology, CoAP and RESTful services requires less resources. Its highly optimized implementation could be executed on a device with 8.5 kB of ROM and 1.5 kB of RAM [39]. It uses Contiki OS and TCP/IP implementations. This architecture could be a great candidate for a remote embedded service.

CoAP was specially designed for constrained devices, while DPWS is a profile (may be also called a port) of Web services techology to devices, with complexity of all WS-* technologies. DPWS require more powerful hardware and is not suitable for the devices where CoAP was designed to run.

2.7 Final target system requirements

All previous sections introduced ideas how various devices can communicate to each other. Each described technology has its own pros and cons. In general, we need to choose a technology without any drawbacks or a technology, which is the smallest evil chosen from list of suitable ones.

Web Services have these advantages:

- Service description
- Service discovery
- Portability and platform independence(XML)
- Standardized protocols and message structures.
- Ability to transfer messages using other transport protocol than HTTP

The RESTful approach enables to model our domain objects as consistent, hierarchical URLs with predictable operations for CRUD(GET, POST, PUT, DELETE). It is also based on HTTP that comes with standard error codes, message types(see also Resources with multiple representations) and generally does most of the transport hard work, so we benefit from not needing to maintain any user-developed protocol and using ready and well defined technologies. One of the main concepts of REST is that RESTful services are stateless and do not store session information. This reduces resource consumption and makes client-server applications more loosely coupled and scalable.

REST and Web services have different ideas. REST is based on resources and their representation, while Web Services use messages to send data and call remote methods between server and client. This technology is based on very simple idea of RPC. RPC has some essential benefits over REST and WS-* technologies: it does not require traditional transport like HTTP, TCP, Ethernet or Wi-fi. It can be implemented using any radio link, serial line or any other suitable transport, that is able to deliver response and request messages. Web Services in theory could also be transport independent, but most WS-* tools assume the HTTP(and underlying technologies) as de facto standard [33]. Support of other underlying protocols for SOAP(the main messaging protocol in WS-*) should be implemented separately and most standard tools does not have this, they just use standard HTTP. WS-* has a huge overhead in doing simple things like implementing small light controlling service in your room. This is not right tool for that. Simple RPC would be enough.

Table 4 defines system requirements and features for service prototype in this work. These are ideas that were kept in mind while developing of embedded service system was in progress.

[33] SOAP protocol transport independence started from SOAP 1.2 [6]

Feature	Description
Transport independent solution	Company did not defined final communication yet. This may be one of these: UART, Bluetooth, RF, HTTP. In this work i implement only first prototype of such system and requirements are about to be changed in future. Therefore i need to implement a portable solution that could be quickly ported to another environment.
Service description and service contract	Gives overview of all service capabilities. Provides an interface definition language. Is a specification(sometimes may be only available one) to the server and server interface.
RPC based communication	Our prototype needs to control coffee machine and execute some requested actions. If we had an Ethernet, the design decision would be REST(with all its benefits and philosophy). Otherwise RPC is the most suitable solution here. This system is message oriented. Some standard RPC solution should be used or ported to the embedded device.
Lightweight and verbose messages	Microcontroller has limited resources. We cannot use huge messages for really small request data [34]. Some text based protocol need to be used. Binary protocols are evil, it is real hard to understand and debug them.
Simple design	Simple is better than complex. Not like WS-*
Modular architecture	System should be divided into separate functional modules. It should be easy to change the internal implementation of these modules, without a need for global refactoring.

Table 4: Ideas about embedded service

[34] internet resource http://www.simple-is-better.org/rpc/ contains a comparison of different message formats)

3 Implementation

This section will cover the implementation of embedded service server and other parts of the system. This system is only a small part of whole service infrastructure, it does not cover discovery, addressing, authentication and security and other essential parts of every production service. This prototype only includes an embedded server and a client application to demonstrate how technologies and methods already implemented in "big computer systems" can be adapted to "small" embedded and resource-constrained devices. Implementing a full application stack of service technologies (for example WS-* or) needs a lot of human and time resources. You can read some standards(amount of pages was already mentioned above in section Advantages and disadvantages of WS-* standards) and count how many human*hours it would take to implement this in embedded system environment with limited amount of resources, low-level application programming using C language and without ready made and off-the-shelf software tools and libraries for that kind of systems. In my opinion, one master student is unable to create a complete server solution only by himself within reasonable time. The scope of this work requires some more resources, a team with several members maybe, to accomplish such task.

The solution that is possible to implement during university project like this is the research about related field and available technologies and a simple system prototype. I have implemented this using collected features from literature and web resources.

First section below covers the general architecture of implemented service. Next come details about embedded server, which contatain the description about hardware and software platform used, program architecture and data flow. There is also an implementation of the client side application library (also called client stub) below. The last section here introduces one possible client application for this service architecture.

3.1 System architecture and device connection scheme

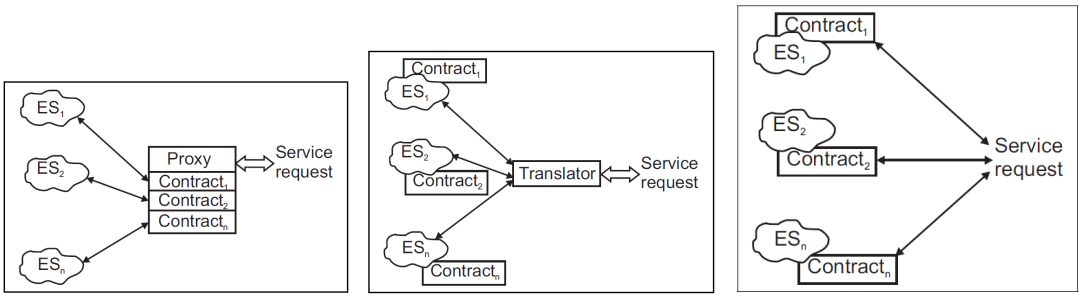
Devices in a system may be interconnected in various ways. Some embedded systems **do not have** any connections at all. Such systems only sense or control the environment and there is no need to send data somewhere. These are usually highly embedded devices with limited amount of functions (alarm display controller, microwave or washing machine controller).

Another group of devices are systems that are able to interchange information using **proprietary** communication methods at **physical and logical** layers. These systems can send information to another systems, but they are using non standard protocols for data transmission.

Third group uses standard (serial line, ethernet with ip protocol) communication techniques at the physical and transport level and some **proprietary logical** protocol above that.

Last group can integrate with all other systems and has **full communication** possibilities. These systems use standard application protocols and transfer data over well known channels.

Research paper [2] introduces three architectures , that cover three possible ways of connectivity between devices in previously mentioned groups: proxy, translator and full architecture.



(a) Proxy approach provides physical and logical conversion

(b) Translator approach provides logical conversion

(c) Full approach where embedded systems directly provide services for the environment

Figure 6: Possible connection architectures for the embedded services

Proxy is a device which is between the client and the embedded service. It provides services to the client and in the same time can communicate with embedded system using closed protocols. Proxy device stores service contracts onboard and know all specifications of connected embedded systems.

Translator approach is similar to the Proxy, but it covers devices with proprietary logical communication. The underlying physical transport is common to client and service provider. The main purpose of Translator is to convert messages that come from clients into into a logical format that the embedded system can understand. Service contracts are stored inside services on the other embedded systems, not on Translator. Translator may not be a separate device and it can be only a software module.

In the Full architecture client and service provider can directly connect to each other without need of any device in the middle.

Our coffee machine system use closed proprietary protocol inside, but all communication messages are transferred over standard serial line. This is more similar to Translator approach, but there is one problem in implementing such architecture. We cannot directly store service contract inside coffee machine system. Coffee machine internal architecture and implementation does not allow us to store any additional code for implementing service functionality. Internal processor is utilized enough and there are no resources for anything else except controlling coffee machine. In addition, the company did not provided to me a specification of internal communication mechanism . There are several microcontrollers inside that are controlling different machine parts. They provided only the external interface communication protocol to me, therefore my implementation is more similar to Proxy approach, where contracts are stored inside Proxy machine and Proxy is a separate physical device. Figure 7 shows the general architecture of created system.

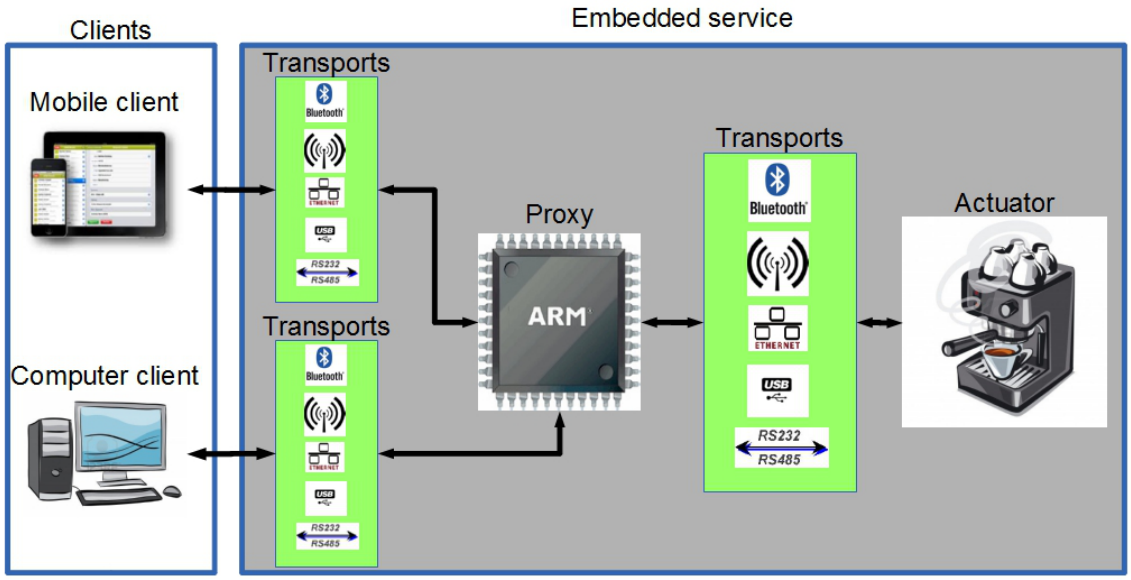


Figure 7: General system architecture

This is a traditional client-server approach where clients (mobile or desktop) send the requests to the server(Proxy embedded device) over some network and physical transport(Bluetooth, various radio frequency connections, ethernet, USB, serial line, . . .). Proxy server is connected to the controlled device (marked on the figure as Actuator), which is the coffee machine in this example application.

There can be different clients and proxy can control and monitor various devices, but connection scheme $Client \leftrightarrow Transport \leftrightarrow Proxy \leftrightarrow Transport \leftrightarrow Controlled\ or\ monitored\ device$ is essential.

In this application mobile clients are connected through Bluetooth wireless. Each client may be connected using supported by the proxy device transport. The proxy is connected to coffee machine by wires and serial line.

Next section covers the internals of Proxy device.

3.2 Implementation of the embedded server

This server can be implemented using any suitable hardware and software. Chosen tools and libraries are not fixed and can be easily changed in future. Loosely coupled modules in the system give a possibility to change everything without a need of global redesign.

The reason why technologies below are used is that they are simple to use and easy to learn. They are also quite lightweight and therefore they can be used in a embedded system.

3.2.1 Hardware

Figure 8: STM32F10X 128K evaluation board (STM3210B-EVAL) [40]

The hardware used in this work are the two ARM Cortex M3 microcontrollers from STMicroelectronics(http://www.st.com/). They were chosen because the company already has a development board and some other products from that manufacturer. There is nothing special in that hardware and similar microcontrollers from other manufacturers may be used in the same way.

The hardware features desribed below are common for almost all microcontrollers and every well known manufacturer has similar device family.

Description will start from the first used STM32 microcontroller and the STM3210B-EVAL evaluation board from STMicroelectronics. These are features that this board has [40]:

- Three 5V power supply options: power jack, USB connector or daughter board
- Boot from user Flash, test Flash or SRAM
- Audio play and record
- 64Mbyte MicroSD card

- Type A and Type B smartcard support
- 8Mbyte serial Flash
- I2C/SMBus compatible serial interface temperature sensor
- Two RS232 communication channels with support for RTS/CTS handshake on one channel
- IrDA transceiver
- USB 2.0 full speed connection
- CAN 2.0A/B compliant connection
- Induction motor control connector
- JTAG, SWD and trace tool support
- 240x320 TFT color LCD
- Joystick with 4-direction control and selector
- Reset, wakeup, tamper and user push buttons
- 4 LEDs
- RTC with backup battery
- Extension connector for daughter board or wrapping board

As you see there are lots of opportunities to apply your creativity. The amount of features is quite big, but we do not need most of them. Required are only connectivity(RS232, USB) and debug(JTAG, SWD) interfaces.

This development board is made for evaluation of STM32F10x family microcontrollers. These are ARM Cortex-M3 core-based mainstream microcontrollers with a maximum CPU speed of 72 MHz and Flash memory amount from 16 Kbytes to 1 Mbyte. They are equipped with large variety of peripherals. Figure 9 covers interfaces of STM32F103VBT6 MCU that is used in this board.

This controller is equipped with 20KB SRAM and 128KB Flash memory. It has three USART transceivers and the debugging interface. These are the main features we need.

The second microcontroller that was used is the STM32F103ZE MCU. It has more memory: 64KB SRAM and 512KB Flash memory. This controller was chosen because the first one has not enough resources, especially SRAM memory. The text messages, that are transferred from client to server and back, require to be stored in the RAM memory and each request has several copies of data while it is processed. Amount of RAM on the first MCU was enough to execute optimized and final version of the server, but during development there is need for storing additional debug information and to try different libraries. The second reason is that service contracts are stored in the flash memory. STM32F103VBT6 has 128 Kbytes of flash, however STM32F103ZE has 512 Kbytes. System becomes more simple when service contract is stored together with application code and there is no need to introduce another level of complexity (connecting external storage device and programming connectivity code for that).

STM32F103ZE has 5 USART interfaces which is more than enough for this kind of system. Three of them are used in the application: One for client-server communication, one for logging and the third one for communicating with coffee machine.

Client and server are connected using Bluetooth-to-serial module LMX9838 from Texas Instruments.This module contains hardware and firmware support of Bluetooth and Serial Port Profile and can be used as simple wireless serial interface in communication between devices.

Server logging interface is connected to a personal computer using FTDI Serial-to-USB chip (http://www.ftdichip.com/). This is popular solution of connecting embedded systems and USB powered hardware. There is the virtual com port on the PC side, that is powered by drivers of operating system. This solution can be used instead of old serial connectors, that are not always available on modern hardware.

Embedded service server and coffee machine are connected using serial line and wires. This is a most simple connection here. Coffee machine has external serial interface and STM32 microcontroller is connected directly to it.

That was a short overview of system hardware. Next comes description of system software and operating system.

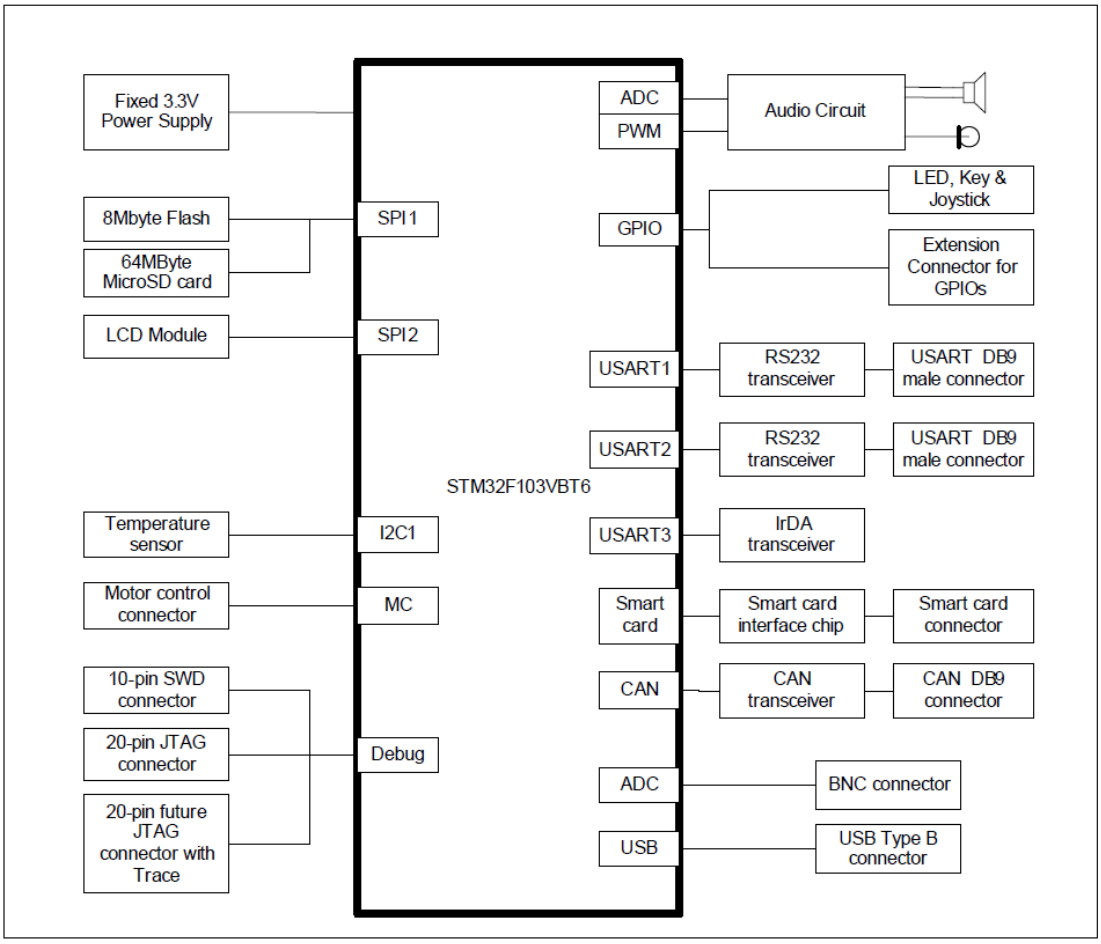


Figure 9: Hardware block diagram of STM32F103VBT6 MCU on STM3210B-EVAL board [40]

3.2.2 Software and operating system

Hardware configuration
All MCU hardware is controlled by the software. The execution of a trivial program (like "Hello world") on a microcontroller requires a long list of instructions. To write some bytes to UART or to blink with LED you usually need to:

1. Configure MCU clock. Select clock source, frequency, clock prescalers
2. Configure clock for peripheral buses. Again, the source and frequency by configuring prescalers. (Advanced Peripheral Bus in case of ARM)
3. Turn on clocking for buses and peripherals.
4. Configure general purpose input/output (GPIO) ports to use required function.
5. Configure interrupt controller for the peripherals.
6. In case of UART or any other communication, set the communication speed and configure the interface parameters.
7. Write your application code
8. Download the code to the device
9. Debug the results
10. Start from the beginning.

Modern MCUs, especially with ARM Cortex-M architecture, have very complex structure. They contain a huge amount of interfaces (see 3.2.1 section and feature list of evaluation board) in a one

single chip. Most of peripherals are separately clocked, which gives a possibility to turn off inactive ones and save the power energy. During MCU system startup programmers code should turn on and configure required peripheral modules.

In contrast, traditional software developing for desktop computers requires only last four steps and the most complicated preparation step is the compiler and IDE environment setup.

Each "configuring" step requires deep knowledge about what you are doing. MCU is configured by values that are stored in memory mapped control registers. Each register has a special purpose and different values written there configure various MCU modules. In general words, you need to know these values to configure the MCU system properly. Each bit in the control register represent some setup setting.

Every controller family from various manufacturers differs from each other. Therefore there is no need to write here in deep details how to configure one special MCU named STM32F103ZE from STM32F1, what values to write into `USART1->CR1` control register and which register bit turns on the UART parity check. These instructions are not portable. When MCU hardware changes(even families from a single manufacturer may highly differ) you need to write this part once more.

There are two possible and more portable ways how to configure a STM32 microcontroller: Using CMSIS[35] from ARM and Standard Peripheral Library from STMicroelectronics. These are hardware abstraction layer libraries that help to write portable code. They define standard application programming interface (API) for the ARM architecture. Hardware vendors provide CMSIS compliant bindings for their hardware and programmers may write their code in standard manner, reducing resources for platform change and reusing existing code.

Standard Peripheral Library is a C language library that provides standard API for programming STM microcontrollers. It also contains CMSIS inside for managing ARM core. There are several data structures, macros and functions that help to configure and manage MCU peripherals. The famous UART may be configured using Standard Peripheral Library like this:

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);

USART_InitTypeDef USART_InitStructure;
USART_InitStructure.USART_BaudRate = 115200;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_No;
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
USART_Init(USART1, &USART_InitStructure);
```

Listing 9: USART1 initialization using Standard Peripheral Library

Most of configuration can be done using preprocessor defines. You may add several defines, for example `#define STM32F10X_MD` to use STM32 Medium density devices (Flash memory density ranges between 64 and 128 Kbytes). Clock rates and different peripherals can be adjusted in a similar way.

The distribution of STM Standard Peripheral Library contains a lot of code and application examples, that helped me a lot. I have found there how to configure a USART DMA[36] controller and how to transfer bytes using USART without intervention of central processing unit. This is quite good source for the beginners like me.

Setup an embedded development environment takes a lot of time and requires deep knowledge about the underlying hardware. It is good to have such tools like standard APIs, that help to make this process more easy.

Operating system

Programs for embedded devices can be written in many different ways. Some traditional ways how to do it:

- One big while loop and polling resources(also called busy waiting)

[35] The ARM® Cortex™ Microcontroller Software Interface Standard (CMSIS) is a vendor-independent hardware abstraction layer for the Cortex-M processor series.

[36] Direct Memory Access

- The same while loop, but it is interrupt driven. Interrupts change some global variables inside interrupt service routine (ISR) and main loop checks them on the next program cycle
- The use of multitasking realtime operating system (RTOS)

The two first methods are often called bare metal approach. Bare metal refers to techniques where programmers directly control and manipulate the underlying core, performing register-level access setting and clearing bits, and moving and operating on bytes of data directly. Everything traces directly to the programmer's next line of code being executed, and there is nothing in between the programmer and the hardware. Resources like timers, interrupts, and I/O have to be accounted for in the application code. The bare metal approach works well when a system has a well defined and deterministic processing algorithm, only a couple of tasks to manage, or needs precise instruction level determinism(hard realtime).

Operating systems for microcontrollers have features like: multitasking/multithreading,task prioritization, interprocess communication, memory management, abstracted I/O drivers, file systems, networking. They give to programmer higher level of abstraction and standard APIs, which help to reuse already existing code. Popular RTOS already come with various protocol stack and drivers implementations. However, the greatest feature they have is the multitasking. The program could be divided into several separate tasks and these tasks could run in the parallel. Parallel execution is achieved by dividing available CPU time between tasks. This process is called *scheduling*. There are two main scheduling policies: cooperative scheduling and preemptive scheduling.

During cooperative scheduling CPU time is used by a task until this task explicitly gives CPU time to other tasks. This means that if the task will not give this time to others the whole system will hang. This approach gives reliable and deterministic control over all tasks and programmer needs to define CPU time release points. There is no need of protection of common resources, because each task controls its execution and another task could run in that moment.

The second preemptive scheduling approach gives each task a regular "slice" of operating time. Each task has defined amount of CPU time during which it is able to do useful work. Scheduler takes this time when this time is over and gives this to another task. These tasks could have priorities, which allows to execute essential tasks before the background are executed. At the moment of the task context switch scheduled decides which task will be next according to the priority of available tasks. Task with a higher priority will be triggered next. Scheduler should also be able to give the execution time to tasks with lower priorities, otherwise the will be never triggered because tasks with higher priorities will be launched instead. Tasks with the same priorities are executed one by one in the loop[41]. Such scheduling algorithm gives the illusion that tasks are running in parallel. Parallel task execution requires protection of common and global hardware resources. For example, the situation when some task can change a global variable, while another task is reading it and makes some decision. The result is that second task receives invalid result, which was modified between assigning a check value to some variable and reading it.

The coffee machine service application contains many different task like: multiple communication handling, request processing, request deserialization, etc. This amount of tasks is not final. This system needs to be extensible. There should be an opportunity to add new request and communication handlers. Request processing by the server is the concurrent process: while several requests are processed, another requests are sent over the network and communication interface needs to receive them. Response transmitting could also run in parallel.

It becomes harder to maintain and develop such concurrent program if you are not using a multitasking system. You need always to carefully think where to insert a functionality into a big polling loop. Each insertion requires reordering of control statements and checking the execution order of all parallel tasks. Moreover, your application may handle two different independent tasks and you need to keep data separated and remember which task does each variable belong. The use of operating system helps to keep things separated and loosely coupled.

FreeRTOS

FreeRTOS is a popular real-time operating system for embedded devices, which has ports for many MCU devices(34 architectures [42]), also including STM32F1 family microcontrollers. This operating system was chosen for the embedded service application implementation in this work. The reason is that this operating system has low entry level for the beginners and good documentation. In addition to that i have found and read a book [41], which is a great introduction to FreeRTOS and embedded multitask systems programming.

FreeRTOS has several features described below[42]:

- Pre-emptive scheduling option

- Co-operative scheduling option
- ROMable
- 6K to 10K ROM footprint
- Configurable / scalable
- Compiler agnostic
- Some ports never completely disable interrupts
- Easy to use message passing
- Round robin with time slicing
- Mutexes with priority inheritance
- Recursive mutexes
- Binary and counting semaphores
- Very efficient software timers
- Easy to use API

All this features and support of available hardware helped to make a choice of using FreeRTOS in this project. I was inspired by this easy to use API, spread documentation and verbose examples. Next i will describe some essential FreeRTOS APIs and how tasks are created and managed.

The next thing you need to make after the setup of the programming environment and writing a trivial program to test it is the definition of system tasks. A task in FreeRTOS is usual C function and listing 10 contains a prototype of this function.

```
void ATaskFunction( void *pvParameters );
```

Listing 10: FreeRTOS task function prototype

Implementation of this function usually contains a separate program which has infinite loop and never exits. Creation of temporary tasks which are finite is also possible. Tasks may be deleted in the runtime using `vTaskDelete()` function. Task function has no return type (void) and `return` statements are not allowed, they should be explicitly deleted. It accepts method parameters of void pointer type, which allows to pass any type parameter there. Required parameter needs to be casted to `void*` before passing the parameter. It can be used inside task function through casting back to original type. This feature is used to pass a complex configuration structure to a system task in the embedded service design.

Task function do some useful application work. In order to start they require a registration in a system scheduler. When microcontroller program first starts, it gets into reset interrupt service routine, where starts the hardware initialization process. The `main()` application method is called after all hardware is initialized. All tasks should be started in that `main()` application method, where the first application code usually starts. Listing 11 shows task creation function prototype.

```
portBASE_TYPE xTaskCreate(
        pdTASK_CODE pvTaskCode,
        const signed portCHAR * const pcName,
        unsigned portSHORT usStackDepth,
        void *pvParameters,
        unsigned portBASE_TYPE uxPriority,
        xTaskHandle *pxCreatedTask
);
```

Listing 11: FreeRTOS task function prototype

The first parameter is the pointer to a task function, which has the application logic. Second parameter is the name of that task. It is used for system needs like debuging. Parameter with name `usStackDepth` configures available stack space for the task. Each context switch and method calls in a task use that memory space for storing application context. Program counter and state of the operation registers are stored there. All local variables are also copied to the stack during context switch. Therefore this parameter should depend on the complexity of the application (function call depth) and amount of local variables used in each child function. The fourth parameter is a pointer to the task parameters. Next parameter defines priority of that task. The last one is the pointer to the task handle structure, which should be defined before this method call. `xTaskCreate()` method returns a handle. This handle may be used to control this task.

This method only creates tasks, but not executes. `main()` application method should contain a `vTaskStartScheduler()` call, that executes the scheduler and registered tasks are executed by this scheduler. `vTaskStartScheduler()` call is normally a blocking call, it never ends and runs until there is some error. On error scheduler method finishes and program execution returns to the `main()` application method, where program continues to run.

Task can force a contex switch by calling `taskYIELD()` method. FreeRTOS scheduling algorithm can be configured as cooperative or preemptive. In cooperative scheduling mode `taskYIELD()` method is used to make a context switch and give CPU time other tasks. In preemptive it forces a context switch before time slice has ended. Another task can start from the middle of the time slice of previous task. This gives better overall system performance.

FreeRTOS uses queues for interprocess communication. Queues can be used to send messages between tasks, and between interrupts and tasks. In most cases they are used as thread safe FIFO (First In First Out) buffers with new data being sent to the back of the queue, although data can also be sent to the front.

The queue operations are mutually exclusive (see listing 12). The same queue element cannot be received twice in different tasks. Messages are sent through queues by copy, creation of the queue allocated memory space and queue insert function copies inserted value to that space. You hold different values in a queue: this may be plain C language primitive type, a structure or even `void*` type. Large structures can be sent by storing a structure reference pointer. Queue functions are also blocking functions. Delay amount measured in system ticks can be provided to queue functions, it is able to lock forever by passing a special system value to these functions. Operation methods also return status of the operation.

```
xQueueHandle xQueueCreate(
        unsigned portBASE_TYPE uxQueueLength,
        unsigned portBASE_TYPE uxItemSize
);

portBASE_TYPE xQueueSendToFront(
        xQueueHandle xQueue,
        const void * pvItemToQueue,
        portTickType xTicksToWait
);

portBASE_TYPE xQueueSendToBack( xQueueHandle xQueue,
        const void * pvItemToQueue,
        portTickType xTicksToWait
);

portBASE_TYPE xQueueReceive(
        xQueueHandle xQueue,
        const void * pvBuffer,
        portTickType xTicksToWait
);

portBASE_TYPE xQueuePeek(
        xQueueHandle xQueue,
        const void * pvBuffer,
        portTickType xTicksToWait
);
```

Listing 12: FreeRTOS queue methods

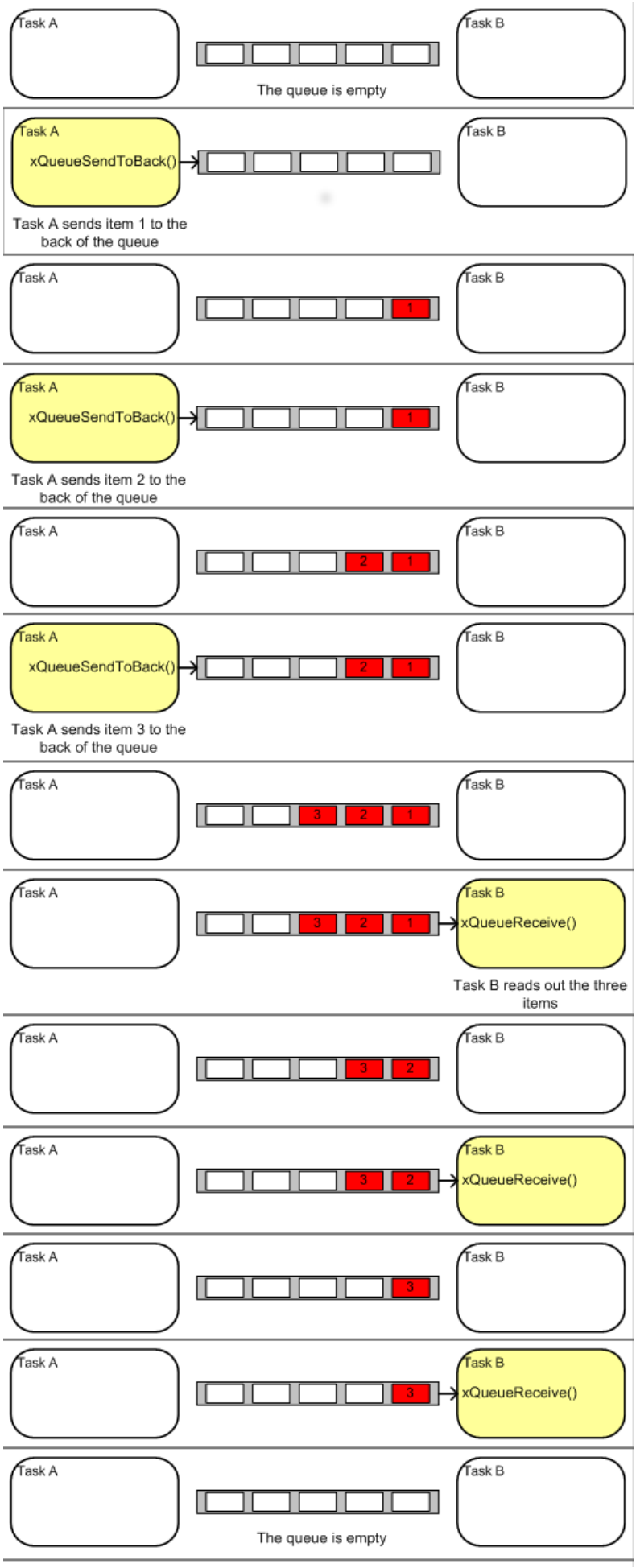


Figure 10: Writing to and reading from a queue.[42]

Semaphores and mutexes are based on the queues and their methods are similar to queue methods. They can protect your sensitive data from being changed from other tasks. `xSemaphoreTake()` and `xSemaphoreGive()` are used for that purpose. There are also available mechanism of critical sections.

FreeRTOS has own memory management mechanism and there are four memory allocation options available:

1. Memory can be allocated, but it cannot be freed. This option is for static applications, which allocate memory only at system startup and use that for the lifetime of program
2. Best fit algorithm is used. Memory can be freed. This approach does not combine free blocks into a single large block and memory gets fragmented. This is not good for applications who allocate random memory size.
3. In this option the standard C library `malloc()` and `free()` functions are used. This is a standard way of memory allocation and it is supported by the compiler and linker.
4. This scheme uses a first fit algorithm and, unlike scheme 2, does combine free memory blocks into a single large block

Options other than 3 are more efficient than standard `malloc()` and `free()` and are thread safe. Their footprint is smaller and work in pair with the OS kernel.

I tried several options and ended up with the last configuration option. Option 3 was not stable for me and system frequently crashed while using it. Therefore last approach was chosen as more stable. Memory heap is stored inside one big byte array and memory allocator takes a free space from there.

FreeRTOS may be used in various applications and it contains lots of useful functionality. It can be extended using different modules (see [42]) like networking modules (TCP and UDP), input output modules, filesystem modules and others. It is a quite good choice for a newbie like me. There is no need to reinvent the wheel.

3.2.3 Service software

Previous section was mainly about available features in the operating system and hardware. This section covers how the software of embedded server is written using some of them.

Initialization

Application starts with hardware initialization code. Microcontroller clock source is external quartz oscillator (8 Mhz) and MCU is configured to use PLL(Phase locked Loop) frequency multiplier, which is used for multiplying its input frequency by a given factor of two to sixteen. Using the PLL, you can generate clocks up to 72MHz.

Next comes initialization of USART communication interfaces. USART are configured to use DMA[37] controller. DMA works in both directions: receive and transmit. The usage of DMA requires initialization of DMA controller, assigning destination/transmitting memory buffer and buffer length, and other configuration parameters. DMA may be configured to use memory increment and data direction (from the peripheral to buffer or from buffer to peripheral). This feature enables sending and receiving data to/from the USART without utilization of CPU.

In case of transmitting the data, program may store data to the memory buffer, trigger DMA send event and continue further processing. DMA controller sends the data bytes to USART one by one until incremented memory address is equal to the length of the buffer. Interrupt is generated after operation completion (or in case of error, you need to check status flags to extract operation status), which signalizes end of transmission operation to user. While DMA is sending data, CPU can prepare next bulk of data to send.

Receiving of data works in a similar way. Three types of DMA interrupts are used for that:

- DMA Full transfer interrupt indicates that full buffer transfer was ended
- DMA Half transfer interrupt routine triggers when the half of the transmitting buffer is sent
- USART idle interrupt indicates that the line is free and there are no more characters.

[37] Direct Memory Access

When the DMA half interrupt occurs user receives first half of the receive buffer. Second half of the buffer may be taken by the user after the full transfer interrupt. Line idle interrupt service routine is triggered when the line is free for some amount of time. Programmer needs to check what is the current status of the transfer and how many bytes are already received by the DMA controller. Current position(first half or second half of the buffer) can be calculated according that value and the right chunk of bytes can be extracted.

Interrupt service routines store incoming bytes in a FIFO(First In First Out) buffer. Application note [43] contains description of this approach. In short, two receive buffers are used: one for DMA controller and another is for user application. There are some problems using traditional one buffer implementation: user software should retrieve data from the receive buffer before the data are overwritten by the next received data. Receive buffers are often implemented in a circular way(DMA also has this option) and when buffer is filled, write pointer returns to the start of the buffer and data may be written again. User needs to read this data or it will be overwritten. In a double-receive-buffer approach the second buffer, which belongs to the user, is not overwritten. The code implemented by me just ignores to write into user buffer when it is full. There is a tradeoff while user is inactive and does not read anything: to loose all the data and overwrite all buffer with new bytes or to loose only some part of it. That is how receiving of the message works in the embedded server described in this work.

After initialization of the USART transceivers and their DMA, program initializes FreeRTOS tasks and data structures. All system queues and semaphores are created at that time. Tasks are registered and task input parameters are passed into task functions. Next scheduler is started and all tasks start to run. Scheduler works forever while system is powered.

This is how successful scenario looks like. There are also another ones:

- There is unable to create tasks and data structures (not enough memory or some error)
- There is not enough memory to start all tasks and scheduler function exits and program crashes.
- A stack of some task gets overflowed and task crashes.
- There is not enough memory in a heap and task cannot allocate memory for new objects. As a result program crashes.

All this events are signalized by the FreeRTOS kernel and programmer needs to decide what to do if some of them happens. Our application is in early stage development and error conditions are handled using unlimited loops. Program gets into this loop and it is able to watch stack trace in the debugger and find a place where it was behaved incorrectly.

System tasks and their functions

The architecture of this embedded server consists of several components. Figure 11 shows how they all are connected together.

Most tasks have similar structure and may be covered with a few lines of code:

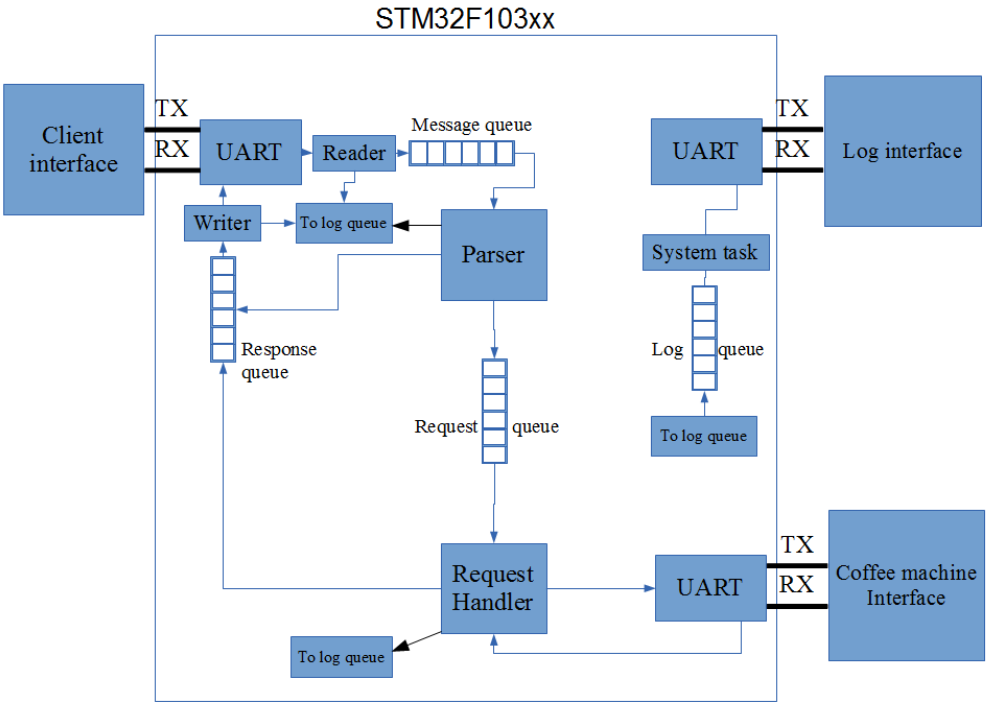


Figure 11: System tasks architecture and the data flow

```
void tskSomeTask(void *pvParameters) {
        while(1) {
                if(uxQueueMessagesWaiting(someQueue) > 0) {
                        xStatus = xQueueReceive(
                                systemMsgQueue,
                                &sysMsg,
                                QUEUE_RECEIVE_WAIT_TIMEOUT
                        );

                        /*
                                make some useful work here
                        */
                }
                vTaskDelay(250 / portTICK_RATE_MS);
        }
}
```

Listing 13: General task function structure

As it was written before, FreeRTOS tasks are made like unlimited loops. Usually tasks are waiting for some messages in a queue. When a message is inserted from somewhere task receives it from the queue and starts processing. When the work gets done, task sleeps for define amount of time.

Most of task in this system are made using this method. Tasks wait for resources and process them. They are in sleep most of time.

Task with a higher priority is the task called `tskSystem`. It is started before all other tasks and is responsible of handling system messages. `tskSystem` is used to output debug information and logging messages. It is waiting for the message with type `MSG_TYPE_LOGGING` and sends it over logging UART. This task wakes between every 250 ms intervals and checks for new logging messages. It also reports the remaining heap memory size.

Logging messages are sent to a system queue from other tasks. Logging gives the ability to trace essential processing steps and to quickly find the broken place. I have implemented logging API which is similar to Apache log4j in Java programming language(see http://logging.apache.org/log4j/1.2/ and search for *log4j* and *slf4j* keywords. In order to write log message you need to call `logger(LEVEL_INFO, "log message text")` method specifying a level of severity for a message and message text. There are lots of these calls in application code. It is necessary to log all essential moments of application lifecycle. Level with lowest severity `LEVEL_TRACE` may be used for tracking separate method calls. Each method may have trace logging call at entry and return points. User can turn of these messages by setting global logging system level to more higher one (the highest is `LEVEL_OFF`).

Other tasks will be covered according to data flow showed in Figure 11.

The first task that meets request from the client is the UART reader task. At the early state of development this was a special and dedicated task for reading from one communicating port. Later i changed it to be more common and function was renamed to `tskAbstractReader()`. Now this can be a reader for any source. It is configurable by task input configuration parameters which have structure showed in listing 14

```
typedef struct _reader_params_t {
        transport_type_t                    transport_type;
        stream_read_char_function_t        read_char_func;
        stream_has_byte_function_t         stream_has_byte;
        xQueueHandle                            dataInputQueue;
        portTickType                            dataInputQueueTimeout;
        xSemaphoreHandle                    dataReadSemaphore;
} reader_params_t;
```

Listing 14: Reader configuration structure

This task become an abstract because it has common processing algorithm (which is similar to algorithm in listing 13). It waits for `dataReadSemaphore` in a loop. This semaphore is released in DMA ISR[38] when UART message arrives. `stream_has_byte_function_t` is a function that returns true when UART incoming message FIFO has bytes to read. `stream_read_byte_function_t` returns one byte from the buffer [39].

This byte can be processed using any algorithm. This is a place where you need to define a protocol that your application will use. There are available million and one more communication protocols and this work covered some of them below. This level requires an analogue to Data link layer protocols from ISO OSI communication model (an example can be Ethernet, Point-to-Point Protocol (PPP), High-Level Data Link Control (HDLC) and others). Their purpose is to provide an defined mechanism how pieces of data bytes compose communication frames or packets. Frame is a digital data transmission unit in computer networking and telecommunication. These protocols define which data bytes are control information and which represent data to be sent.

There should be special byte sequences (or single bytes) which indicate the beginning and the end of each communication message. The first control symbol i started to use was ASCII newline symbol (’
n’). Incoming sequences of bytes were separated by a byte with value `0x0A`. The messages were line oriented and each incoming line was a new message.

There is a problem using such control mechanism. If your message contains a newline(which is very predictable if you are working with text) you receive an incomplete message and there is impossible to find if received message is wrong or not. Message gets splitted into two message packets.

I started to improve this solution and the next step was usage of ASCII control characters like EOT(end of transmission, value `0x04`), ETB (end of transmission block, value `0x17`) and similar. Using these control characters you can detect messages in incoming stream, but this does not help in case of transfering data bytes, which can be any value between `0x00` and `0xFF`.

I started a research of suitable protocols that have a solution for that problem. My mistake

[38] Interrupt service routine

[39] Most programming language have a concept named *streams* or *data input output streams*. The tutorial located at http://docs.oracle.com/javase/tutorial/essential/io/streams.html (accessed 13-August-2013) introduces this abstraction.

was that i started from wrong direction. I started to read specification for Point-to-Point Protocol and other Internet related technologies. My thoughts were about realization of a complex data packet structure, with lots of packet members and a complex data flow control in there, like it is in these technologies. My friends, who are programmers, advised me to create a custom protocol using `start_byte:length:checksum:data:end_byte` message structure. This might be that exact structure, but finally i found a solution from the beginning of my master thesis research - the website[40] of Roland Koebler, which is named *"Simpler is better"*. The title page of this site contains a beautiful quote:

> Technology always develops from the primitive, via the complicated, to the simple.
> — Antoine de Saint-Exupéry

I have found there a very simple solution how to transfer data packets. It is called *netstrings* (`http://cr.yp.to/proto/netstrings.txt` is the source specification, last accessed at 13-August-2013). A netstring is a self-delimiting encoding of a string which has very simple format: `<LENGTH>:<DATA>,`. Length is number of characters to read, colon indicates the start of the message, message is read until message length becomes equal to `<LENGTH>`, last comes message separator - the comma. netstrings method was enough to fulfill application needs. Current application requirements do not define data link protocol. This approach defines a logical message protocol and it can be used in the demonstration of service application. If the requirements change this protocol can be replaces by any other, but for now it is enough.

Let's get back to UART read task. This task contains a simple finite state machine (FSM) for parsing netstrings. When a message is received, UART read task puts the payload to the `dataInputQueue` and continues to read next messages.

The last config parameter `transport_type_t` specifies a transport, from which messages are transfered. This is a system wide enumeration constant which identifies a source of the message. This architecture allow to have many different sources of requests. The core is transport independent, it takes only messages with defined structure from message queue. The information sources can be UART lines, CAN[41], Ethernet, any wireless or wired physical transport solution. To add a new transport to the system you need to write a source reader, which understands underlying physical data stream protocol and can extract information data from that, and a writer, which is connected to response queue and can take messages from there and write them to the stream.

Messages in this system have a structure defined in listing 15 and are named *packets*.

```
typedef struct _packet_t {
        json_int_t          id;
        packet_type_t       type;
        transport_type_t    transport;

        union {
                strbuffer_t                 *stringData;
                json_t                         *jsonDoc;
        } payload;

        int locked;
} packet_t;
```

Listing 15: System packet structure. Packets are used to deliver messages between different parts of the system

Packet contains an identificator, type or the meaning of that packet and data payload. Variable `locked` is used for simple synchronization (mutex lock).

The idea of using such data structures was inspired by the Chain of Responsibility software design pattern[42].

[40] `http://www.simple-is-better.org/`

[41] Controller Area Network

[42] Description of that pattern in russian language `http://cpp-reference.ru/patterns/behavioral-patterns/chain-of-responsibility/`

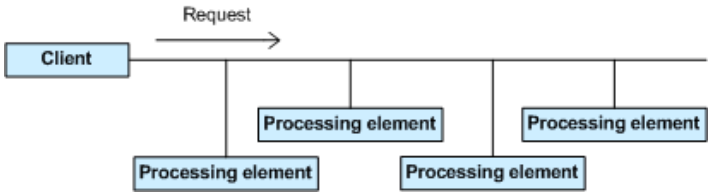


Figure 12: The idea of Chain of Responsibility software design pattern

The idea of this pattern is that client request is processed by a chain of processing units. Each processing unit takes request from previous chain member, makes processing and gives processed object to the next chain member. This chain can be easily changed by adding or removing processing units and replacing their position in the chain.

The reader needs to create a new packet, assign type `PKG_TYPE_OUTGOING_MESSAGE_STRING`, assign payload (`strbuffer_t` is a simple character buffer structure) and to put this packet to `dataInputQueue` from the task configuration.

UART writer pulls the same structured packet from the response queue, checks that packet type is `PKG_TYPE_OUTGOING_MESSAGE_STRING`, extracts the payload and sends the data back to the client. It is also responsible of cleaning the memory, it needs to destroy this packet (by calling appropriate method)

Reader gives the work to message parser task. This embedded server implementation uses JSON data structures and JSON-RPC protocol as primary message passing technology. They were covered before and some client-server communication was introduced (see listing 6). The message parser task takes string messages from message queue and converts them into JSON request objects. It receives system packet with a type `PKG_TYPE_INCOME_MESSAGE_STRING` and makes a `PKG_TYPE_INCOME_JSONRPC_REQUEST` from it. Parsed JSON-RPC requests are added to the request queue by this task.

JSON website [32] contains an overview of available JSON serialization and deserialization libraries. I have tried some of them, which claimed to be very lightweight: Jansson (`http://www.digip.org/jansson/`), parson (`http://kgabis.github.io/parson/`) and jsmn (`http://zserge.bitbucket.org/jsmn.html`). I was also tried to write my own JSON parser, but after some progress i realized, that i am writing something similar to those ready libraries. I decided to take one of them and to use in my project.

The last one, jsmn JSON parser library, is the most lightweight solution i have ever seen. It suits well for the embedded systems without dynamic memory allocation. This is a stream based parser and it does not hold copied of data, but holds only references to them. The regular usage includes declaration of static array of token structures. A token is a structure that represents JSON object. It has a type and pointers to the original character data of character JSON message. There are no separate copies stored, API provides only pointers to data. You need to define the array of tokens in the code, that will be enough to process a new message. Parser API function returns a status code that can be a success, parsing error or lack of tokens in the token array. You can process the tokens if there was a success at parsing step. Some finite state machine needs to be implemented for this. As the result, it requires more complex processing code (like stream based parser do). It is not convenient to use such API, this library requires a more beautiful wrapper.

The main choice of JSON parsing library consisted of the battle between Jansson and parson. Jansson was chosen to be used in this project, because of the custom memory allocation and more elegant API. It also has a very good documentation with examples and explanations.

```
json_t *array, *integer;

array = json_array();
integer = json_integer(42);

json_array_append(array, integer);
json_decref(integer);
```

Listing 16: Examples of using Jansson library API

This library uses the object reference counting for managing the lifecycle of the objects. The reference count is used to track whether a JSON object is still in use or not. When a value is created, it's reference count is set to 1. If a reference to a value is kept (e.g. a value is stored somewhere for later use), its reference count is incremented, and when the value is no longer needed, the reference count is decremented. When the reference count drops to zero, there are no references left, and the value can be destroyed and memory is freed. This gives a possibility to make complex nested object structures and there is no need of tracking each object separately. Complex objects can be deleted by decrementing the reference count of the root objects. Jansson library recursively decrements count of all nested objects and if reference count becomes zero object is destroyed.

Income message parsing task uses Jansson library to parse incoming JSON-RPC requests. When request object is parsed it is checked for compatibility with JSON-RPC version 2.0(checks for valid JSON object members). Request identificator (`id`) is exctracted from JSON object and assigned to the packet identificator inside parser task. The `id` is a positive integer which has `json_int_t` type (this is actually defined as `unsigned long long` inside Jansson library). It is more convenient and efficient to store and later use raw id value for the packet, than to extract the `id` from JSON object every time (which requires a hash table lookup inside member extract function, which is more expensive than reading a single variable)

The next processing unit in the request chain of responsibility is the request handler task. This is the main request handling module in this system. Listing 17 contains an algorithm description of request handler task written in C like pseudocode:

```
while(1) {
        if(requestQueueHasRequests()) {
                request = getRequestFromQueue();
                if(request && (typeof(request) == JSONRPC_REQUEST)) {
                        response = handleRequest(request);
                } else {
                        reportError();
                }
                destroyRequest(request);
                if(response) {
                        sendResponseToResponseQueue(response);
                }
        }
}
```

Listing 17: Request hadler algorithm

The request handling method contains a table of available methods and the method name from the request is searched in that table. If method was found, a function that corresponds to that particular method is called and the request is passed as the parameter to that function. The method not found error response is returned if there is no such method available. I have developed several demo methods to demonstrate coffee machine service functionality Table 5 contains description of available methods in this system.

Method name	Input parameters	Method output	Description
"system.help"	–	Full service description.	Service contract is returned by this method. This is a structured JSON document, that contains a description of available mehtods similar to this table.
"machine. getInfo"	–	Information about coffee machine	This method returns to a client a JSON object with name of connected coffee machine and machine firmware version.

Continued on next page

Table 5 – continued from previous page

Method name	Input parameters	Method output	Description
"machine. getProducts"	–	List of available products	A complete list of available products is returned. The members of that list are JSON structures that define a product. They contain the name of a product, its code and the price.
"machine. orderProduct"	A product code	Returns a status of order operation.	This method call starts coffee prepearing procedure on the coffee machine. Valid product code, that was received as a result from "machine.getProducts" should be provided. Method returns the status of operation which can be one of follows: "PRODUCT_STATUS_STARTED" "PRODUCT_STATUS_FAILED" "PRODUCT_STATUS_BUSY" The last status is returned if this product is already running.
"machine. getProductStatus"	A product code	Returns a status of previously started product order.	This method call is used for retrieving a status of coffee product currently being prepared. Valid product code, that was received as a result from "machine.getProducts" should be provided. Method returns the status of operation which can be one of follows: "PRODUCT_STATUS_STARTED" "PRODUCT_STATUS_IN_PROGRESS" "PRODUCT_STATUS_FINISHED" "PRODUCT_STATUS_FAILED"
"machine. cancelProduct"	A product code	Returns a status of cancel operation	Possible values are: "PRODUCT_STATUS_CANCELLED" "PRODUCT_STATUS_FAILED"

Table 5: Embedded service remote methods

Each handler method receives and returns an JSON-RPC message object. Some methods are communicating to a directly connected coffee machine and trigger some events there. This communication is using closed proprietary protocol and these methods are working with a special library from the manufacturer. This library was provided to me and was partially implemented by me. User code need only to execute a methods from that library and receive a result. Library internals write requests to the coffee machine and receiving response messages using UART transciever. Library specification is the C language header file, where are defined all function prototypes and return codes. The coffee machine manufacturer insisted of not publishing the details of protocol used and the implementation of that library. We are using it here in the design with the assumption, that it does its communication work well, with any knowledge of internal behaviour of that software module (black box). This was one of the main reasons why a proxy/translator architecture (mentioned here 3.1) may be used to integrate proprietary and legacy devices into other systems.

Returned response JSON-RPC messages are serialized back to string character data in the request handler task and are inserted to a response queue, where they could be picked by a UART writer task for further writing (see Figure 11). The last UART writer task writes data to the client interface UART and totally destroys that response message.

Now we have described the whole data flow cycle of this system architecture. One of the main design goals it that system packets are created and deleted only once inside Reader and Writer software modules near the corresponding client interface. Other processing units are only changing the internal packet structure by changing packet type and payload of the packet. This reduces overall memory consumption.

The architecture design that is based on queues and separate processing tasks help to produce an extensible and scalable systems. When some task becomes a throughput bottleneck there is always an opportunity to add the another same task in parallel. Two or more tasks can receive the work from one queue and more than one task could push the results into one destination queue.

Processing units in between can be changed or totally replaced, because input and output interface of the modules is defined. For example, if we decide to change the netstrings message encoding for something else, we only need to rewrite Reader and Writer modules. JSON to XML data serialization replacement requires more fundamental changes (because JSON object is used as primary data transfer objects in the request handle methods, therefore you need to rewrite all request methods to solve that problem), but if the underlying communication transport structure is not changed, it is not necessary to touch this part of the system.

3.2.4 Service description contract

As you already know, the whole system i based on the JSON-RPC specification. Service description contract is also made using similar technologies. Appendix A provides an examples of service contracts. JSON version of contract is inspired from JSON Schema Service Descriptor Draft at http://www.simple-is-better.org/. There is a specification of service interface description, which is similar to the WSDL from WS-* technologies. Service contract of coffee machine service is made according to that specification.

It uses a JSON-Schema[43] specification to describe the JSON data format. JSON object could also be self desctiptive and automatically verified by the machine. There is no official standard available yet, but most of these technologies are in the development phase right now. These are mostly working drafts. JSON-Schema is supported by the IETF(Internet Engineering Task Forse) organization and there is a big hope that these JSON technologies become an production standard.

```
{
    "description": "A geographical coordinate",
    "type": "object",
    "properties": {
        "latitude": { "type": "number" },
        "longitude": { "type": "number" }
    }
}
```

Listing 18: JSON-Schema definition of geographic coordinate object

JSON is a great and more lightweight alternative for XML. Web Services and related XML technologies become a de facto standard for large corporate service systems. There are developed by large corporations like Microsoft, IBM, Oracle who have personal interest of these technologies. These standards are more bureaucratic that simple and verbose JSON solutions.

Author of this work tries to say that service oriented technologies can be implemented not only using XML, SOAP and HTTP. There are lots of other alternatives and the implementation of correct solution depends on the target application. I needed more compact and lightweight protocol, because of the target hardware.

Provided JSON service descriptor contains the declaration of implemented methods and their input parameters structures and types of returned objects. Service contract can be received as a response of "system.help" method The communication could look like:

```
-->  {"jsonrpc": "2.0", "method": "system.help", "id": 1}
<--  {"jsonrpc": "2.0", "result": { Service contract object here}, "id": 1}
```

Listing 19: Acquisition of service contract

There could be also any defined character sequence that tells the server to response with the documentation object. Imagine that you are connecting to an old legacy device using serial line communication. You have found it in your basement within the old gadgets of your grandfather. You connect some wires, open a serial terminal and do not know what to do next. There is no protocol description nearby and the internet has only few unclear reference to the text written on the device box. You start typing some chaotic text in the terminal and send it to the device, and device does not

[43] http://json-schema.org/

respond you. After some retries are made, you receive a huge text containing the specification of this device. For example, server may respond to some of received character sequences:

```
-h
--help
{"jsonrpc": "2.0", "method": "system.help", "id": 1}
"^*[Hh][Ee][Ll][Pp]*$" regular expression values
... to be continued
```

Listing 20: Server may return service contract description after he receives these characters from communication line

PROFIT! Now you could game with the recovered device.

Service contract provide an easy and descriptive way how to search for available piece of functionality. You can call remote procedures right after you have received this kind of specification.

3.3 General purpose client library implementation

3.3.1 The technology used

This section will define a client side of embedded service application. JSON-RPC messages are used as the main application protocol. Client part should also support all these features that server offers.

Figure 4 showed the main architecture of RPC. RPC client application usually consists of two modules: client application code and client RPC stub, that communicates with remote server. Client calls usual methods on a stub and the code in a stub handles all the communication magic, it sends requests, receives responses, serialize and diserialize data objects.

There were already some examples of a small RPC application (see listings 4 and 5) Our system requirements and environment does not allow to use Python programming language. The client application in this work is written for Google Android smartphone platform, which has a JVM[44] virtual machine inside, and the Java programming language is used.

Java is very popular programming language which is based on *"Write once, run anywhere"* philosophy. This means a programmer can develop code on a PC and can expect it to run on any Java enabled hardware, from small embedded systems to huge server mainframes. This is quite mature programming language with lots of libraries and already written code that solves different problems. For example, see a list Java tools for JSON at `http://www.json.org/`.

Android system uses Java as one of the main programming language. Most programs for android are written using Java and Google provides good Android development tools for Java programming language. Android client application will be covered in the next section, while this one will describe the client stub library which is written in Java. This code can be executed on every hardware that is supported by Java virtual machines. Application architecture in this projects is based on the ideas of extensibility and portability and therefore Java is a good choice to start.

I have implemented an universal library for the RPC communication with the previously described embedded service. That library is easy to add to any Java project and start to develop a new control application. It can be extended to add some required funtionality.

3.3.2 Library structure

The library interface should be identical to a service interface description or service contract. The JSON service descriptor from the Appendix A has enough information to create a binding library to that embedded service.

This client code can be realized in two different ways: static and dynamic binding. The library described here, is a static binding library, because it has an already defined interface class and client methods are known at the compile time.

Dynamic libraries can determine available server methods at the runtime. Client library may connect to a server and receives a list of operations. For example JSON service contract that is used in this work, can be obtained and parsed at the runtime. Dynamic proxy class can be created from received list of methods and methods can be called only by a name without the compile time checking. Client may know only a method name or a part of that name and decide while application is running, which methods to call. Domain keywords for Java are: *reflection*, *dynamic proxy class*, *duck typing*. The Python example above uses this approach and you can retrieve list of available methods from server by calling `listMethods()` on a `ServerProxy` object. This techniquie is more suitable for dynamically typed programming languages.

I preferred to make more robust and simple solution. I created a Java interface class with all methods that server has. The definition of the main interface classes in the library is provided below.

[44] Java Virtual Machine

```java
public interface Service {

    // connect methods
    void connect(Reader inputReader, Writer outputWriter);
    boolean isConnected();
    void disconnect();

    boolean addListener(RPCServiceListener listener);
    boolean removeListener(RPCServiceListener listener);

    void setTimeoutMs(long timeoutMs);
    long getTimeoutMs();

    void setRequestProcessor(RequestProcessor requestProcessor);
}

public interface CoffeeMachineService extends Service {
    ServiceContract getServiceContract();

    Map<String, Object> getInfo();

    // products
    List<Product> getProducts();
    Product.Status orderProduct(int productId);
    Product.Status cancelProduct(int productId);
    Product.Status getProductStatus(int productId);
}
```

Listing 21: RPC client main interface class

Some object oriented design techniques are used here to get a more abstract code. `CoffeeMachineServi` extends another interface `Service`, which has method that are common to every RPC service. `CoffeeMachineService` interface contains only coffee machine related methods and inherits other methods from the parent class.

Client can use these classes in the application code and does not need to know the implementation details. To start using it client needs to create an instance of `JsonRpcCoffeeMachineService` (the implementation of `CoffeeMachineService` interface), connect it to `Reader` and `Writer` stream objects and after that user can call RPC methods.

Character encoding

Java programming language has the abstraction of `Reader` and `Writer`. These are character stream interfaces that allow to write or read characters to/from underlying information sources. JSON-RPC is a text based protocol and we only need to handle character data in our application, but not binary data.`Reader` and `Writer` are parents of `InputStreamReader` and `OutputStreamWriter`, which are bridge from byte streams to character streams. They transfer bytes to characters and back using a specified character set[45]. JVM internally stores characters as 16-bit Unicode variables, but in other systems character can be: an US-ASCII seven-bit value, UTF-8 multi-byte encoding, Extended Binary Coded Decimal Interchange Code (EBCDIC) 8-bit character encoding and others. Character encoding needs to be specified for `InputStreamReader` and `OutputStreamWriter` in order to convert data properly. We cannot use byte streams in our library because we cannot predict in which encoding character data will be received from the data byte source. It might happen that some byte will have different meaning from what we expect. Therefore we use an abstraction of `Reader` and `Writer` here.

Client of our library should receive an `InputStream` object (operates with byte streams) from somewhere, provide valid character encoding and create an `InputStreamReader` from `InputStream`, and pass it to `connect()` method of the RPC library. The source of an `InputStream` may be the array of bytes in the memory, a file on the disk, a network socket or even String objects. In general

[45] the mapping between characters and sequences of bytes

words, client should decide where from data bytes should come, where they need to be written, and wrap these bytes into character streams.

Message encoding

This library assumes that received characters are encoded using netstrings[46] message encoding. `MessageReader` class in the library has the implementation of similar finite state machine parsing algorithm, that was used in the embedded server implementation. It reads the netstrings encoded messages, extracts the data from them and sends extracted characters to `MessageHandler` for processing.

The `MessageWriter` class composes a valid netstring message and uses the provided by a client `Writer` object to write that message to the destination. `MessageWriter` receives JSON objects,converts them into a String data, wraps them in a netstring message and sends this character data to the destination.

JSON serialization and the JSON-RPC

This JSON-RPC client is based on the `jsonrpc2-base` library from Vladimir Dzhuvinov (`http://software.dzhuvinov.com`). This library provides a ready classes for handling JSON-RPC requests and responses. Table 6 shows the entire lifecycle of a RPC call.

Step	Side	Action	Used methods
1	Client	Create a new request	`JSONRPC2Request()`
2	Client	Serialize request to string and send	`JSONRPC2Request.toString()`
3	Server	Parse received string back to request object	`JSONRPC2Request.parse()`

4	Server	Get the request data	`JSONRPC2Request.getMethod()`
			`JSONRPC2Request.getParamsType()`
			`JSONRPC2Request.getPositionalParams()`
			`JSONRPC2Request.getNamedParams()`
			`JSONRPC2Request.getID()`

5	Server	Create a response	`JSONRPC2Response()`
6	Server	Serialise response to string and send back	`JSONRPC2Response.toString()`
7	Client	Parse received string back to response object	`JSONRPC2Response.parse()`

8	Client	Check the response for success, get the result/error	`JSONRPC2Response.indicatesSuccess()`
			`JSONRPC2Response.getResult()`
			`JSONRPC2Response.getError()`

Table 6: jsonrpc2-base library RPC methods [44]

`jsonrpc2-base` is able to serialize Java data types to JSON objects Listing 22 shows how RPC request object is created from the standard Java data types and Table 7 shows how JSON to JAVA mappings are done.

[46] `<LENGTH>:<DATA>,` format, for more details see 3.2.3

```java
// The remote method to call
String method = "makePayment";

// The required named parameters to pass
Map<String,Object> params = new HashMap<String,Object>();
params.put("recipient", "Penny Adams");
params.put("amount", 175.05);

// The mandatory request ID
String id = "req-001";

// Create a new JSON-RPC 2.0 request
JSONRPC2Request reqOut = new JSONRPC2Request(method, params, id);

// Serialize the request to a JSON-encoded string
String jsonString = reqOut.toString();

// jsonString can now be dispatched to the server...
```

Listing 22: RPC request creating from Java standard data types [44]

JSON	Java
true or false	java.lang.Boolean
number	java.lang.Number
string	java.lang.String
array	java.util.List
object	java.util.Map
null	null

Table 7: JSON↔Java data type mapping in `jsonrpc2-base` library [44]

Data flow

This paragraph describes the client stub library RPC call in details. Although `jsonrpc2-base` has classes for handling JSON-RPC message objects, it does not provide any client functionality. We need to write a request handling logic to support the communication with hardware embedded service.

When client of our library calls any method on `CoffeeMachineService` class this method starts to prepare a new request. Service interface implementation add necessary parameters to RPC request (name, method parameters, an id), and sends it to request processor.

The whole RPC call process visualized at the Figure 13

`RequestProcessor` is a simple object with a `Object processRequest(Object request)` method which encapsulated request processing logic. Strategy design pattern is implemented here. Each request can be processed in a many various ways. Some methods have slow execution on the server and require bigger timeout time on the client side. Another methods could fail and additional error handling logic is required. This design methods help to define different request handling strategies and apply them at the runtime. For example the `getProducts()` method has very short response time, because it does not trigger any events on the coffee machine. Proxy device just returns a list of products from memory storage. Another method, the `orderProduct()`, needs additional communication and verification of parameters. These two methods may have two different algorithms of processing client RPC request, while programming interface needs to stay fixed.

There is only one strategy for the `RequestProcessor` implemented yet. This processor sends a request to a message handler only ones and starts to wait for a response from a `RequestWaiter` help class. After some timeout it returns the result or a `null` value if there was not any response received.

Message writer module of the client stub implements the `RPCServiceListener` (pay attention to the `addListener()` and `removeListener()` methods in listing 21) interface. This is a

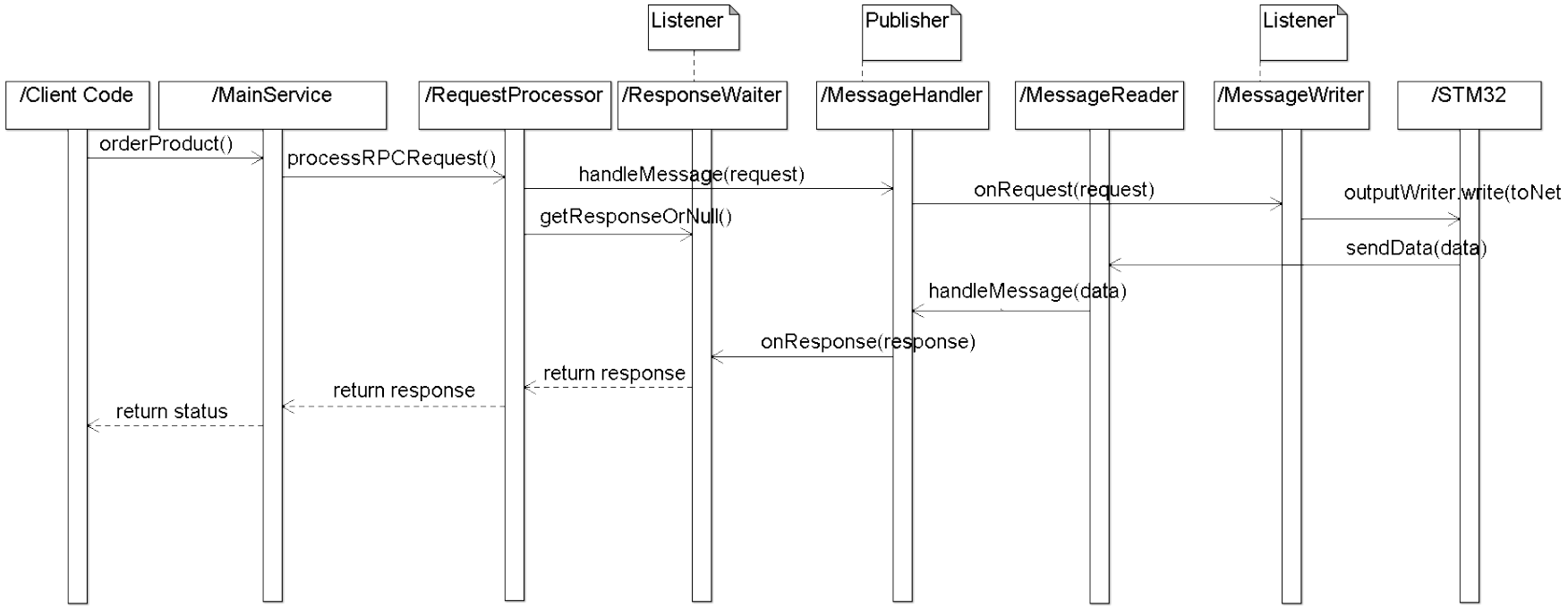


Figure 13: RPC call in the client stub library

publish-subscribe mechanism used in this library for event handling. Service implementation contains a list of service listeners, who are listening for various RPC messages. Listing 23 contains a list o messages that can be captured by the `RPCServiceListener`. It also contains a interface of a class that publishes internal system events.

```
public interface RPCServiceListener {
    void onRequest(RPCRequest request);
    void onResponse(RPCResponse response);
    void onNotification(RPCNotification notification);
    void onUnknownMessage(String messageText);
}

public interface MessageHandler {
    void handleMessage(String msg);
    void handleMessage(RPCRequest request);
    void handleMessage(RPCNotification notification);
    void handleMessage(RPCResponse response);
}
```

Listing 23: RPC event listener and event publisher interfaces

Message writer thread listens for the `onRequest()` and `onNotification()` events. It starts to write request message data to remote embedded device using the output character `Writer` when these methods are triggered by the request publishers.

System event methods are triggered by a `MessageHandler`. This is a small routing class that decides where to put moving requests, notification, responses and errors according to their types. For example, if `handleMessage(RPCRequest request)` message handler method is called from somewhere in the code, message handler publishes received `RPCRequest` object to all subscribed listeners. Message handler calls `onRequest(RPCRequest request)` method on each service subscriber and passes new `RPCRequest` as parameter.

Messages from the the remote device are read by a `MessageReader` class. This class reads incoming netstring packets and passes extracted data to the `MessageHandler`. `MessageHandler` parses incoming JSON data and detects that this data type is a JSON-RPC response object. This Response message becomes published for all listeners including the `ResponeWaiter` help class, who receives message and return it to `RequestProcessor`.

Now the `RequestProcessor` have received a response it was waiting for and it can be returned to `MainService` class. Not the data from the response message can be extracted and remote call result may be returned to a client code, from where it was initially called.

This is how JSON-RPC messages flow inside service client library.

It might seem quite complicated, but it has lots of advantages. Using a publish-subscribe mechanism you might add multiple event listeners to the system very easily. For example if you need a request logger, you create a class which implements a `RPCServiceListener` interface and register it in a main service class by calling special methods. Now your logger can listen all RPC messages and log the information about them.

Conclusion

This library can be used in the client application code as the abstract interface of a remote coffee machine. It makes programming of client code more elegant and easy. System programmer does not need to develop RPC related code, he can simply use this interface in his projects. The only necessary step is the connection of a service instance class to the data input and output character streams. This library uses the netstrings character data encoding, which is a very simple way how to transfer character data messages. The whole design is based on the JSON-RPC communication protocol.

3.4 Implementation of Android client example applications

The prototype of client-server application was designed during this research project. The application program was executed on the Google Android operating system powered hardware.

Main idea of this application is the remote wireless control of some electronic device. The controlled device example here is a coffee machine, that has some functions like preparing a cup of coffee. We needed that these functions became available for remote control.

The developed application is a small application with trivial user interface, which allows to prepare products by selecting a product from a small catalogue. The screen shot of a ready application is provided in the Figure 14

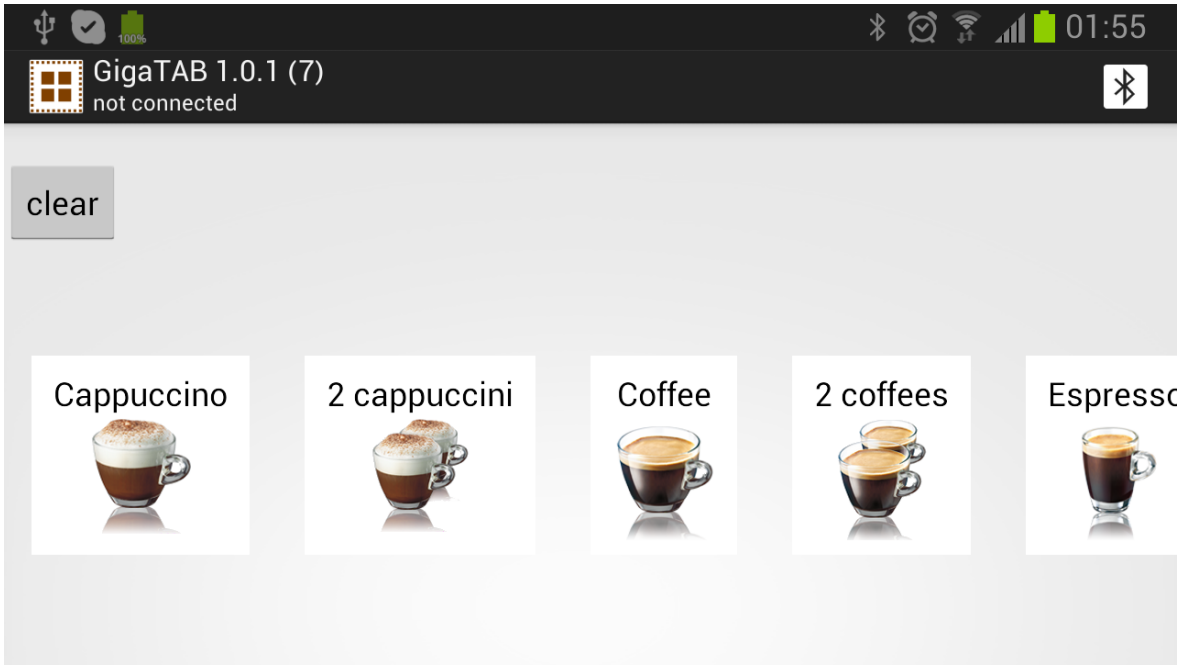


Figure 14: Android application visual interface

The usage scenario starts from the connection to coffee machine. There is a small button with Bluetooth icon in the top right corner of the application. This button activates a device select dialog, where you can find a remote device by name and MAC address and connect by selecting it in the list. When connection is established some available products become showed in the horizontal list layout. It is able to point to each of these products and open a dialog box with detailed information about each product. User can start coffee preparing from that dialog. While operation is in progress, it is still possible to cancel the product preparing operation.

I will not cover in details the whole process of the user interface and application creation. This is out of the scope of this research work and it requires some additional domain knowledge. You need to get familiar with Google Android development tools, read a documentation course, follow the tutorials and study by doing. There are available lots of application examples and it is not very hard to produce a similar application. This application is partially based on the Bluetooth chat communication example from the Android Software development kit.

Whole communication is performed over wireless Bluetooth protocol. It is assumed, that Bluetooth is the abstract transport that can send data bytes over radio link. The Android Bluetooth API functions can search to nearby Bluetooth devices and provide you a list of `BluetoothDevice` objects. You can find your wireless device in that list and tell the Android OS to connect to that device. This device list is also used to fill a device choosing graphical dialog described before.

When devices get connected it is able to receive a communication socket object and get a standard `java.io.InputStream` and `java.io.OutputStream` from that. This is a final step of establishing the communication and it is able to transfer data over received stream objects.

The client RPC library is connected to these streams and it is able to communicate with a remote device over the wireless serial interface. RPC calls may be started right after application gets connected to the Bluetooth input and output streams.

Now it is up to your imagination how to use and extend this remote embedded service. You can implement any rich functionality and create whatever complex system you want or your hardware resources allow you. This extensible service oriented embedded architecture allows you to do so.

4 Conclusions

Several possible variants of SOA implementation were covered in this work. Some of them are mature and standardized tools which are used in many production systems nowadays. Another ones are lightweight and beautiful alternatives to the first group. They can be used even in the small microcontroller devices. Their specifications does not tie a programmer to work with a limited specific set of technologies. Some of covered tools can be used in any environment and support all creativity of the developers. JSON and XML, the ideas of RPC, they all can be used in many various ways.

Author was trying to analyse all of them in his research. This work chosen some comparison of SOA implementations and related technologies is the context of the embedded systems. Some technologies were chosen for the trial. The choice of the technologies does not mean that they are the best and the only suitable solution for the application described here. Author's aim was to show that it is possible to create the systems of small devices using lightweight protocols and data structures.

Some specific application field was chosen as an example in this work. Lets watch how it was realized and overview the results.

4.1 Results

There was a defined goal of this work: make a research of available machine-to-machine communication techniques and deliver a functional system prototype to fulfill some business needs.

Author has a freedom of choosing any suitable technology and was limited only with the available hardware platform. This work can be characterize as a research activity of detecting possible facilities, not than an implementation of specific task of interconnecting a mobile phone and the coffee machine. There should be a goal to reach, some problem which needs a solution in order to start watching around. I have defined this abstract task and decided to make a case study about it.

One of multiple ways how to design a client-server system is the SOA. Services are self-describing and open components, which use standard communication schemes. Author of this work was inspired by this approach and have created a SOA based system.

This paper contains a research of different technologies that help to implement services on the devices. It is unable to find and analyse all possible solutions. The most popular and open approaches were analysed instead. Author has chosen a concept of remote procedure calls and found a simple and lightweight solution of JSON-RPC.

I had no experience of programming microcontrollers before. The company, where i was doing my university internship, provided me one specific microcontroller family and my task was to learn how it can be applied. To find out the hardware possibilities you need to study the architecture of a specific devices. This work became for me not only a research project, but a complete microcontroller practical course. I have learned how to use different development tools and STM32F1 and other ARM Cortex-M3 microcontroller hardware platform features, exercised low level C programming and even soldered wires on development boards to repair boundary scan debugger. This was done only by myself with the help of internet resources and other information sources. This was the largest laboratory work i have ever experienced.

A small remote service was finally implemented on a STM32F103ZE microcontroller. This server is able to response for remote client requests and provide some useful services. This device is a proxy bridge between client application and coffee machine hardware. Only a few set of functions were implemented, but this functionality reflects all the architecture features. It is able to determine the structure of this system by tracing how a single RPC request is transferred between different system modules..

This solution uses embedded realtime operating system FreeRTOS, so i was involved in a development of a parallel and multitasking system. Server program consists of parallel tasks, which are communicating to each other and process incoming requests. Internals are implemented as chain of processing units, which pass work to each other.

The client side of application is designed as a Java library that can be used in any kind of client application. It provides remote device methods and handles all communication between the client code and embedded service server.

This library is used inside the Android demo application, that is executed on the mobile phone or tablet. Android program was designed to be an example application case. This provides some controls over the remote coffee machine.

The goal of this work was achieved. This was a research of a big bunch of technologies: embedded hardware related highly constrained and optimized software was studied together with

modern internet technologies that connect billions different machines.

Lots of new things and interesting computer science approaches were found during accomplishing the tasks and information mining. Obtained experience and knowledge would help me to deliver more quality products in future and not to make those mistakes that every young computer system engineer does.

4.2 Future work

This small implementation of client-server application connects together only two devices. This does not define any additional methods how to find devices in a network and connect other devices. This is only an abstract service architecture that was designed to be universal.

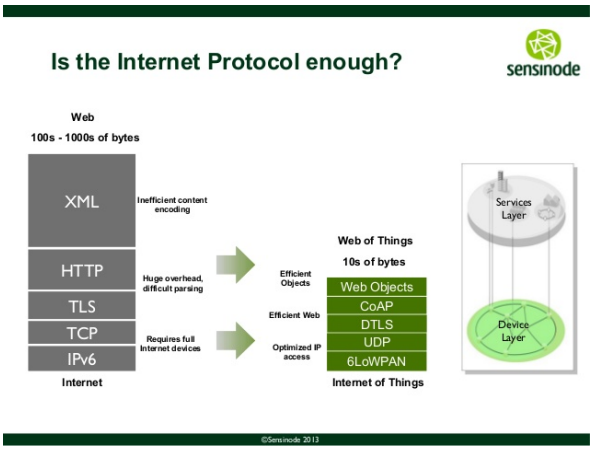
There is no security, addressing, discovery and quality of service. It requires some reliable transport protocol stack that covers data transferring from physical communication layer up to application layer.

In order to become a real world product and be integrated to other systems there should be dedicated connectivity requirements. Devices should be connected in some standard way.

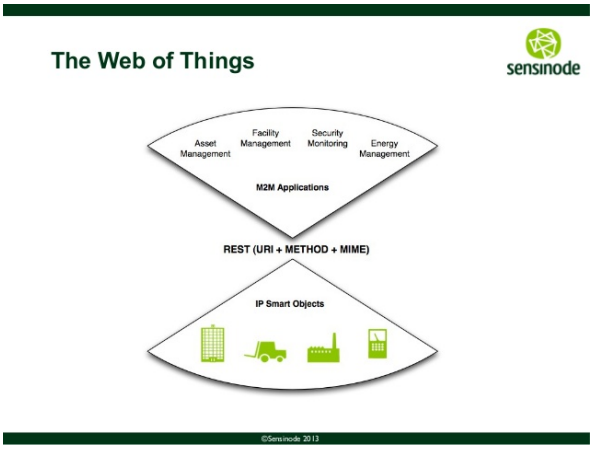
Most of reviewed information sources covered the development of embedded services in context of internet and TCP/IP networks.

I see the next development cycle of this architecture to be the integration to the Internet of Things. Many research groups are working on developing distributed communication for small embedded devices and wireless sensor networks on the Internet. Their work papers include keywords like 6LoWPAN, IEEE 802.15.4, IPv6, RPL, UDP, COAP, RESTful embedded services.

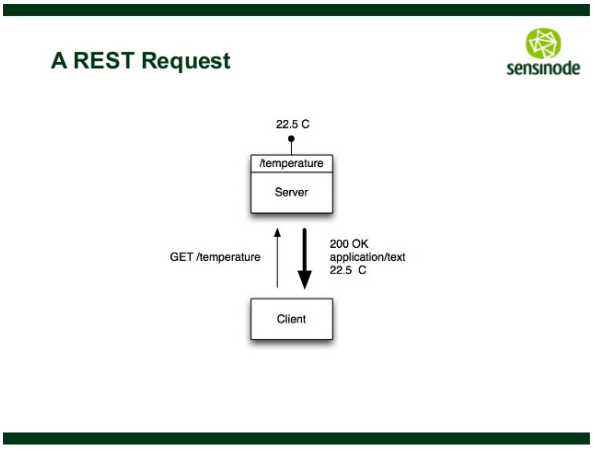
Figure 15 shows how this technologies could be applied.



(a) Optimized IP protocols



(b) RESTful Web things Things architecture



(c) RESTful temperature sensor usage example

Figure 15: Optimized internet technologies for networked embedded systems [45, 46]

During this research i have found a project named Contiki. Contiki is an open source operating system for networked, memory-constrained systems with a particular focus on low-power wireless Internet of Things devices. Examples of where Contiki is used include street lighting systems, sound monitoring for smart cities, radiation monitoring systems, and alarm systems.[47]

This project supports very resource efficient full IP networking[48] and is optimized for tiny systems, having only a few kilobytes of memory available. This is a good candidate to be the main technology of this kind of system we have been developed here.

Our design could be rewritten for the Contiki OS and we could migrate to the hardware used

[47] http://www.contiki-os.org/

[48] uIP TCP/IP Stack from Adam Dunkels. See [47]

in wireless sensor networks. This could be a very tiny chip with some wireless interface onboard. It is able to build very low-power and low-cost embedded web service using this technology stack.

This kind of work requires another exhaustive research and laboratory experiments. This field is the topic of interest of many scientists and companies, who produce networked embedded solutions. It is quite popular nowadays and there are many use cases and benefits using distributed embedded network computing.

More and more devices gets connected to the internet. This technologies could help to improve the quality of our lives by introducing a network of low-cost and low-power sensors and actuators.

A Examples of service contracts

Example of WSDL contract from WSDL documentation [9]:

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/ns/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    xmlns:wsoap= "http://www.w3.org/ns/wsdl/soap"
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsdlx= "http://www.w3.org/ns/wsdl-extensions">

  <documentation>
    This document describes the GreatH Web service.  Additional
    application-level requirements for use of this service --
    beyond what WSDL 2.0 is able to describe -- are available
    at http://greath.example.com/2004/reservation-documentation.html
  </documentation>

  <types>
    <xs:schema
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://greath.example.com/2004/schemas/resSvc"
        xmlns="http://greath.example.com/2004/schemas/resSvc">

      <xs:element name="checkAvailability" type="tCheckAvailability"/>
      <xs:complexType name="tCheckAvailability">
        <xs:sequence>
          <xs:element  name="checkInDate" type="xs:date"/>
          <xs:element  name="checkOutDate" type="xs:date"/>
          <xs:element  name="roomType" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>

      <xs:element name="checkAvailabilityResponse" type="xs:double"/>

      <xs:element name="invalidDataError" type="xs:string"/>

    </xs:schema>
  </types>

  <interface  name = "reservationInterface" >

    <fault name = "invalidDataFault"
            element = "ghns:invalidDataError"/>

    <operation name="opCheckAvailability"
            pattern="http://www.w3.org/ns/wsdl/in-out"
            style="http://www.w3.org/ns/wsdl/style/iri"
            wsdlx:safe = "true">
        <input messageLabel="In"
              element="ghns:checkAvailability" />
        <output messageLabel="Out"
              element="ghns:checkAvailabilityResponse" />
        <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
    </operation>

  </interface>

```

```
58    <binding name="reservationSOAPBinding"
59            interface="tns:reservationInterface"
60            type="http://www.w3.org/ns/wsdl/soap"
61            wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
62
63      <fault ref="tns:invalidDataFault"
64        wsoap:code="soap:Sender"/>
65
66      <operation ref="tns:opCheckAvailability"
67        wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
68
69    </binding>
70
71    <service name="reservationService"
72         interface="tns:reservationInterface">
73
74       <endpoint name="reservationEndpoint"
75                 binding="tns:reservationSOAPBinding"
76                 address ="http://greath.example.com/2004/reservation"/>
77
78    </service>
79
80  </description>
```

TODO put last version of contract here Example of JSON service descriptor:

```
{
        "$schema": "http://json-schema.org/draft-04/schema#",
        "id" : "http://ttu.ee/denis_konstantinov/embedded_service.schema",
        "description" : "This is a embedded service of a coffee machine.",
        "version" : "0.1",
        "operations" : {
                "system.getLimits" : {
                        "description" : "Get limits of physical capabilities of a sistem.",
                        "type" : "method",
                        "params" : null,
                        "returns" : {
                                "description" : "Limits of JSON-RPC server",
                                "type" : "object",
                                "properties": {
                                        "jsonMaxDocLength": {
                                                "description" : "Maximum length of a JSON document string that
                                                "type": "number",
                                                "minimum" : 0
                                        },
                                        "maxIncomeMessages": {
                                                "description" : "Maximum simultaneous messages that server cou
                                                "type": "number",
                                                "minimum" : 1
                                        }
                                },
                                "required": ["jsonMaxDocLength", "maxIncomeMessages"]
                        }
                },
                "system.help": {
                        "description" : "Prints this help document.",
```

```
                "type" : "method",
                "params" : null,
                "returns" : {
                        "description" : "Full service description",
                        "type" : "string"
                }
        },
        "machine.getInfo" : {
                "description" : "Get information about current coffee machine. This returns some machi
                "type" : "method",
                "params" : null,
                "returns" : {
                        "description" : "Coffee machine version info",
                        "type" : "object",
                        "properties": {
                                "machineName": { "type": "string" },
                                "machineFirmvareVersion": {
                                        "type": "string"
                                }
                        },
                        "required": ["machineName", "machineFirmvareVersion"]
                }
        },
        "machine.getProducts" : {
                "description" : "Get list of products that current machine can prepare.",
                "type" : "method",
                "params" : null,
                "returns" : {
                        "description" : "Coffee machine version info",
                        "type": "array",
                        "minItems": 0,
```

```
                        "items": { "$ref": "#/definitions/Product" },
                        "uniqueItems": true
                }
        },
        "machine.orderProduct": {
                "description" : "Order product of your interest. Specify product identificator",
                "type" : "method",
                "params" : [ { "name": "id", "type": {"$ref": "#/definitions/positiveInteger"}, "requi
                "returns" : {
                        "description" : "Product order status status of product",
                        "enum": [ "PRODUCT_STATUS_STARTED", "PRODUCT_STATUS_FAILED", "PRODUCT_STATUS_F
                }
        },
        "machine.getProductStatus": {
                "description" : "Get status of ordered product. Specify product identificator",
                "type" : "method",
                "params" : [ { "name": "id", "type": {"$ref": "#/definitions/positiveInteger"}, "requi
                "returns" : {
                        "description" : "Current status of product",
                        "enum": [ "PRODUCT_STATUS_STARTED", "PRODUCT_STATUS_IN_PROGRESS", "PRODUCT_STA
                }
        }
        "machine.cancelProduct": {
                "description" : "Order product of your interest. Specify product identificator",
                "type" : "method",
                "params" : [ { "name": "id", "type": {"$ref": "#/definitions/positiveInteger"}, "requi
                "returns" : {
                        "description" : "Status of cancel operation",
                        "enum": [ "PRODUCT_STATUS_STARTED", "PRODUCT_STATUS_FAILED", "PRODUCT_STATUS_F
                }
        }
```

```
        },
        "definitions": {
                "Product": {
                        "description": "The product of current coffee machine.",
                        "type" : "object",
                        "properties": {
                                "id": { "$ref": "#/definitions/positiveInteger" },
                                "name": { "type": "string" },
                                "price": { "$ref": "#/definitions/Price" },
                                "image": {
                                        "description": "Base64 encoded image of a product",
                                        "type": "string"
                                }
                        },
                        "required": ["id", "name"]
                },
                "Price": {
                        "type" : "object",
                        "properties": {
                                "amount": { "$ref": "#/definitions/positiveNumber" },
                                "currency":  { "enum": [ "EUR", "USD", "LTL", "LVL", "RUB", "SEK" ] }
                        },
                        "required": ["amount", "currency"]
                },
                "positiveInteger": {
                        "type": "integer",
                        "minimum": 0
                },
                "positiveNumber": {
                        "type": "number",
                        "minimum": 0.0
```

```
124                             }
125                         }
126     }
```

List of Figures

List of Tables

References

[1] F. Jammes, A. Mensch, and H. Smit, "Service-oriented device communications using the devices profile for web services," in *Advanced Information Networking and Applications Workshops, 2007, AINAW '07. 21st International Conference on*, vol. 1, pp. 947–955, 2007. 9, 28, 29

[2] N. Milanovic, J. Richling, and M. Malek, "Lightweight services for embedded systems," in *Software Technologies for Future Embedded and Ubiquitous Systems, 2004. Proceedings. Second IEEE Workshop on*, pp. 40–44, 2004. 9, 32

[3] Wikipedia, "Service-oriented architecture." http://en.wikipedia.org/wiki/Service-oriented_architecture, 2013. [Online; accessed 17-July-2013]. 9

[4] H. Kreger, "Web Services Conceptual Architecture." WWW, May 2001. 10, 11, 71

[5] World Wide Web Consortium, "Web Services Architecture," tech. rep., 2004. [Online; accessed 19-July-2013]. 12, 13, 71

[6] World Wide Web Consortium, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)," tech. rep., 2007. [Online; accessed 19-July-2013]. 12, 30

[7] Wikipedia, "WS-Security." http://en.wikipedia.org/wiki/WS-Security, 2013. [Online; accessed 29-July-2013]. 13

[8] Distributed Management Task Forse, Inc., "DMTF Fact Sheet: WS-Management." http://www.dmtf.org/standards/wsman, 2013. [Online; accessed 29-July-2013]. 13

[9] World Wide Web Consortium, "Web Services Description Language (WSDL) Version 2.0 Part 0: Primer," tech. rep., 2007. [Online; accessed 19-July-2013]. 13, 14, 64, 71

[10] Wikipedia, "Web Services Description Language." http://en.wikipedia.org/wiki/WSDL, 2013. [Online; accessed 17-July-2013]. 13, 14, 71

[11] OASIS UDDI Specification Technical Committee, "UDDI Spec Technical Committee Draft, Dated 20041019," tech. rep., 2004. [Online; accessed 29-July-2013]. 13

[12] R. Elfwing, U. Paulsson, and L. Lundberg, "Performance of soap in web service environment compared to corba," in *Software Engineering Conference, 2002. Ninth Asia-Pacific*, pp. 84–93, 2002. 15

[13] C. Groba and S. Clarke, "Web services on embedded systems - a performance study," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pp. 726–731, 2010. 15, 28, 29, 30

[14] Wikipedia, "Common Object Request Broker Architecture." http://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture, 2013. [Online; accessed 30-July-2013]. 15

[15] D. Guinard, I. Ion, and S. Mayer, "In search of an internet of things service architecture: Rest or ws-*? a developers' perspective," in *Proceedings of Mobiquitous 2011 (8th International ICST Conference on Mobile and Ubiquitous Systems)*, (Copenhagen, Denmark), pp. 326–337, Dec. 2011. 15, 28

[16] Bray, Tim, "WS-Pagecount." http://www.tbray.org/ongoing/When/200x/2004/09/21/WS-Research, 2004. [Online; accessed 31-July-2013]. 15

[17] R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. 15, 18, 19, 71

[18] Stefan Tilkov, "A Brief Introduction to REST." http://www.infoq.com/articles/rest-introduction, 2007. [Online; accessed 1-August-2013]. 16, 19

[19] Wikipedia, "World Wide Web." http://en.wikipedia.org/wiki/World_Wide_Web, 2013. [Online; accessed 1-August-2013]. 16

[20] Wikipedia, "Representational state transfer." http://en.wikipedia.org/wiki/REST, 2013. [Online; accessed 1-August-2013]. 17, 71

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc 2616, hypertext transfer protocol – http/1.1," 1999. 17

[22] A. Tanenbaum and M. V. Steen, "Distributed systems principles and paradigms," 2007. 19, 22

[23] Python Software Foundation, "xmlrpclib — XML-RPC client access." http://docs.python.org/2/library/xmlrpclib.html, 2013. [Online; accessed 5-August-2013]. 21

[24] American Arium, "Endianness and ARM® Processors. Application note," 2003. 22

[25] JSON-RPC Working Group <json-rpc@googlegroups.com>, "JSON-RPC 2.0 Specification." http://www.jsonrpc.org/specification, 2010. [Online; accessed 13-August-2013]. 23, 24

[26] Malin Eriksson, Victor Hallberg, "Comparison between json and yaml for data serialization," Master's thesis, Royal Institute of Technology, Sweden, 2011. 25

[27] Wikipedia, "Comparison of data serialization formats." http://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats, 2013. [Online; accessed 6-August-2013]. 25

[28] Tim Anderson, "Introducing XML." http://www.itwriting.com/xmlintro.php, 2004. [Online; accessed 6-August-2013]. 26

[29] Nurzhan Nurseitov and Michael Paulson and Randall Reynolds and Clemente Izurieta, "Comparison of JSON and XML Data Interchange Formats: A Case Study," in *Proceedings of the ISCA 22nd International Conference on Computer Applications in Industry and Engineering, CAINE 2009, November 4-6, 2009, Hilton San Francisco Fisherman s Wharf, San Francisco, California, USA* (D. Che, ed.), pp. 157–162, ISCA, 2009. 26, 27

[30] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," tech. rep., 2008. [Online; accessed 6-August-2013]. 26

[31] World Wide Web Consortium, "HTML 4.01 Specification." http://www.w3.org/TR/1999/REC-html401-19991224/, 1999. [Online; accessed 6-August-2013]. 27

[32] Douglas Crockford, "Introducing JSON." http://www.json.org/, 2013. [Online; accessed 6-August-2013]. 27, 48

[33] G. Wang, "Improving data transmission in web applications via the translation between xml and json," in *Communications and Mobile Computing (CMC), 2011 Third International Conference on*, pp. 182–185, 2011. 27

[34] R. Kyusakov, H. Makitaavola, J. Delsing, and J. Eliasson, "Efficient xml interchange in factory automation systems," in *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*, pp. 4478–4483, 2011. 27

[35] B. Villaverde, D. Pesch, R. De Paz Alberola, S. Fedor, and M. Boubekeur, "Constrained application protocol for low power embedded networks: A survey," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pp. 702–707, 2012. 28

[36] Web Services for Devices (WS4D), an initiative bringing., "Devices Profile for Web Services." http://ws4d.e-technik.uni-rostock.de/technology/dpws/, 2013. [Online; accessed 7-August-2013]. 28, 29, 71

[37] Internet Engineering Task Force (IETF), "Constrained Application Protocol (CoAP). draft-ietf-core-coap-18." http://datatracker.ietf.org/doc/draft-ietf-core-coap/, 2013. [Online; accessed 7-August-2013]. 29

[38] Wikipedia, "Constrained Application Protocol." http://en.wikipedia.org/wiki/Constrained_Application_Protocol, 2013. [Online; accessed 8-August-2013]. 30

[39] M. Kovatsch, S. Duquennoy, and A. Dunkels, "A low-power coap for contiki," in *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, pp. 855–860, 2011. 30

[40] STMicroelectronics, *UM0426 User manual. STM3210B-EVAL evaluation board*, 2007. 34, 36, 71

[41] R. Barry, *Using the FreeRTOS Real Time Kernel: A Practical Guide*. Real Time Engineers Limited, 2010. 38

[42] Real Time Engineers Ltd., "FreeRTOS website." http://www.freertos.org/, 2013. [Online; accessed 12-August-2013]. 38, 42, 43, 71

[43] STMicroelectronics, *AN3109 Application note. Communication peripheral FIFO emulation with DMA and DMA timeout in STM32F10x microcontrollers*, 2009. 44

[44] Vladimir Dzhuvinov, "jsonrpc2-base – Minimalist Java library for composing and parsing JSON-RPC 2.0 messages." http://software.dzhuvinov.com/json-rpc-2.0-base.html, 2013. [Online; accessed 14-August-2013]. 55, 56, 71

[45] Zach Shelby, Chief Nerd, "Standards Drive the Internet of Things." http://www.slideshare.net/zdshelby/standards-drive-the-internet-of-things, 2013. [Online; accessed 15-August-2013]. 62, 71

[46] Zach Shelby, Chief Nerd, "CoAP: The Internet of Things Protocol." http://www.slideshare.net/zdshelby/coap-tutorial, 2013. [Online; accessed 15-August-2013]. 62, 71

[47] A. Dunkels, *Programming Memory-Constrained Networked Embedded Systems*. PhD thesis, Swedish Institute of Computer Science, 2007. 62