

Remote Control of Device behind firewall using a RESTful service

M. Christian van Zanten

DTU



Kongens Lyngby 2012
IMM-MSc-2012-114

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-MSc-2012-114

Summary (English)

This thesis describes how a Web server on a device behind a firewall can be exposed via a central mediator. Brüel & Kjær is a world-leading manufacturer and supplier of sound and vibration test and measurement solutions. They produce the hand held type 2250 device which is used as a portable device or form part of stationary monitoring systems. The 2250 can be controlled remotely via the Web server present on the device. This currently requires that the Web server can be addressed and that relevant ports are open on firewalls.

This project describes how these requirements can be circumvented by introducing a central service to mediate traffic between user and the 2250. State of the art solutions combined with user scenarios are the basis of the requirements for the developed proof of concept prototype. This prototype is based on cloud computing on the Windows Azure platform and RESTful Web service architecture. In relation to these technologies key concepts, risk and risk mitigation are discussed.

Based on these technologies the design and implementation of the prototype is chosen and specified. The implementation of the prototype is validated using functional and performance tests. It is found that mediating traffic via a central service hosted on a cloud computing platform is a viable solution, but the increased latency makes it unfit for real time remote controlling. It does however open a wide range of possibilities for controlling a device where response time is not critical. Before this can be used commercially security, scope, and business case have to be defined.

Keywords: Remote control, Remote access, HTTP, REST, RESTful Web service, Cloud Computing, Windows Azure, Reverse Proxy, CAP Theorem.

Summary (Dansk)

Denne afhandling beskriver hvordan en Web server på en enhed bag en firewall kan eksponeres via en central service. Bruel & Kjær er en ledende fabrikant og leverandør af lyd og vibrations løsninger til test og målinger. Brüel & Kjær fremstiller den håndholdte enhed 2250, der bruges som en bærbar måle enhed eller som del af et stationært overvågnings system. 2250'eren kan fjernstyres via enhedens Web server. Dette kræver at Web serveren kan adresseres og at de relevante porte er åbne på firewalls.

Denne afhandling beskriver hvordan disse krav kan blive omgået ved at introducere en central service til at formidle trafik mellem bruger og enhed. State-of-the-art løsningers funktionalitet kombineret med brugerscenarier danner basis for kravene til den udviklede proof-of-concept prototype. Prototypen er baseret på cloud-computing på Windows Azure platformen og RESTful Web service arkitektur. I relation til disse teknologier er centrale begreber, risici og risicireduktion gennemgået.

Prototypens implementering og design er blevet valgt og specificeret på baggrund af disse teknologier. Prototypens implementering er blevet valideret vha. funktionelle- og ydelsestests. Konklusionen på dette er at formidling af trafikken via en central service placeret på en cloud-computing platform er en funktionsdygtig løsning, men den yderligere latens gør den uegnet som realtids-fjernstyring. Den åbner derimod en bred vifte af muligheder for fjernstyring af en enhed, hvor responstid ikke er kritisk. Før denne løsning kan bruges i kommercielt øjemed skal sikkerhed, gyldighedsområde og business case defineres.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics. The thesis was prepared in collaboration with Brüel & Kjær Sound and Vibration.

The thesis deals with the opportunities and risks with remote controlling a device via a local RESTful Web service on the device via a third part service hosted on a cloud platform.

Embrace the beauty of the HTTP standard and enjoy the reading

Lyngby, 3-September-2012

A handwritten signature in blue ink, consisting of a stylized 'M' and 'C' followed by a long horizontal stroke.

M. Christian van Zanten

Acknowledgements

I would like to thank:

Lars Damsgaard for guiding me through the use of the embedded platform

Christian Bækdorf for architectural inspiration and design ideas

Niels Bruun Svendsen for technical input and help with project management

my supervisor **Bjarne Poulsen** for sharing knowledge about thesis writing

Tomasz Cielecki for teaching me about push notifications

Ulrik Andersen for his insights on networks and communication

Connie Hansen for proof reading and mental coaching

My family for accepting the fact that writing a thesis takes time

Contents

Summary (English)	i
Summary (Dansk)	iii
Preface	v
Acknowledgements	vii
1 Introduction	1
1.1 Brüel & Kjær	1
1.2 Hand-held Analyser Type 2250	2
1.3 Noise Monitoring Terminal	4
1.4 Vision	6
1.5 The Problem	6
1.5.1 Scenarios	7
1.5.2 Thesis Definition	7
1.6 Methodology	8
1.7 Outline	9
2 Analysis	11
2.1 State of the Art	11
2.1.1 RemoteAPI and Browser Interface	11
2.1.2 LogMeIn	12
2.1.3 Microsoft Push Notification Service	14
2.1.4 Azure Service Bus and Windows Communication Foundation	16
2.1.5 Functionality Overview	17
2.2 Functional Requirements	17
2.3 Non-Functional Requirements	20
2.4 Use Cases	21

2.5	Use Case Coverage	24
2.6	Domain Analysis	24
2.6.1	Device/2250	24
2.6.2	Customer	27
2.6.3	Registration	27
2.6.4	2250WebServer	27
2.6.5	User Account	27
2.6.6	Preferences	28
2.6.7	Setup	28
2.6.8	Data	29
2.6.9	Status	29
2.6.10	Browser Interface	29
2.6.11	RemoteAPI	30
2.6.12	RelayService	31
2.6.13	RelayClient	31
2.7	Mockups	32
2.8	Chapter Summary	34
3	Technology Analysis	35
3.1	Cloud Computing	35
3.1.1	Cloud Computing Categories	37
3.1.2	Scalability	38
3.1.3	Security Risks in Cloud Computing	40
3.1.4	Choice of Cloud Computing Category and Cloud Provider	41
3.2	Windows Azure	42
3.2.1	Execution Models	42
3.2.2	Messaging	46
3.2.3	Data Management	49
3.2.4	Risk Mitigation in Windows Azure	51
3.3	Web Services	53
3.3.1	SOAP	53
3.3.2	REST	54
3.3.3	Windows Communication Foundation	57
3.3.4	Choice of Web Service technology	57
3.4	REST in depth	58
3.4.1	Client Server	58
3.4.2	Addressability	59
3.4.3	Statelessness	60
3.4.4	Uniform Interface	61
3.4.5	Connectedness	63
3.4.6	Layered System	63
3.4.7	Cache	63
3.4.8	Web Service Security	64
3.4.9	REST Security	66

3.5	Chapter Summary	66
4	Components and Communication	67
4.1	Component Design	67
4.2	Overall Communication	68
4.2.1	Addressable Devices	69
4.3	Communication between RelayClient and RelayService	71
4.3.1	Polling	71
4.3.2	Tunnel	72
4.3.3	Push Notification	74
4.3.4	Choice of Communication Strategy	75
4.4	RelayService and RelayClient Detailed Communication Protocol	76
4.4.1	Packet Types	78
4.4.2	Packet Structure	79
4.5	Chapter Summary	80
5	RelayClient	81
5.1	Behaviour	81
5.2	Design	82
5.2.1	Communication	84
5.2.2	Client Communication	87
5.2.3	CustomHttp	88
5.2.4	RelayClient	88
5.2.5	Detailed Behaviour	89
5.3	Implementation	91
5.3.1	Quirks of The Embedded Platform	92
5.3.2	RelayClient a Generic Proxy	92
5.3.3	Integrating in BasicEnv	93
5.4	Chapter Summary	93
6	RelayService	95
6.1	Choosing an Execution Model	95
6.1.1	Choosing a Messaging system	96
6.1.2	RelayServiceBasic Design	97
6.2	RelayServiceBackend Design and Implementation	99
6.2.1	RelayServiceBackend Implementation	102
6.3	StatusInterface Design and Implementation	102
6.3.1	Choosing Storage Type	106
6.4	Frontend Design and Implementation	106
6.5	Chapter Summary	107

7 Discussion	109
7.1 Validating the Solution	109
7.1.1 Functionality Acceptance Test	110
7.1.2 Performance Acceptance Test	110
7.2 Security	115
7.3 A RelayService Without Cloud	116
7.4 Consistency, Availability, Partition-Tolerance and Scalability	117
7.5 Evaluating Solution	119
8 Conclusion	121
8.1 Findings	121
8.2 Overall Conclusion	123
8.3 Future Work	124
A Use Cases	125
A.0.1 Use Case: Authenticate 2250	126
A.0.2 Use Case: Authenticate Customer	127
A.0.3 Use Case: Register 2250	128
A.0.4 Use Case: See 2250 Status	129
A.0.5 Use Case: Unstable Network	129
B Screenshots of StatusInterface text/html Representation	131
C Unit and Integration Tests	135
D Performance Test Data	141
Bibliography	145

CHAPTER 1

Introduction

This chapter will focus on the background of the company Brüel & Kjær and the company history regarding environmental management solutions. It will describe the hardware which are integrated parts of these environmental management solutions, namely the hand-held analyser type 2250 and the noise monitoring terminal. A vision which describes the long term goals of this project will be given. The problems Brüel & Kjær faces in regards to fulfilling the vision will be identified. Scenarios will be introduced to further quantify the problem and a thesis definition will be given to describe how this thesis aims to solve the general problem in regards to fulfilling the vision. A methodology section will describe the development methodology and the modelling standard. Finally an outline section will describe the further content in this dissertation.

1.1 Brüel & Kjær

"Brüel & Kjær Sound and Vibration Measurement A/S supplies integrated solutions for the measurement and analysis of sound and vibration. As a world-leader in sound and vibration measurement and analysis, we use our core competences to help industry and governments solve their sound and vibration challenges so they can concentrate on their primary task: efficiency in commerce and administration."[\[11\]](#)

The company has a rich history. It was founded in 1942 by Per Vilhelm Brüel and Viggo Kjær. After a slow start the company flourished up until 1992 where it was sold to a German holding company, due to financial problems. The company was split into six, Brüel & Kjær Sound and Vibration Measurements A/S (the core sound and vibration market), Brüel & Kjær Vibro (machinery condition monitoring), B-K Medical (ultrasonic medical diagnostic instruments), Innova Air Tech Instruments A/S (gas analysis instrumentation), and Danish Pro Audio (studio microphones). I will use Brüel & Kjær or B&K to describe Brüel & Kjær Sound and Vibration Measurements A/S. [21]

In 2009 Brüel & Kjær bought the Australian company Lochard, the global leader in supplying environmental management solutions for airports. B&K Environmental Management Solutions (EMS) combines noise and climate measurements in airports and cities with the purpose of minimizing the environmental impact and to ensure that national and international regulations are met[30].

1.2 Hand-held Analyser Type 2250

In this section key aspects of the hand-held analyser type 2250 will be introduced. The 2250 is used for recording, logging and post-processing of sound or accelerometer data. The 2250 covers many areas of application and it is very versatile, both in respects of hardware as well as software[5]. It is the equivalent of a Swiss army knife of sound measurements. The hardware is shown in figure 1.1. From an user interaction perspective the following objects are present:

- touch screen
- event pushbutton: for marking events.
- navigation pushbuttons: up, down, left and right buttons.
- back/erase/exlude pushbutton: to mark data with exclude marker or erase last 5 seconds of data measurement.
- reset measurement pushbutton: for resetting the measurement.
- power switch: for turning the device on and off.
- commentary pushbutton: for attaching recorded messages to measurements.
- accept pushbutton: to accept changes.
- store pushbutton: for storing measurements.



Figure 1.1: Hardware overview of the 2250. Source: [5, p. 4]

From a connection point of view the following objects are present:

- 3.5 mm stereo socket: for connecting headphones

- USB interface: for connecting to a PC
- Output socket: output software determined signals
- Trigger input: for input that will be used as trigger
- Input: for AC/DC or CCLD signals
- CF slot: slot for inserting a Compact Flash card
- SD slot: slot for inserting a Secure Digital memory card

The device is running Windows CE version 3.0 with .Net compact framework version 3.5. The device can run a variety of different software depending on the specific application use. The 2250 can also run a Web server which enables remote access and remote control. There are two interfaces which enables the remote control, a the RemoteAPI Web service and the browser interface that looks like any other Web page. These interfaces are detailed in section [2.6 Domain Analysis](#).

1.3 Noise Monitoring Terminal

The Noise Monitoring Terminal is a stationary monitoring system used in EMS. The NMT can be placed around airports, construction sites and cities to measure sound levels and help create a better environment. The NMTs are often placed in remote locations, like the Australian desert or mounted at inaccessible places like near the top of an electrical mast. Locations like these seldom have an Ethernet connection available. The data collected is streamed via a GRPS connection to a central data center and thereafter made available to/for customers. The components in a NMT is described in figure [1.2](#) The NMT consists of two batteries, a GPS receiver, a GRPS router, a weather station, an external microphone, a 2250 hand-held analyser and a Compact Flash to LAN adapter[6]. The GRPS router model varies depending on the country where it is set up¹.

The NMT streams the data to a central data center by creating an outgoing HTTP connection. The address of this data center is manually configured on the initial setup. The customer can access this data using a platform called Noise Sentinel. It is interesting from a business perspective because the service produces recurring revenue.

¹Most routers are actually upgraded to 3G routers at the present time. The actual technology used is without consequence for this project, therefore the term GRPS router will still be used.

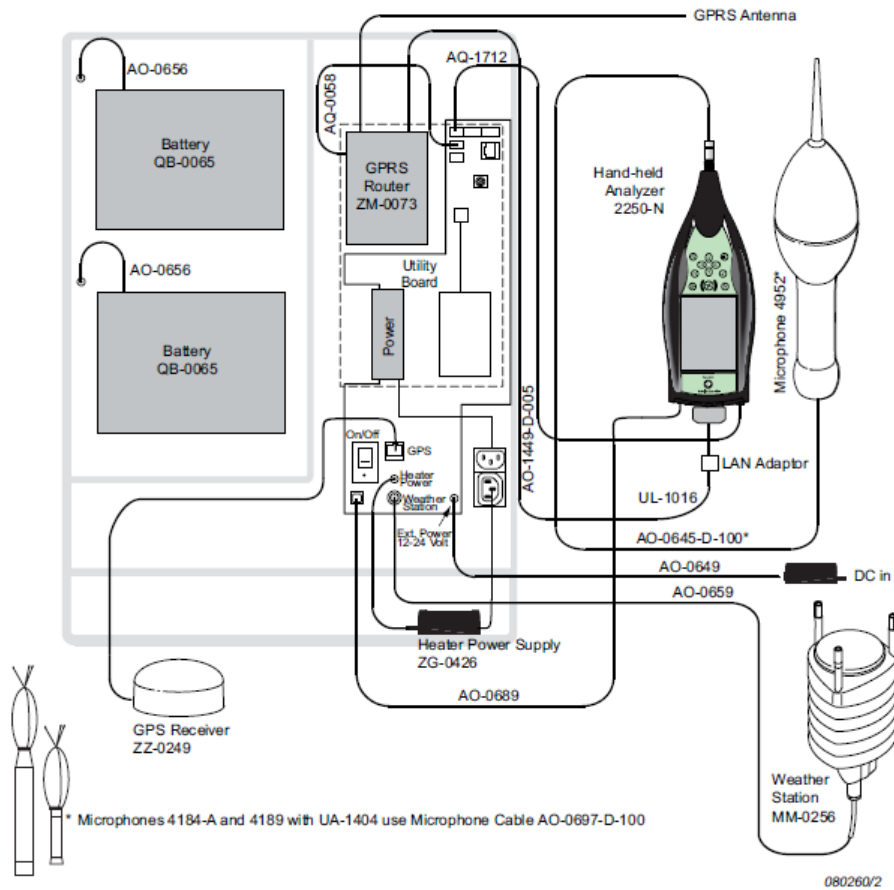


Figure 1.2: Hardware overview of the Noise Monitoring Terminal. Source: [6, p. 12]

Going to the Australian desert or up electrical masts to configure a device after the initial setup is not a viable scenario -this makes remote control of the NMT a very valuable function. To enable the remote access as mentioned in section 1.2, the GRPS router must be manually configured to forward port data. The Web server on the 2250 must be configured to start. Furthermore the NMT needs a public IP address.

The actual port forwarding setup varies depending on the router model. As a public IP address can be dynamic, dynDNS is used. dynDNS allows users to have a sub domain that points to a computer with regularly-changing IP addresses, such as those served by many Internet service providers. An update

client built into the computer keeps the hostname up to date with its current IP address. The update client does not work on all networks. On the networks where this does not work the IP address must be manually setup in dynDNS, for obvious reasons this only works when the IP address is static.

1.4 Vision

The vision of this project is to provide a proof of concept prototype where the 2250 is remote controlled without it having a public IP address. Creating devices that can be initialized and remote controlled from anywhere in the world potentially reduces the time spent on configuring NMTs by B&K staff and provides added value for customers by providing remote control and access of a 2250. This furthermore enables B&K to decouple the physical interface on the 2250 from the remote control interface enabling B&K to provide faster and better ways of remote controlling a device off-site.

These are important steps in ensuring Brüel & Kjær's position as the world leading provider of sound and vibration measuring equipment.

1.5 The Problem

For Brüel & Kjær the overall problem manifests itself in the following two sub problems. The first problem is how to provide a customer with the status of his NMTs. Right now the only "life sign" is streamed data. Beyond this you must obtain remote access to the device to see status information. This is a viable solution if the customer needs to check one or two devices, but if the customer has ten, this becomes tedious. The second problem bounds in the fact that an inbound connection on the NMT is needed for remote access to the device. The difficulty with an inbound connection is due to several issues. First of, the entire configuration is rather long and varies depending on the specific hardware setup. If any errors occurs in this setup a person must go the actual location of the device and reconfigure it. Secondly it often proves to be both troublesome and time consuming to get a public IP from the internet provider. It requires a special SIM card. In countries like Great Britain there is a long delivery time on these SIM cards, resulting in a postponed delivery date for the NMT. In other countries like Australia, the price of a public IP address is increased by the IPv4 address shortage. At several occasions the internet provider has promised public IP addresses but provided B&K with another product.

1.5.1 Scenarios

User scenarios are included to help quantify the problem and describe the motivation for and application of the proposed solution. The two scenarios below describe typical scenarios for users of the proposed solution. In [2.4 Use Cases](#) the scenarios will be used as basis for the use cases.

1.5.1.1 Scenario 1

The company Easy Airport has bought three Noise Monitoring Terminals to measure the effect they have on the local environment. Bryan from Easy Airport wants to register his NMTs to his company. Bryan logs onto the NMT control Website via a Web browser. Registers the three NMTs which are now bound to the account he logged in with. He goes to the overview and sees that two of them are online and one is not.

1.5.1.2 Scenario 2

Anders wants to remote control his NMT device. He logs on to the NMT control Website. He goes to the overview of devices picks his one device and chooses it for remote control. The device does not have a public IP address so Anders can not remote control it directly.

1.5.2 Thesis Definition

This thesis aims to solve the general problems behind the issues Brüel & Kjær are experiencing with the remote control of their 2250 devices. The starting point will be the existing RESTful remote control interface on the 2250. The thesis aims to solve this issue by combining cloud computing and RESTful Web services technology. The problem statement will be addressed by meeting the objectives defined in this section. The methodology used to develop the prototype is defined in [1.6 Methodology](#) and a outline of the thesis is given in [1.7 Outline](#) which also specifies where the specified objectives are addressed.

The objectives of this project are to:

1. Introduce the needed background information from the application domain in Brüel & Kjær.
2. Discuss the current solution and compare this to existing solutions which can be considered state of the art in access and communication with a

remote device.

3. Identify, model and prioritize requirements for the application and describe use cases realizing the requirements.
4. Introduce different types of Web services and key concepts regarding security in Web services.
5. Describe RESTful Web services in depth and security for RESTful Web services.
6. Introduce the concept of cloud computing, important concepts related to cloud computing, and security risks in cloud computing.
7. Introduce the Windows Azure platform and discuss risk mitigation.
8. Identify components in the proposed solution, survey communication solutions between these components, and design communication protocols between components.
9. Design with scalability in mind.
10. Design with extensibility and versatility in mind.
11. Implement a proof of concept prototype of the proposed solution using relevant and reasonable design patterns.
12. Validate the prototype with acceptance tests based on use cases as well as reasonable performance tests based on the non functional requirements.
13. Discuss scalability, CAP theorem, and possible restrictions and limitations related to the prototype.
14. Discuss how the thesis addresses the problem statement, how it fulfils the objectives, and future work in this area.

1.6 Methodology

The development of this project will follow an iterative process. The iterative development project starts with an initial planning. Afterwards the process will iterate through the analysis, design, implementation, test and evaluation phases a number of times until it is deemed mature for deployment. The goal of this project is to supply a proof of concept prototype. This prototype will be the final deployment in the iterative process. The analysis, design, implementation, test and evaluation will be presented in their final state in the iterative process. Therefore the progress will not be presented and it will appear as if the

development followed the waterfall method. Throughout the process design and analysis will be visualised using the Unified Modelling Language (UML).

In the initial analysis phase use cases will be defined based on user scenarios. Based on these use cases functional, non functional and runtime requirements will be defined. The overall design with focus on components and communication protocols. The communication protocols will be modelled with UML protocol state machine diagrams. The structure inside the components will be modelled with UML class diagram. The behaviour of key objects inside each component will be modelled with UML state machine diagrams.

As the primary focus of the project is a back end implementation, the following iterations will deal with refinement of functional requirements and mitigation of technical risks.

The design must describe how security can be incorporated in the solution and the implementation can implement security measures if deemed necessary from Brüel & Kjær's point of view.

1.7 Outline

This section outlines the chapters in the report, as well as gives a brief introduction to their content.

Introduction (Objectives addressed: 1)

This chapter introduces the background information for the problem statement. The vision and problem statement are defined and the methodology and outline for the project is specified.

Analysis (Objectives addressed: 2, 3)

In this chapter the existing remote control interfaces on the 2250 are analysed and the functionality compared with other state of the art solutions. The requirements are derived and use cases are specified which cover the highly prioritized requirements. Key concepts of the domain are introduced based on requirements and use cases.

Technology Analysis (Objectives addressed: 4, 5, 6, 7)

In this chapter the concept of cloud computing is clarified. Key aspects such as scalability and the CAP theorem, as well as cloud categories are discussed. Security risks in cloud computing are identified. Windows Azure as a cloud computing platform is described and relevant design options are discussed. It is also specified how Windows Azure mitigates

the identified risks.

A definition of Web services is given and the different types of Web services are introduced. RESTful Web services is chosen and described in-depth. Security risks for Web services are identified and is specified how these risks can be mitigated using RESTful Web services.

Components and Communication (Objectives addressed: 8, 9)

Components in the solution are presented. Overall communication on a abstract level is discussed and concrete communication strategies are presented. The Tunnel strategy is chosen and its protocol specified.

RelayClient (Objectives addressed: 10, 11)

In this chapter the behaviour of the RelayClient prototype is defined. The implementation, and design including key software patterns are specified.

RelayService (Objectives addressed: 9, 10, 11)

Design and implementation of the RelayService prototype are defined in this chapter. Cloud computing design options are chosen to ensure scalability.

Discussion (Objectives addressed: 12, 13)

This chapter briefly describes how the prototype design is validated using unit, integration, functionality acceptance and performance acceptance tests. Security, how the RelayService can be hosted in-house, scalability and CAP theorem applied to this project are discussed before the prototype is evaluated.

Conclusion (Objectives addressed: 14)

In this chapter it is assessed how well the vision, the project statement and the objectives of this thesis are met. Based on this an overall conclusion is provided and future work described.

2.1 State of the Art

The desire to remote control a device is not new, neither are the problems that present themselves when both controller and the device controlled are behind firewalls. This section will describe the basic functionality of the existing interface on the 2250 device and with this as basis investigate already existing solutions for communicating with non addressable devices and remote controlling devices. A functionality overview of all the solutions will be given in [2.1.5 Functionality Overview](#). In this section remote controlled devices will be referred to as hosts, remote controllers as clients and any mediator between as gateway.

2.1.1 RemoteAPI and Browser Interface

The existing software for remote controlling a 2250 consists of the Browser Interface and a remote control interface (RemoteAPI). They are both available via the Web server on the 2250. The Browser Interface allows a client to see and control the running GUI on the host. Both interfaces are available using TCP and the HTTP protocol. Authentication is based on the HTTP Basic Access Authentication standard[20] and the communication is not encrypted. The communication between client and host is direct and it is therefore required that the client can make a connection directly to the host. As mentioned in the

previous chapter this requires a public IP address for the host. The existing solution offers no solution for seeing an overview of devices or seeing the status of device if the device is unavailable.

RemoteAPI and Browser Interface Functionality Summarized

Expose GUI

Functionality exposes the GUI on the host to the client. Thereby the client is able to interact with the host the same way as if he was using it on site.

Direct connection

The solution functions when a direct connection can be achieved between the client and the host. This allows data to flow directly from client to host without having to channel it through a gateway.

Expose Web applications

Functionality exposes the Web server and the running Web applications to the client. The client is able to interact with these applications as with any other server.

Extensible

The solution can be combined with custom application running on the host, ensuring versatility for specialized hosts. In this project this is important so that the solution can be combined with the BasicEnv¹ software and otherwise extended if need be.

Windows CE compatible

The solution can be installed and run on the Windows CE platform.

2.1.2 LogMeIn

The LogMeIn solutions are commercial products for remote controlling and file sharing. In this subsection the solution will be described from a technical perspective. It is made for hosts running Windows or Mac OS while the client only needs to run a modern Web browser. The LogMeIn setup is described in figure 2.1.

The LogMeIn host creates an outbound SSL secured connection to the LogMeIn gateway, which is placed in the LogMeIn data center. Because the connection is created by the host and is outbound the firewall will accept it like secure Web traffic. The client browser creates a connection to the LogMeIn gateway

¹B&K Basic Environmental Client software

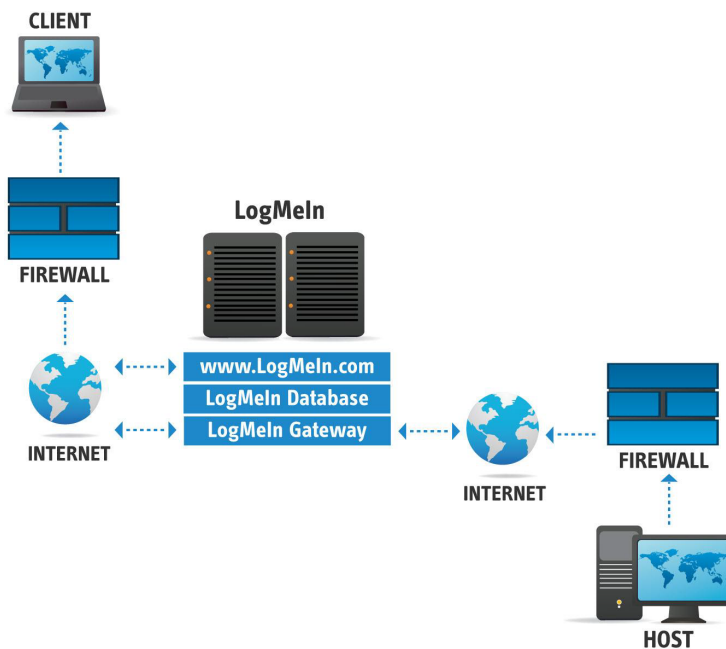


Figure 2.1: Diagram showing the LogMeIn communication flow. Source: [26].

and authenticates itself. After authentication the client will be authorized to exchange data with the hosts belonging to the user's account. The gateway forwards the encrypted data between client and host. As an extra level of security the client also needs to authenticate himself to the host. Once the client authenticates himself to the host and it authorizes his access the remote session begins[26].

As LogMeIn themselves state, there is a great benefit in using a gateway to mediate the traffic between the client and the host:

The benefit of using the gateway, instead of establishing a direct link between the client and the host, is that either the client or host (or both) can be firewalled. The LogMeIn gateway ensures that users do not need to configure firewalls.[26]

The LogMeIn service is made in such a way that it discovers if a direct connection can be made between the client and the host after authentication and if possible makes such a connection thereby reducing stress on the gateway and reducing latency. To further enable this they have also implemented a solution using the

User Datagram Protocol (UDP) protocol which is less often filtered by firewalls and thereby supports more direct connections.

LogMeIn Functionality Summarized

Gateway

The solution is built so that a gateway can facilitate traffic between the client and the host. This implementation allows a connection to be established indirectly between the client and the host when both are behind firewalls and neither can accept incoming connections.

Access without public IP address

The Gateway implementation allows data to be sent between device and host when neither can accept incoming connections. This functionality is a direct implication thereof and it states that a host without a public IP address can be communicated with.

Expose GUI

This functionality is described in [2.1.1](#)

Direct connection

This functionality is described in [2.1.1](#)

Confidentiality

Data transmitted between client and host is encrypted and confidential.

Host Overview

This service allows you to see an overview of the hosts available for a specific client and see if a particular host is online.

Info even when offline

Functionality allows a client to see status information about a host even if it is offline. This requires a gateway or another central service which stores this data.

2.1.3 Microsoft Push Notification Service

In this subsection a state of the art smart phone communication method will be described.

The Microsoft Push Notification Service in Windows Phone offers third-party developers a resilient, dedicated, and persistent channel to send data to a Windows Phone application from a Web service in a power-efficient way.[\[31\]](#)

Many modern smart phone applications have a client running on a phone and a server part often running as a service in the cloud. To be able to push data from the server to the client the push notification services were created. The Microsoft Push Notification Service solutions consists of four components a client application, a push client service, the push notification service and the cloud service.

Figure 2.2 shows how the client application running on the phone can request a push notification URI from the Push Client Service (1). The Push Client Service then negotiates with the Microsoft Push Notification Service (MPNS) and returns a notification URI to the client application (2 and 3). The client application can then send the URI to the cloud service (4). When the Web service has information to send to the client application, it uses the URI in sending a push notification to the Microsoft Push Notification Service (5), which in turn routes the push notification to the application running on a Windows Phone² device (6)[31].

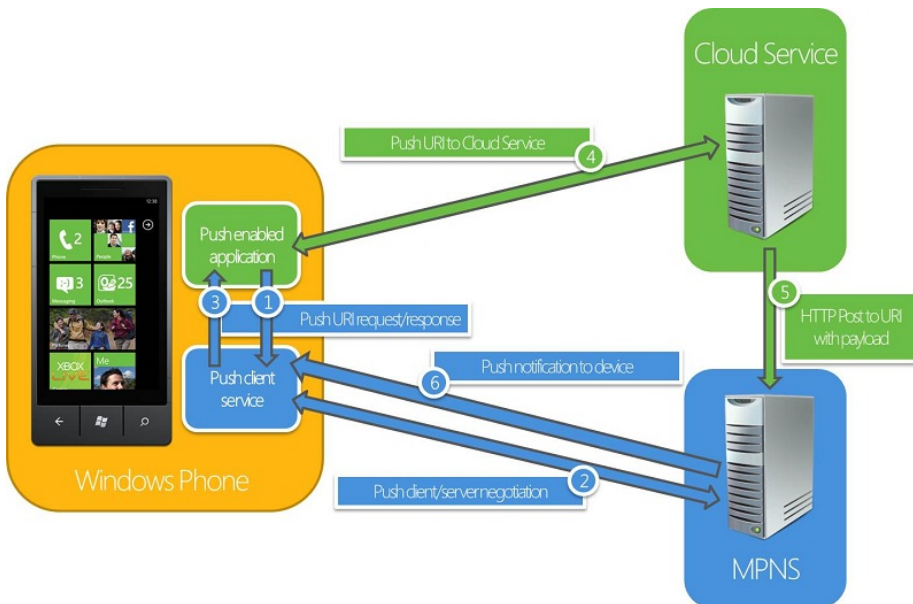


Figure 2.2: Diagram showing the Microsoft Push Notification Service structure. Source: [31]

²Push Notification is also available for Windows 8 applications

The Push Notification provides a way of sending notifications to the host from the client, but it does not provide the functionality³ nor flexibility⁴ to function as the sole means of communication and send requests and receive responses. Therefore additional communication for this must be incorporated into the solution. It is worth noting that the Microsoft Push Notification Service is not unique, similar services exist for Apple and Android devices.

Microsoft Push Notification Service Functionality Summarized

Gateway

This functionality is described in [2.1.2](#)

Access without public IP

This functionality is described in [2.1.2](#)

Extensible

This functionality is described in [2.1.1](#)

Confidentiality

This functionality is described in [2.1.1](#)

2.1.4 Azure Service Bus and Windows Communication Foundation

The Azure Service Bus is a part of the Microsoft Azure cloud platform which will be detailed in section [3.2](#). In combination with the Windows Communication Foundation (WCF), detailed in [3.3.3 Windows Communication Foundation](#), the Azure cloud platform offers the Service Bus Relay. The Service Bus Relay enables a local service running in an enterprise environment behind a firewall to connect to the Azure Service Bus and expose its interface. The exposed interfaces will be available to specific consumers or to anyone, depending on the settings. This effectively allows a service to circumvent any firewall and to expose a service without a public IP address. The communication between the Service Bus and the service is handled by the WCF framework and is seamless for the publisher when configured. It does however require that the service exposed is a WCF service.[\[40\]](#) It is worth noting that WCF does not run on the Windows CE platform and that the Service Bus Relay does not per default give an overview if a service is online or offline.

Azure Service Bus and WCF Functionality Summarized

³Push Notification only allows information flow from Client to Host.

⁴There are size limitations on notifications (currently 1KB in header 3KB in payload).

Gateway

This functionality is described in [2.1.2](#)

Access without public IP

This functionality is described in [2.1.2](#)

Expose Web applications

This functionality is described in [2.1.1](#)

Extensible

This functionality is described in [2.1.1](#)

Confidentiality

This functionality is described in [2.1.1](#)

2.1.5 Functionality Overview

An overview of the functionality offered by the different solutions can be seen in table [2.1](#). The table marks what functionality is offered by what solution. The functionalities listed are the union of all the functionalities offered. From the overview it is clear that none of the existing solutions will be satisfactory. One of the most obvious reasons for this is that only the existing B&K solution runs on Windows CE. This also means that even though some of the solutions are Extensible they cannot be used. The existing B&K solution however lacks other functionality e.g the *Info even when offline* functionality. This is a requirement from B&K which will be identified in the next section together with the additional functional and non-functional requirements.

2.2 Functional Requirements

One of the main problems in this project is how to connect two instances where neither accepts incoming connections. How can two people communicate if they do not have phone numbers and their phones are only able to make outgoing calls. If they have a common friend with two phones and phone numbers they can call him. The common friend can hold the two telephones up against each other and now a conversation can take place. The concept of introducing a common friend who mediates communication is also the core concept in the Log-MeIn, Microsoft Push Notification Service, and Azure Service Bus and Windows Communication Foundation solutions. The functional requirements will build upon this design choice. The functional requirements are furthermore inspired by the existing solutions and based on communications with B&K employees as

⁵The Push Notification can give status information about the device, such as if it is idling, the screen is turned on etc. However this is only if the device is offline and it is not the status info desired by the users in our scenarios

Functionality	Existing Solution	LogMeIn	Push Notification	Service Bus WCF +
Gateway		x	x	x
Direct connection	x	x		
Confidentiality		x	x	x
Access without public IP address		x	x	x
Expose GUI	x	x		
Extensible	x		x	x
Expose Web applications	x			x
Info even when offline		x	(x) ⁵	
Host overview		x		
Windows CE compatible	x			

Table 2.1: Use cases versus requirements.

well as the user scenarios. The functional requirements presented, their prioritization and the non-functional requirements specified in the following chapter are for the proof of concept prototype and not a final product. The functional requirements are categorized based on their priority into the two categories *must have* and *nice to have*.

Must have requirements (MH)

- MH1. Solution must work when neither customer nor 2250 accept incoming connections. This must be done by facilitating communication via a central RelayService to RelayClient software present on the 2250. This requirement is the combination of the *Gateway* and *Access without public IP address* functionality described in the previous section.

- MH2. RelayService must expose the Web applications running on the 2250 Web Server, including RemoteAPI and the Browser Interface as is. This ensures the *Expose GUI* and *Expose Web applications* functionalities as they are present in respectively the Browser Interface and RemoteAPI. The further restriction that the remote control interfaces are exposed as is, ensures backwards capability with existing client software, consuming either of the two interfaces. This is a further requirement of B&K.
- MH3. Customer can see current status of a 2250 device or last available status. Fulfilling this requirement realizes the *Info even when offline* functionality. This allows to troubleshoot problems better and to ensure information is available even if the device is temporarily not reachable.
- MH4. Customer must be able to see overview of all the 2250's he has registered. This corresponds to the *Host overview* functionality described in the previous section.
- MH5. Customer must authenticate to 2250 with an existing User Account when accessing its interfaces. This functionality protects 2250 devices from unauthorized access.
- MH6. The device must authenticate to the RelayService. This protects the solution from malicious users posing as a 2250 and also serves to identify the individual devices.
- MH7. RelayService should always give a response. This ensures that the customer will get a response within a reasonable time period even if the device is offline. This is a usability requirement and can be related directly to the non-functional NFR3 requirement.
- MH8. Customer must only see status of 2250 that are registered to him. This ensures that status information is not available to other customers or malicious users.
- MH9. Customer must authenticate to RelayService before he can see overview of devices or register any devices. This requirement dictates that the customer must authenticate a necessity for the authorization needed in MH4 and MH8.
- MH10. The solution must be resilient to fault at and offline periods for the RelayService and RelayClient. The RelayClient and RelayService should constantly seek states where relay of requests is possible.

Nice to have requirements (NH)

- NH1. Customer is able to register a unregistered 2250.

- NH2. RelayService and 2250 must be able to process multiple remote control requests. Not necessarily in parallel.
- NH3. RelayService must be able to handle multiple active customers.
- NH4. RelayService must be able to handle multiple active 2250.
- NH5. Communication is encrypted. This maps to the *Confidentiality* functionality described in the previous section.
- NH6. Customer will automatically switch to direct control of a device if this is possible. This maps to the combination of the *Gateway* and *Direct connection* functionality only offered by the LogMeIn solution.

2.3 Non-Functional Requirements

The non-functional requirements are based on communication with B&K and are based on the target platform, the application use, as well as B&K's business model.

- NFR1. Any new software on the 2250 must run on Windows Compact Edition with .Net Compact Framework 3.5 like the existing software. This maps to the *Windows CE compatible* functionality described in the previous section.
- NFR2. Brüel & Kjær is not in the data center business and therefore require that any gateway or central service introduced will be hosted elsewhere.
- NFR3. Additional latency introduced by our system must be minimized. Prototype criteria: round trip time must be less than a 5 second increase compared to direct remote controlling.
- NFR4. Minimum of network traffic overhead. Limited bandwidth available. Must take up as little as possible of the bandwidth. Prototype criteria: it must be able to co-function with NMT streamer software
- NFR5. Any new software on the 2250 must have minimal CPU and memory usage. Prototype criteria: it must be able to co-function with the BasicEnv software.
- NFR6. Prototype must be backwards compatible ensuring that no functionality of the device is compromised by the RelayClient software. Prototype criteria: it must be able to co-function with the BasicEnv software.

2.4 Use Cases

The use cases introduced in this section serve the purpose of further detailing the user scenarios presented in 1.5.1 *Scenarios*. The aim of these use cases is to quantify the work flows in which the functional requirements are realized and use this to substantiate the functional requirements. These use cases will in subsection 2.6 be used to map important concepts in the domain. The use case diagram in figure 2.3 shows the seven use cases, their relations and the two actors, the customer and the 2250. Three use cases will be included in this

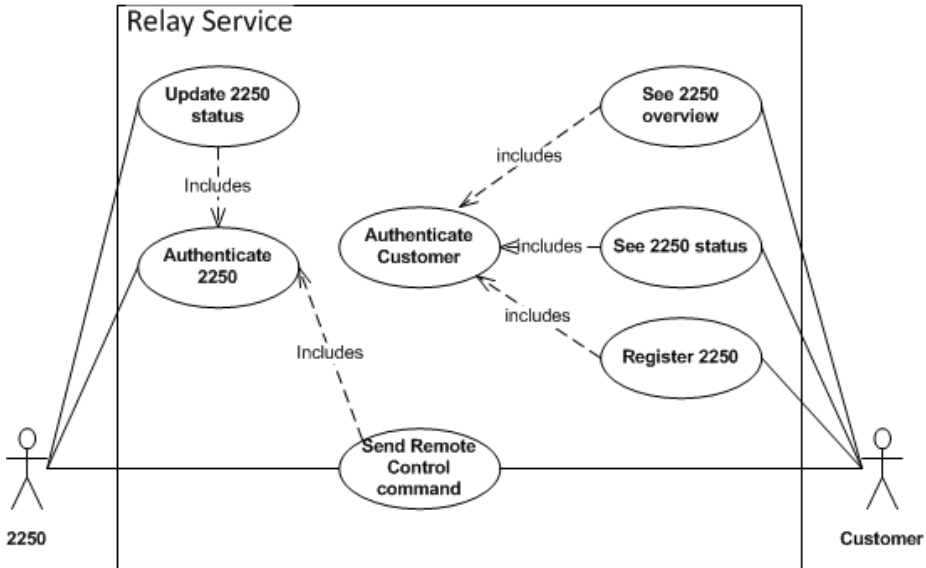


Figure 2.3: Use case diagram

section namely *See 2250 overview*, *Update 2250 Status* and *Send Remote Control Command* while the remaining four use cases *Authenticate 2250*, *Authenticate Customer*, *Register 2250*, and *See 2250 Status* will be included in the appendix and only described textually in this section. The use cases will be described using the common use case style described in Fowler, p. 101[18], as it offers a suitable level of complexity.

See 2250 overview	
Actor	Customer
Description	See overview of the 2250 devices registered to a specific customer.
Precondition	Customer is authenticated.
Postcondition	None.
Main path	<ol style="list-style-type: none"> 1. Customer requests to see 2250 overview. 2. RelayService receives request. 3. RelayService returns overview info about the 2250 devices registered to the customer. 4. Customer receives overview.

Update 2250 Status	
Actor	2250
Precondition	2250 is authenticated.
Postcondition	2250 status is updated or no update acknowledgment is received by the 2250.
Success End Condition	2250 status is updated and saved in the RelayService.
Failed End Condition	2250 status is not updated in the RelayService.
Main path	<ol style="list-style-type: none"> 1. RelayService sends update request to 2250. 2. 2250 receives update request. 3. 2250 sends 2250 status information. 4. RelayService receives status information. 5. RelayService saves status information.
Extensions	<ol style="list-style-type: none"> 2a 2250 does not receive update request. 4a RelayService does not receive status information. 4a1 RelayService sends update request to 2250.

Send remote control command	
Actor	Customer, 2250
Precondition	None.
Postcondition	None.
Main path	<ol style="list-style-type: none"> 1. Customer sends one or more remote control request to RelayService. 2. RelayService receives one or more remote control requests. 3. RelayService verifies that the 2250 is reachable. 4. RelayService forwards the received remote control requests to RelayClient on the 2250. 5. The RelayClient receives one or more remote control requests. 6. The RelayClient forwards the request to the 2250 Web server and sends a remote control response for each request. 7. RelayService receives remote control responses. 8. RelayService forwards the remote control responses. 9. Customer receives remote control responses.
Extensions	<ol style="list-style-type: none"> 3a The RelayClient and the 2250 are not reachable. 3a1 The RelayService notifies the customer that the 2250 is not reachable. 5a The RelayClient does not receive remote control request. 5a1 RelayService notifies Customer that remote control request could not be delivered. 7a RelayService does not receive remote control responses. 7a1 RelayService notifies customer that the 2250 is not reachable. 9a Customer does not receive remote control responses.

Authenticate 2250

This use case is included in [A.0.1 Use Case: Authenticate 2250](#). In this use case the 2250 authenticates itself to the RelayService which validates the credentials and informs the 2250 of the outcome.

Authenticate Customer

This use case is included in [A.0.2 Use Case: Authenticate Customer](#). In this use case the customer authenticates himself to the RelayService which validates the credentials and informs the customer of the outcome.

Register 2250

This use case is included in [A.0.3 Use Case: Register 2250](#). In this use case the customer registers a ownership of a 2250 at the RelayService.

See 2250 Status

This use case is included in [A.0.4 Use Case: See 2250 Status](#). In this use case the customer chooses a single 2250 and sees its status. Preconditions are that the customer is authenticated and the 2250 is registered to him.

2.5 Use Case Coverage

Use case coverage describes how well the existing use cases cover the specified requirements. Having a high coverage also means that the use cases can be translated into acceptance tests which will be satisfying to prove program functionality. The use case coverage is described in table [2.2 Use Cases versus Requirements](#).

2.6 Domain Analysis

In this section key concepts in the domain which are relevant to this project are identified. The concepts are identified based on the functional requirements in section [2.2](#), the non-functional requirements in section [2.3](#) and the use cases in section [2.4](#). These descriptions help map the domain, disambiguate terms and aid the development process later in this project. An overview of these key concepts is available in figure [2.4](#).

2.6.1 Device/2250

The 2250 is a physical piece of hardware. It is versatile and exist in many application contexts. The definition given here is only sufficient for a 2250 used in

Requirement	Authenticate 2250	Update 2250 Status	Authenticate Customer	See 2250 Overview	See 2250 status	Register 2250	Send Remote Control Command
MH1							x
MH2							x
MH3		x			x		
MH4				x			
MH5							x
MH6	x						
MH7							x
MH8				x			
MH9			x				
MH10	x						
NH1						x	
NH2							x
NH3							
NH4							
NH5							
NH6							

Table 2.2: Use Cases versus Requirements

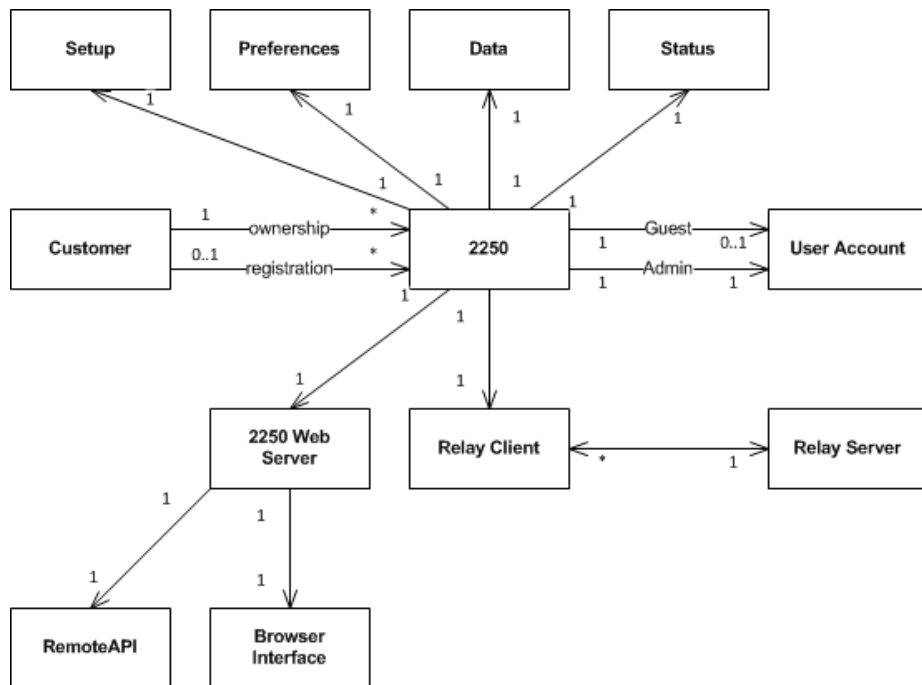


Figure 2.4: Diagram showing an overview of the domain

environmental solutions. They are uniquely identified by a serial number and are sold by B&K to customers. They can be connected to a Weather Station to detect pressure and/or a GRPS router to provide a wireless internet connection. In other systems they may be connected to different pieces of interesting hardware. The operating system for the 2250 is the Windows CE. When the 2250 is used in a Noise Monitoring Terminal it runs the NMT Client software, this software runs on top of the B&K Basic Environmental Client software BasicEnv. Remote access to a device happens via two BasicEnv Web application which are exposed via the Windows CE inbuilt Web server.

The BasicEnv Web applications are exposed via the Web server built into the Windows CE OS. When the Web server exposes these interfaces it will be referred to as the 2250 Web Server. In any state the device has Status, Setup, Preferences, Data and Commands. Access to the device is controlled via User Accounts. The device will always have an Admin User Account and may have an Guest User Account.

2.6.2 Customer

A customer is a person or company that owns one or more 2250s. It is the customer who accesses and remote controls a device. B&K has no other access than the one granted by the customer so if a B&K employee is accessing a device it is on behalf of a customer with the customers credentials. Therefore B&K employees do not represent an individual actor.

2.6.3 Registration

A registration is a relationship between a customer and a 2250. A customer can register a 2250 by notifying and proving this ownership to B&K.

2.6.4 2250WebServer

The exposed interfaces are offered through combination of the Web server and the BasicEnv Client software. To reduce complexity, the combination of the exposed interfaces and the Web server will be modelled as a single component named the 2250 Web Server which provides two interfaces as illustrated in figure 2.5.

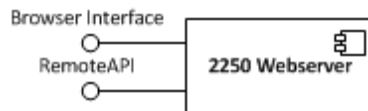


Figure 2.5: The 2250 Web Server component

The two interfaces provide ways of remote accessing and controlling the device, one is the Browser Interface enabling remote control via a normal browser, the second is the RemoteAPI a REST interface enabling remote control via a custom software client.

2.6.5 User Account

The term User Account covers a user present on the 2250. There are two user accounts per default, the admin user account and the guest user account. These user accounts govern access and authorization to the two interfaces.

2.6.6 Preferences

The preferences is a group of settings used to configure system, hardware and network settings. Below is an excerpt of settings groups in Preferences.

Power Settings These settings configure the power saving settings for the device such as when to turn off the backlight, when to dim the backlight and when to enter standby mode.

Regional Settings These settings configure settings that varies depending on the region such as decimal point, date separator, date format, time zone, language and keyboard layout.

Storage Settings These settings configure automatic naming of projects as well as project prefixes.

Users Configures whether multiple users are allowed.

Modem/DynDNS Settings These settings configure the modem and DynDNS. The DynDNS relates to the DynDNS settings described in [1.3 Noise Monitoring Terminal](#).

Network Settings These settings configure/shows network information such as DHCP attributes, DNS attributes, MAC address and status.

Web Server Settings The settings configure the Web server status and user accounts.

2.6.7 Setup

Setup defines the measurement setup. The measurement setup is primarily concerned with acoustic settings. Understanding sound values and parameters is beyond the scope of this project. The settings are however included to give a quick overview for the interested reader. The setup is split up into the following categories.

Input Input defines the microphone input. Which socket to retrieve data from, the transducer used, triggerinput etc.

Frequency Weightings Defines how to weight broadband exclusive peak and broadband peak.

Statistics Defines what to base statistics on.

Measurement Control Defines whether a measurement should be controlled manually or automatic.

Signal Recording Defines control attributes for a recording, such as minimum duration, maximum duration, peak recording level, etc.

Output Socket Signal Defines whether to output a signal on a socket and which signal to output.

Occupational Health Defines threshold level peaks over time such as heat.

2.6.8 Data

The Data contains attributes representing the ongoing measurements. The data is volatile as the measured data varies over time.

2.6.9 Status

The status of a 2250 is a collection of attributes that convey information that is persistent over longer periods of time and describe the system and network status. These attributes will be a subset of the attributes found in Preferences in the RemoteAPI. The status can, because of its persistent constraint, be stored on a remote location and only updated when the status information is changed on the 2250. The status attributes of a device is described in table 2.3.

Attribute Name	Attribute Description
ModelVersion	Firmware version of the NMT
LANIPAddress	Local IP Address of the device
LANMACAddress	MAC address
LANSubnetMask	Local network subnet mask
InstrumentType	Type of instrument (in this case a 2250)
Device	Serial number for the device
Online	Describes whether a 2250 is online and accessible. This is redundant information when you are accessing the device and it is therefore not included in the RemoteAPI Preferences.

Table 2.3: Attributes describing the status of a device

2.6.10 Browser Interface

The Browser Interface is accessible via any modern Web Browser. It is used for remote access to the device and mirrors the actual UI on the device. Figure 2.6

shows a screenshot of the Browser Interface. In the right side a series of buttons are placed which allows the customer to perform certain standard actions. These buttons are copies of the pushbuttons described in section 1.2. To the left a 240x320px image is placed. This image is a screenshot from the actual device. As it is a screen shot the content of it will vary depending on the software and software state.

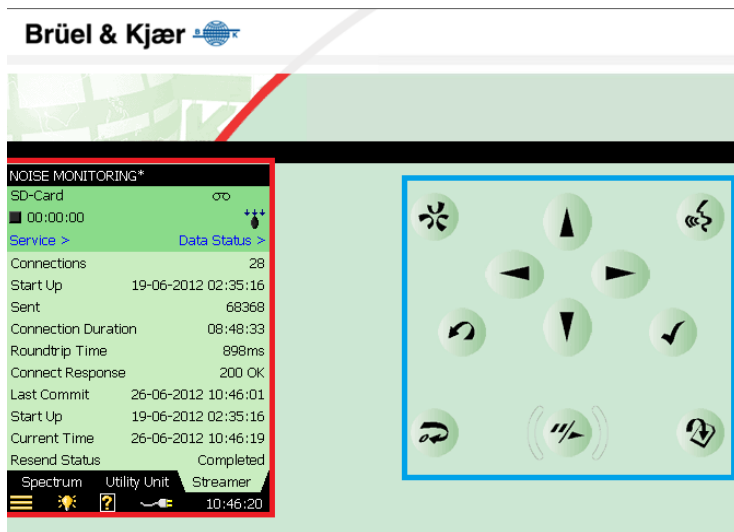


Figure 2.6: Browser Interface UI

The entire interface is built on AJAX. Whenever a fresh screenshot is received a new one is requested from the server. This potentially adds up to a picture being loaded every half second. All interactions are sent via HTTP POST or HTTP GET requests to the Web server. Clicks on screen include the coordinates and will simulate a physical user click on the screen.

2.6.11 RemoteAPI

The RemoteAPI is accessible on the 2250WebServer via the path "/RemoteAPI". The RemoteAPI strives to be a REST interface. It is composed of four components: Preferences, Setup, Data and Commands. Each component contains a number of attributes which are all direct descendants of the component. This means that all attributes are addressed via the following hierarchical path structure RemoteAPI/<component>/attribute. The following list describes the four components

RemoteAPI Preferences The Preferences resource consist of all the attributes in display settings, power settings, regional settings, storage settings, head-phone settings, users, printer settings, modem/DynDNS settings, network settings and Web server settings, described in subsection 2.6.6. Because of the flattened structure the individual categories have been removed and all attributes are grouped together.

RemoteAPI Setup The Setup resource consist of all the attributes in input, frequency weightings, statistics, measurement control, signal recording, output socket signal and occupational health described in subsection 2.6.7. Because of the flattened structure the individual categories have been removed and all attributes are grouped together.

RemoteAPI Data The Data resource contains attributes of the type mentioned in 2.6.8 Data.

RemoteAPI Commands The Command component contain a varying number of attribute depending on the specific application running on the device and its state.

2.6.12 RelayService

In this solution communication is facilitated through a central mediator named RelayService. Requests will be made to the RelayService functions as a reverse proxy. The request will be forwarded to the 2250 and the response will be sent from the 2250 to the RelayService and then forwarded to the customer. The indirect access via the RelayService combined with the requirement to expose the existing interface as is, implies that the RelayService must expose the Browser Interface and the RemoteAPI interface. The requirement for a customer to see the status of a device will be realized with a status interface (StatusInterface). This interface should also be available on the RelayService so that it is accessible from anywhere.

2.6.13 RelayClient

The additional software on the 2250 used for communicating with the RelayService will be referred to as the RelayClient. The RelayClient is a part of the 2250, but is not contained in the 2250 Web Server, but should instead be regarded as a separate component.

2.7 Mockups

The new functionality described in the functional requirements demands an extended user interface. In this project the extended user interface are offered by the RelayService either as a Web service or as a Web page. This user interface is described in this section using a state machine to describe the interaction flow and several mockups showing the user interface in different states of the interaction. Providing mock ups in the early face of a project allow users and key stakeholders to give an input and/or validate the given interface. The five states of the user interface are the Sign In, the Device Overview, the Device Status, the Register Device and the existing Browser Interface.

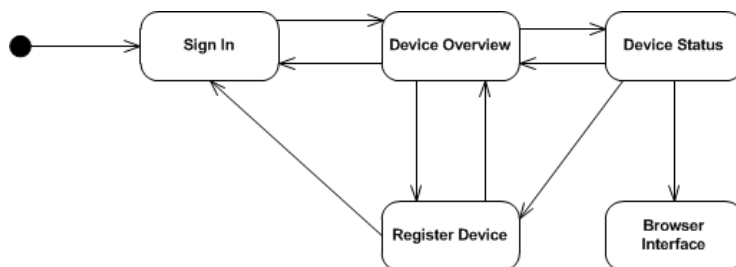


Figure 2.7: User Interface flow

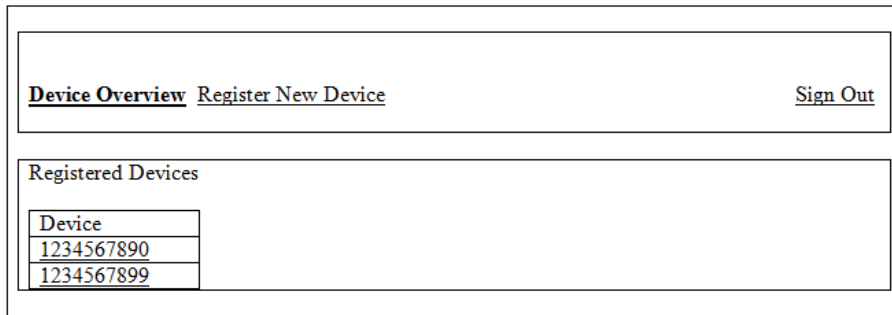
As figure 2.7 illustrates, the interaction starts at the Sign In state. The customer must sign in before he can access any of the other functionality as described in requirement MH9. This will be done with a username and a password as shown in figure 2.8.

Username: _____
 Password: _____
Sign In

Figure 2.8: Sign in mock up

When the customer is signed in he reaches the device overview page illustrated in figure 2.9. In this page the customer can see an overview of all his registered devices. It is possible for the customer to navigate to the register device page, to sign out of the system or to click on a device to see its status.

In the Register Device page shown in figure 2.10 the customer registers a device

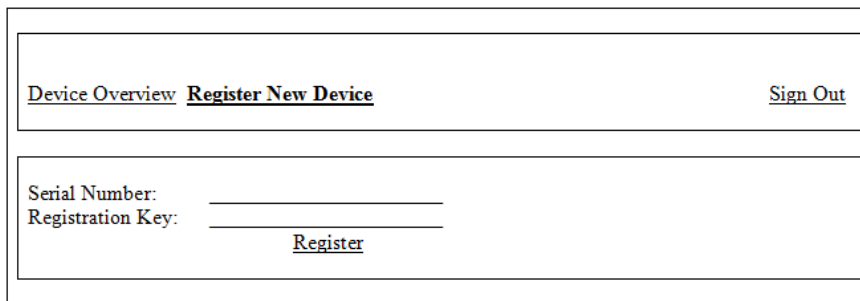


The mockup shows a header bar with two links: [Device Overview](#) and [Register New Device](#). On the right side of the header bar is a [Sign Out](#) link. Below the header bar is a section titled "Registered Devices" which contains a table with two rows of device information.

Device
1234567890
1234567899

Figure 2.9: Device overview mock up

by typing the serial number and the registration key. When the registration is complete the customer returns to the Device Overview page. The customer can at any time choose to abort the registration and sign out or go to the Device Overview page.



The mockup shows a header bar with two links: [Device Overview](#) and [Register New Device](#). On the right side of the header bar is a [Sign Out](#) link. Below the header bar is a form with two input fields: "Serial Number:" and "Registration Key:". Below the "Registration Key:" field is a [Register](#) button.

Figure 2.10: Register device mock up

In the Device Status page show in figure 2.11 the customer can see the device status as defined in 2.6.9. From the device status the customer can choose to navigate to the Browser Interface, the Register Device page or the Device Overview page.

The Browser Interface is the already existing Browser Interface introduced in 2.6.10 [Browser Interface](#). A screenshot showing the Browser Interface is shown in figure 2.6.

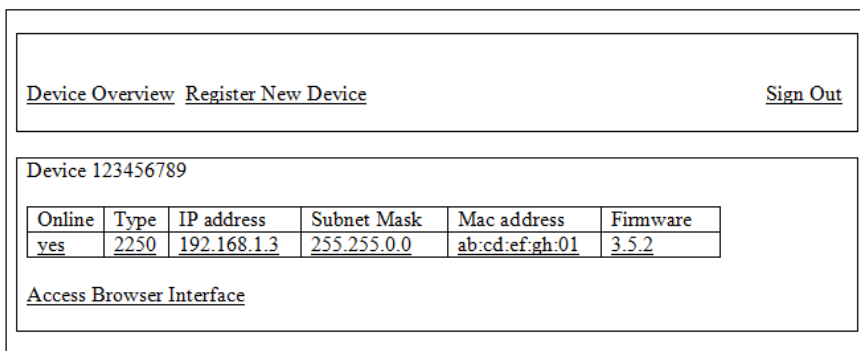


Figure 2.11: Device status mock up

2.8 Chapter Summary

In this chapter functionality of existing solutions have been identified and compared. Due to lack of support in Windows CE the only usable solution is the existing B&K software, which will be extended. Functional and non-functional requirements have been identified based on the functionality of existing solutions, user scenarios and communication with B&K. Use cases have been made which realize the requirements and which will serve as basis for acceptance test. To ensure coverage a table with requirements versus use cases have been made. Based on the requirements and the use cases main concepts in the domain have been identified and described, thereby providing the necessary background information about this specific domain to understand the further design. Based on the use cases and with the domain analysis as reference, mock ups have been made to show the desired user interface.

CHAPTER 3

Technology Analysis

In this chapter the main technologies used in this project will be described in depth. These technologies will serve as the foundation for the solution design. The concept of cloud computing will be clarified, the main implications will be described, and the security risks related to cloud computing specified. An overview of the relevant subjects in the Microsoft Azure cloud platform will be given. The concept of Web services will be introduced and the different types of Web services discussed. It will justify why RESTful Web services are chosen and RESTful Web services will be described in depth.

3.1 Cloud Computing

A central service like the RelayService needs a place to be hosted. It is evident in NFR2, that B&K is not interested in hosting a service themselves. One of the popular ways of hosting, and the one used by B&K, is cloud computing. In the following section cloud computing will be discussed. The following terminology will be used to describe the actors in cloud computing.

Cloud provider

The provider of a cloud platform and the data center in which it runs.

Cloud consumer

The consumer of the cloud platform. The Cloud consumer uses cloud

computing to host one or more services. In the current scenario this will be B&K

Cloud user

The user of a cloud service. In theory it does not matter to a user whether a service is hosted in the cloud or somewhere else. In the current scenario this will be the customer described in 2.6.2.

The concept of cloud computing is open for interpretation and the term can cover anything from simply outsourcing your hardware to the solution to all a company's worries. From a hardware perspective Cloud Computing is offering utility computing. To really understand the benefits of utility computing one must first understand the troubles of conventional hosting.

When supplying server utility for a company's service the keyword is provisioning. The usage of a service is seldom evenly distributed throughout a week or even a day. Instead a service experiences peak periods with intense usage while the usage will be low the majority of the day. A company must choose a provisioning strategy for supplying server resources for the service. The two options available are underprovisioning and provisioning for peak periods.

Underprovisioning means having less server capacity than necessary. In best case scenario this results in an unavailable service to some of the users, of them a fraction will not return as users. This can be quite devastating for a company as their reputation suffers and they potentially lose many users[1].

Provisioning for peak periods means having server capacity enough to handle all users at the busiest moments. This leads to a low average utilization of the servers as the peak period often represents a limited time period. This strategy is further complicated by user growth. If the user base grows new hardware must be purchased. Delivery time could be weeks, this means that company must predict the user growth several days ahead. For new services this can be quite hard as services can experience rapid growth and rapid decline. In contrast to conventional hosting utility computing has the three advantages described below[1].

- It provides the illusion of unlimited resources. For the consumer of utility computing there are unlimited resources. It is possible to scale up fast, thereby eliminating the need to plan far ahead for provisioning.
- It eliminates the need of initial investment. Companies no longer need to invest in hardware upfront, but can instead start small and gradually increase capacity as needed. In business language this is known as a shift from capital expenditure to operation expenditure.

- Pay as you go. This enables scalability as a consumer can pay for a large number of resources for a short period of time and then scale down afterwards.

All three benefits either reduce cost or enable scaling, but scalability is more than provisioning resources it is also scalable software. The remaining sections will discuss cloud computing from a software perspective. From a software and application perspective the important concepts in cloud computing are the cloud computing categories, the framework supplied by the cloud provider, the ability to accommodate the possibilities in utility computing and the new risks introduced by cloud computing.

3.1.1 Cloud Computing Categories

Cloud computing is categorised depending on the level of abstraction the cloud provider provides for the cloud consumer. In this subsection the different categories will be described. The choice of cloud computing category will be made in [3.1.4 Choice of Cloud Computing Category and Cloud Provider](#) based on these descriptions.

3.1.1.1 Infrastructure-as-a-Service

Infrastructure-as-a-Service (IaaS) abstracts CPU, memory and storage, offering these to the consumer. It is the lowest level of abstraction in the cloud. The cloud provider manages the physical resources and supply virtual instances of operating systems to the cloud consumer. The consumer is given full ownership of the virtual resources and he can configure them as he sees fit. This results in better control and more flexibility for the cloud consumer but also require more administration^[41]. This is very similar to conventional hosting from a software perspective. Examples of IaaS cloud providers are Amazon EC2¹ and Rackspace².

3.1.1.2 Platform-as-a-Service

Platform-as-a-Service (Paas) is the step up from IaaS and provides the next level of abstraction for the cloud costumer. Here the cloud provider also manages a service oriented application infrastructure, which provides the cloud consumer with the essential building blocks needed in his own service. This infrastructure

¹<http://aws.amazon.com/ec2>.

²<http://www.rackspace.com/cloud>.

can contain access control, messaging services, load balancing etc. The cloud consumer can use these basic building blocks to compose his own service.

When using PaaS you lose flexibility and control over the IT configuration and the software environment as this is provided and handled by the cloud provider. This also means that the cloud consumer can abstract away from maintenance, patching operating systems and software etc. as this is handled by the cloud provider[41]. In other words the PaaS can be regarded as a means of deploying applications without dealing with the necessary server configuration and maintenance. Examples of a PaaS cloud providers are Windows Azure³ and Google App Engine⁴.

3.1.1.3 Software-as-a-Service

Software-as-a-Service (SaaS) is a term used beyond the scope of cloud computing. From a cloud user's perspective almost everything in the cloud is SaaS.

SaaS provides the highest abstraction layer in cloud computing for cloud consumers. The cloud provider manages the infrastructure, the platform as well as one or more key applications for the cloud consumer. Where the applications in PaaS enabled cloud computing from a technical point of view the applications in SaaS are responsible for business functionality as consumer relationship management and supply chain management.

In SaaS the infrastructure, platform and application is run by the cloud provider. The cloud consumer has little control and flexibility and all maintenance is handled by the cloud provider[41]. An example of a SaaS cloud provider is Salesforce⁵.

3.1.2 Scalability

One of the great promises of utility computing is scalability. The term scalability describes the ability to handle growth/decline or the ability to increase/decrease size to handle growth/decline[3]. There exist two types of scaling; vertical scaling (scale up) and horizontal scaling (scale out). Vertical scaling will typically consist of adding more resources to an existing node or instance. A general example is adding more RAM and CPU power to a server. It is clear that such a scaling method have serious limitations in the ability to scale, as there are limitations in how much CPU and RAM you can add to a single computer.

³<http://www.windowsazure.com>.

⁴<http://developers.google.com/appengine>.

⁵<http://www.salesforce.com>.

Horizontal scaling adds more nodes to an existing system. This could be adding another server to a server park.

It is a common misconception that cloud computing will solve all scalability issues. The number of instances you can spin up and down and the speed at which this can be done, are irrelevant if the application itself is not scalable. True scalability comes from a scalable platform combined with a scalable application which can be distributed across multiple running instances. The different cloud computing categories provide different levels of aid in creating a scalable application. Using IaaS the cloud consumer must facilitate scalability himself across multiple virtual machines. In PaaS the software environment provides support for scalability but the application must still consider it. In SaaS the scalability must be handled by the cloud provider.

3.1.2.1 Brewers CAP theorem

Brewers CAP was introduced in *Towards Robust Distributed Systems* [9] based on previous work in “Harvest, yield, and scalable tolerant systems” [8] and “Cluster-based scalable network services” [19]. The theorem is essential for scaling horizontally. Brewers CAP theorem states that it is impossible for a distributed system to simultaneously guarantee Consistency (all nodes see the same data at the same time), Availability (a guarantee that every received request is processed) and Partition-Tolerance (the system continues to operate despite arbitrary message loss or failure of part of the system)[10]. The CAP properties are modelled in figure 3.1.

According to this theorem a system can at most guarantee to be either Consistent and Available (CA), Consistent and Partition-Tolerant (CP) or Available and Partition-Tolerant (AP). Brewers CAP theorem was formally proved by Gilbert and Lynch in “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”[23].

More than a decade later the perspective on the CAP has changed. While the CAP theorem does state that only two out of the three criteria can be perfectly guaranteed, they are not all or nothing and can be implemented to various degrees. The choice of C, A, or P, is not static but can in fact be changed dynamically. These shifts in perception have changed the focus of guaranteeing C, A, or P, perfectly to which degrees Consistency and Availability are guaranteed under a network partition[7]. It is important for the application designer to decide to what degree he wants to guarantee two of the properties and as a cloud consumer he must also consider if and how the cloud provider supports Consistency, Availability and Partition-Tolerance.

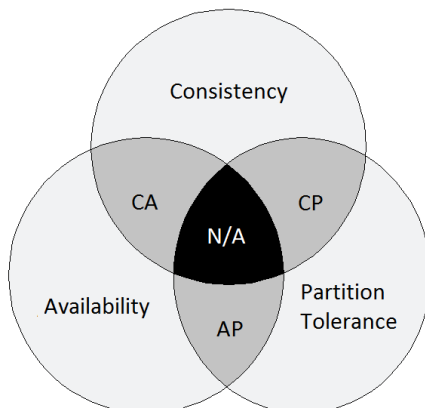


Figure 3.1: Venn diagram showing the three properties Consistency, Availability and Partition tolerance. Any system can guarantee at most two of these three properties at any given time.

3.1.3 Security Risks in Cloud Computing

One of the main concerns regarding Cloud Computing is security. These security concerns can be the deciding factor when a future cloud consumers and users determines whether to use cloud computing. Therefore it is important to identify what security risks cloud computing present and investigate how the individual cloud provider mitigates these risks by addressing them and/or by giving the consumer tools to address them. In this subsection risks unique to cloud computing will be identified based on *BizCloud Overview of Top 10 Security Threats of Cloud Computing*[27] and “Assessing the Security Risks of Cloud Computing”[25]. It is based on these security risks that risk mitigation by the chosen cloud platform provider will be discussed in [3.2.4 Risk Mitigation in Windows Azure](#) and this risk mitigation will be discussed again in [section 7.3](#).

3.1.3.1 Unauthorized User Access and Malicious Insider

When migrating to cloud computing the consumer loses control over the internal security control. This is handled by the cloud provider and the consumer’s security controls are outside the cloud security mechanism. The threat of a malicious insider existed before the migration to the cloud, but it is worth mentioning that now the malicious insider can come from the cloud provider. Furthermore the consumer no longer has physical access to servers. These are quite substantial risks for companies where such an infiltration can cause damage to finances,

production and reputation[27].

3.1.3.2 Geographical Data Location

In cloud computing you are totally separated from the hardware. This means that you may not know where your data is located and maybe not even know which country. This can be an issue because of national privacy regulations. If your data is of highly sensitive manner it may also be an issue storing it in countries like the United States where they have the Patriot Act. If the cloud provider offers redundancy and backup you must likewise consider the location of these services[25].

3.1.3.3 Data Segregation

For cloud environments and especially for PaaS and SaaS, cloud consumer data is stored in a shared environment. 3.4.8 Web Service Security will discuss how data can be protected while in transit, but it is equally important that it is protected when at rest. This new risk is due to the fact that the cloud consumers now share infrastructure, in contrast the regular hosting in data centers do not have this issue as little or no infrastructure is actually shared. As a service provider it is equally important that you can ensure that your users data is segregated. This can be done by programatically compartmentalize the resources or encrypt resources on a per user basis[25].

3.1.3.4 Uptime

The cloud is not flawless and even though cloud computing scales horizontally and Availability and Partition-Tolerance are valued higher than consistency, it still happens that the cloud is down. Redundancy offered by the cloud provider in the form of running an application in multiple data centres might not be enough. A recent example of this is the breakdown of Microsoft Azure due to a leap year bug[29].⁶ Using cloud computing does not guarantee 100% uptime[27].

3.1.4 Choice of Cloud Computing Category and Cloud Provider

In the choice of cloud computing category the most important factor is abstraction. Choosing a platform that provides the highest level of maintenance

⁶Apparently Microsoft developers have not learned their lesson from the Zune leap year bug <http://www.nytimes.com/2009/01/01/technology/personaltech/01zune.html>

abstraction allows focusing on other issues. It is however clear that there is no SaaS platform provider which offers the desired functionality for this project. Therefore PaaS has been chosen as this provides maintenance abstraction while allowing implementation of custom functionality.

In this project Windows Azure will be used to host the RelayService. There are four main arguments for choosing Windows Azure:

- B&K is already using this platform and are therefore familiar with it.
- Good integration with the development environment used (Visual Studio).
- It has a flexible and extensive framework.
- Allows use of same programming language on RelayClient and RelayService.

The framework will be described in the following section.

3.2 Windows Azure

Windows Azure (or simply Azure) is Microsofts cloud computing platform build to host and scale Web applications through Microsoft data centers. Azure is classified both as PaaS and IaaS, because it supports both categories. The platform consists of the following components Execution Models, Data Management, Networking, Business Analytics, Messaging, Caching, Identity, High-Performance Computing, Media, Commerce and the Software Development Kits. In this section the following three core components Execution Models, Data Management and Messaging will be explained. On the background of these core components explained how Windows Azure mitigates the risks identified in [3.1.3 Security Risks in Cloud Computing](#).

3.2.1 Execution Models

The Execution Model term covers the running instances in the cloud computing platform. The term role is used to describe the different categories of computing instances available. In this subsection the different categories will be introduced. The choice of execution model is covered in section [6.1](#) after the initial design decisions have been made. The three types of roles are Virtual Machine Role, WebRole and WorkerRole. All roles are virtual instances running on top of a physical host server. The use of the roles for different applications will be

described later in this section. Each has virtual CPU power, virtual RAM memory and a virtual hard drive. The Azure platform does not regard the data stored on the virtual hard drive or in the memory as persistent, nor does it facilitate consistency of this data between the roles.

If the host server for an instance experiences a hardware failure the instance will be lost. Any unique data stored in the virtual hard drive or memory of the host is lost. Windows Azure will spin up a new instance with the specific role[28]. The 99.95% uptime in the Windows Azure Service Level Agreement[32] is only guaranteed if at least two redundant instances run in different Upgrade Domains⁷. It is clear from this that in regards to execution models Windows Azure sacrifices Consistency for Partition-Tolerance and Availability. In this case Windows Azure takes this sacrifice to the extreme as it makes no attempt to have consistent roles. Even though consistency and persistence is not guaranteed in Azure roles it is present in the Azure platform. Persistent storage is offered with Azure data management and allows Azure roles access and manipulate a common set of data. This will be discussed in [3.2.3 Data Management](#).

The execution models support horizontal scaling by creating multiple instances of the same role⁸. When multiple instances of the same image is running traffic is routed via a load balancer. The load balancer decides which instance to forward to based on resource usages etc., this is illustrated in figure 3.2. There is no way to know which instance will receive a certain request. This further complicates having state information on a role.

3.2.1.1 Virtual Machine Role

The virtual machine role represents a virtual machine. This is similar to the virtual machines running on other IaaS providers such as Amazon⁹. Some of the advantages of running a virtual machine in Azure is the world wide reach of the Azure data centres and the benefits of the Azure framework[28].

⁷Upgrade Domain refers to a set of Windows Azure compute nodes to which platform updates are concurrently applied. Two identical instances will per default be in different Upgrade Domains. This also implies that the instances are running on different physical hardware.

⁸This can be done manually in the azure cloud manager or auto scaling can be enabled based on different load variables

⁹<http://aws.amazon.com/ec2/>.

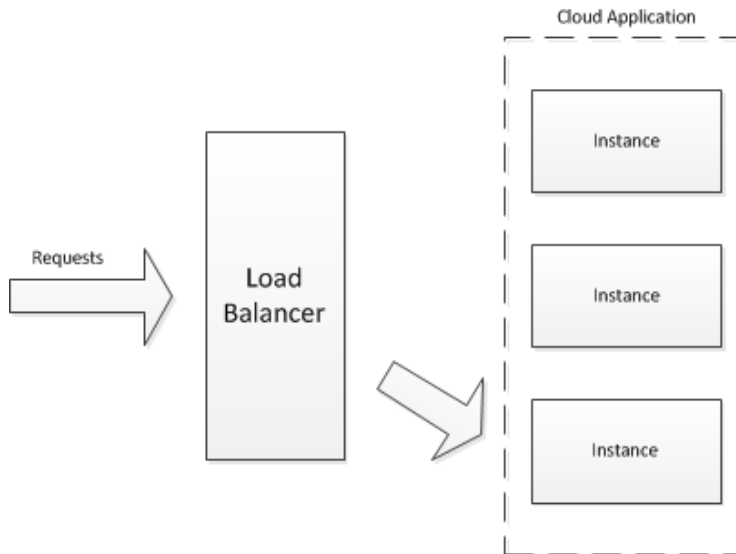


Figure 3.2: Diagram showing the Load Balancer which delegates incoming requests to an instance of the role. It is not possible to know which instance will process a request.

3.2.1.2 WebRole

The WebRole instance is a Web application hosted on a Azure Virtual Machine running IIS¹⁰. The Web Role is best suited as a Web based front end for an application. The WebRole is not suitable for long running processes[28].

3.2.1.3 WorkerRole

David Platt who teaches .Net programming on the Harvard University has made a quite interesting description of the difference between a Worker Role and a Web Role.

When you go to the store to buy a fish, you are welcomed by a nice pretty lady. She takes the order, and handles it back to the person who cut the fish. This pretty lady does not know anything about how you cut a fish. And the person, who is cutting the fish, does not know anything about consumer service. So to clarify it for you. The nice pretty lady is the Web Role, looking good, and is visible

¹⁰Internet Information Services, a Microsoft Web Server

and accessible for the consumer. The person who cuts the fish, is the Worker Role. He/she does not have to be good looking at all, but needs to know what he/she's doing to get the process done.

The WorkerRole is comparable to a Windows service running in the background. It is primarily suited for running background tasks and services[28]. Figure 3.3 describes the lifecycle of a WorkerRole. After the WorkerRole is instantiated

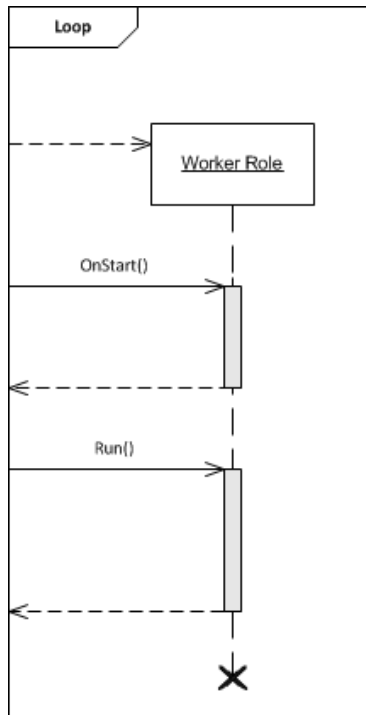


Figure 3.3: Sequence diagram showing the life cycle of a Worker Role

the `OnStart` method is called. After this has finished the `Run` method is called. When the `Run` method is finished the `WorkerRole` is destroyed and a new instance of the `WorkerRole` is instantiated. Because of this lifecycle the `Run` method should contain a loop that does not finish unless it is absolutely necessary.

3.2.1.4 Separation of Background and Front End Processes

One of the major arguments for separating background and foreground services onto different instances is scalability. The separation based on the service they

provide enables better server resource administration. Keeping the two apart also ensures that neither are using the others resources. Imagine if Google was slow because they were busy indexing behind the scene. The two processes can also scale independently based on their needs and it makes a more flexible design as the front end is no longer tied to a single back end instance. It enables the front end to make a request to any specific back end.

To accommodate the separation of back end and front end services Microsoft has introduced the Messaging framework which will be discussed in the following subsection.

3.2.2 Messaging

In the distributed cloud environment communication between instances are done via messaging. The Azure framework present three ways of exchanging messages between roles: Direct Connection between message producer and consumer which requires a proprietary message system, the Azure Cloud Queue and the Azure Service Bus. The Azure Cloud Queue represents a queue model while the Service Bus both supports a queue and a topic/subscription model. [3.2.2.3 Message Systems Pros and Cons](#) summarizes the pros and cons of the different message systems. In section [6.1](#) an execution model will be chosen that requires messaging. In [6.1.1 Choosing a Messaging system](#) the choice of messaging system will be made based on the descriptions in this subsection.

3.2.2.1 Cloud Queue and the Service Bus Queue

The Cloud Queue and the Service Bus Queue, which is a part of the Service Bus, both implement a queue and are therefore somewhat similar. They both follow the basic concept of a message being added to the tail of the queue and dequeued in the head of the queue. There are however implementation differences. The Cloud Queue is part of the Azure Storage and have strength in data storage while the Service Bus Queue is facilitated by the Service Bus which supplies more advanced functionality. Mizonov and Manheim[[33](#)] describes the differences between the two queue implementations. The table [3.1](#) highlights some of the important differences between the implementations.

3.2.2.2 Service Bus Topics/Subscriptions

The Service Bus Topics/Subscription is an implementation of the publish-subscribe pattern. Publish-subscribe is a message pattern where message senders, called publishers, do not know the intended receiver(s) of a message. The messages

Comparison Criteria	Windows Azure Queues	Service Bus Queues
Ordering Guarantee	No	Yes - First-In-First-Out (FIFO)
Delivery guarantee	At-Least-Once	At-Least-Once and At-Most-Once
Message groups	No	Yes (through the use of messaging sessions)
Maximum message size	64 KB	256 KB
Maximum queue size	100 TB	1, 2, 3, 4 or 5 GB
Maximum message Time To Live	7 days	Unlimited
Maximum number of queues	Unlimited	10,000
Average latency	10 ms	100 ms
Maximum throughput	Up to 2,000 messages per second	Up to 2,000 messages per second

Table 3.1: Queue Capabilities

are sent to an intermediary node at which receivers (also called subscribers) have subscribed to messages of interest. Messages of interest are forwarded to the individual subscribers. The sender and receiver have no direct relationship and this gives a loose coupling. In the Service Bus subscribers subscribe to a topic and optionally adding a filter. The filter can be based on several standard properties of the messages or the message content if it is XML serialized. Like the Service Bus Queue the Service Bus Topics support At-Least-Once, At-Most-Once and deadlettering. Likewise the topics have similar restrictions in size and capacity.

3.2.2.3 Message Systems Pros and Cons

Below the pros and cons of the different message systems are summarized.

Direct Connection

- + Independent of cloud
- + Unlimited message size
- + Low latency
- N Worker Roles and M Web Roles require N*M connections
- Does not specify a specific message standard

Azure Queue

- + Good for large volume of messages
- + Only 10 ms of latency
- Message size only up to 64 KB
- If messages are intended for a specific instance, multiple queues are needed to prevent receivers from stealing each others messages because it is impossible to know who the intended receiver is before reading the message.
- No ordering guarantee

Service Bus Queue

- + FIFO ordering
- + Message size up to 254 KB
- If messages are intended for a specific instance, multiple queues are needed to prevent receivers from stealing each others messages because it is impossible to know who the intended receiver is before reading the message.
- 100 ms of latency

Service Bus Topic/Subscription

- + FIFO ordering
- + Message size up to 254KB
- + Publish/subscription pattern allows multiple receivers and senders
- + Allows filtering via SQL messages
- + Allows fast filtering on CorrelationId
- 100 ms of latency

3.2.3 Data Management

Data stored on a virtual hard drive on a role is neither persistent nor durable. Furthermore the idea of storing data on individual clients complicates horizontal scaling. To accommodate this Windows Azure introduces a persistent storage outside a role which is available to all running instances of an application. In this central storage three types of storage exist; Blob, Azure Table and Azure SQL.

3.2.3.1 Blob

Blob is an acronym for Binary Large Object. Blobs are used for storing individual data items. A single blob will typically contain a document, picture, audio file or a video. A Blob is stored in a Blob container tied to a storage account. The Blob container is similar to a directory in normal computing terms. There are several features available for managing Blobs. Blobs can be accessed via a RESTful Web service interface, it can be publicly exposed as read only, snapshots of a Blob can be made for revision control and the Blob can be locked so that it only accessible by a single application[39].

3.2.3.2 Azure Table

Azure Table Storage is a key value storage for rows such as orders news feeds and other data that does not require server side computations such joins, sorts, views, and stored procedures. The data is stored in rows which, unlike a relational database table, can contain a varying set of properties[39]. In fact the Azure Table Storage is a NoSQL database[12]. Similar to Blob storage, access can be obtained via a RESTful Web service interface.

3.2.3.3 Azure SQL Database

Azure SQL Database is a relational database available in the Azure Storage solution. Azure SQL supports a subset of the Transact-SQL¹¹ for SQL Server 2008 and thus have an tabular data stream (TDS)¹² interface similar to the one of SQL Server 2008. Behind this interface there are notably differences. SQL Azure servers and databases are virtual objects and do not correspond

¹¹: Transact-SQL is a language that contains commands used to administer instances of SQL Server including creating and managing all objects in an instance of SQL Server, and inserting, retrieving, modifying, and deleting all data in tables. Applications can communicate with an instance of SQL Server by sending Transact-SQL statements to the server.

¹²TDS is SQL Server native communication language.

to physical servers or databases and a database is not a single database but rather a cluster consisting of three database nodes in the same data center[2]. Requests made from an application are initially handled by the SQL Azure Gateway Layer. The Gateway Layer has two functions. First it acts as a proxy, finding the database nodes tied to the storage account. Secondly it functions as a statefull firewall which understands TDS. As a statefull firewall it checks the TDS packages and only forward the packages to the database nodes if they meet requirements such as encryption, username, password, order, etc. Valid requests are sent to the primary database node which forwards it to the secondary nodes as shown in figure 3.4. A response is only sent when at least one of the secondary nodes have also processed the request[2]. SQL supports scaling out by introducing federations. In practice this means that tables are divided onto multiple instances based on the primary key of a table. E.g all tuples with primary keys lower than 1000000 are stored on federation member one, all tuples with primary keys larger than 1000000 and lower than 2000000 are stored on federation member two etc.

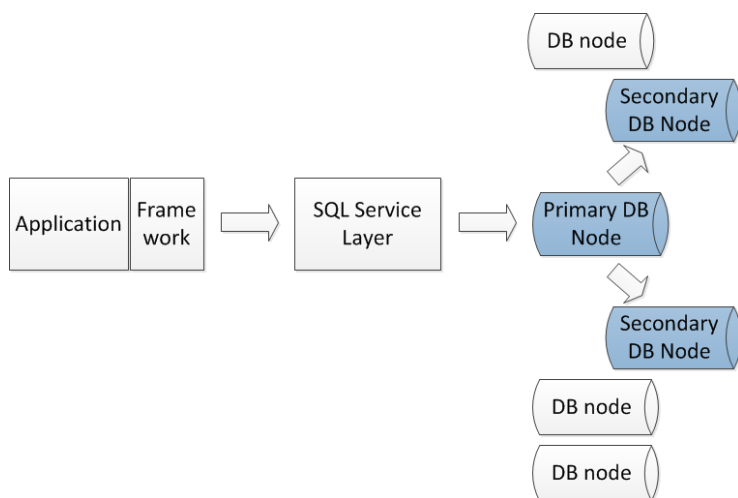


Figure 3.4: Flow diagram showing how a SQL transaction is processed in the system.

3.2.3.4 Azure Table and SQL Azure Compared

The Azure Storage is often depicted as a single uniform data storage. At a lower level of abstraction this is not the case as the different types of storage have different characteristics. The Blob and Table both assure BASE (Basically Available, Soft state, Eventually consistent). “Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency” [13] sug-

Comparison Criteria	Azure Table	SQL Azure
Geographical replication of data	Yes	No
Table Schema	Relaxed (Each row in a table can have different properties)	Managed (Fixed schema for each table)
Relations between data	No	Yes (using foreign keys)
Server side processing	No (Only supports CRUD)	Yes (Supports CRUD + joins, views, etc.)
Maximum data size	100TB per table	150GB per database
Charge	Storage size + transactions	Storage size

Table 3.2: Azure storage comparison

gests that Azure Table contains elements of both CP and AP, however from the cloud customers perspective it is the AP which is offered. The SQL Azure database guarantee the more traditional ACID (Atomicity, Consistency, Isolation, Durability) which is equivalent of CP in Brewers theorem. Azure Tables and Azure SQL database are overlapping meaning that a solution could use either for data storage. The choice between the two are among other things the choice between Availability and Consistency[34]. In reality both choices provide strong consistency and an impressive availability, therefore other characteristics are important in the choice of storage solution. Table 3.2 shows some of the important differences between the two storage types.

The two storage types address two different needs. One is massively scalable but with limited data management while the other has more limited scalability but excels in data management. In fact the two are complimentary and many solutions use both Azure Table and SQL Azure. It is understand that the two are not mutually exclusive and that part of a data storage could be kept in Azure Table and part of the data storage could be kept in SQL Azure. In 6.3.1 [Choosing Storage Type](#) the choice of storage type will be made based on needs highlighted by the design as well as the differences described here.

3.2.4 Risk Mitigation in Windows Azure

It is important to understand how the cloud provider mitigates risks. Not only to convince cloud consumers that moving their solution to the cloud is a sound choice, but also to guarantee to the cloud users that their accounts and data are handled in a secure and satisfactory manner. This subsection describes how

the Azure platform mitigates risks identified in [3.1.3 Security Risks in Cloud Computing](#).

3.2.4.1 Geographical Data Location

Windows Azure allows the cloud consumer to choose the geographical location of his data storage as well as his running instances. The consumer has the choice of eight geographical locations; four in north America, one in Western Europe, one in Northern Europe and two in Asia. This mitigates the Geographical Data Location risk introduced in [3.1.3.2 Geographical Data Location](#). However this can still be unacceptable for some consumers. Sometimes the consumer may be required to store the data within a specific country (e.g. China) and other times storing data in the cloud will be unacceptable due to legislation or official demands. The only way to solve such demands would be to host the services in-house, this will be discussed in [7.3 A RelayService Without Cloud](#).

3.2.4.2 Data Segregation

In [3.1.3.3 Data Segregation](#) the data segregation risk was identified. The Microsoft Storage does not apply encryption on a per consumer basis. However they do allow the individual consumers to encrypt data themselves. This is the primary mechanism provided by the Azure platform to mitigate the Data Segregation risk.

3.2.4.3 Uptime

The Uptime risk described in [3.1.3.4 Uptime](#) is mitigated by having both local and off site redundancy available. Microsoft offers a 99.95% monthly uptime Service Level Agreement (SLA) if the consumer have geographically separated redundancy. The different components of the Azure platform have been investigated on the background of the Brewers CAP theorem. The conclusion of this is that it is theoretically possible to ensure Availability and Partition-Tolerance, but it is difficult in practice. The main issue is that while the instances are separated they still rely on the same platform and are still vulnerable to total Azure breakdowns. The Azure platform has become more mature and stable but consumers and users exist whom will require a better uptime than promised in the SLA. The primary way to achieve such an uptime is still to have redundant instances hosted a place that is tolerant to Azure failures.

3.3 Web Services

The previous sections discussed important concepts in regards to hosting the RelayService. This section will discuss how the StatusInterface can be realized on the RelayService. The interface offer a service via Web communication and this section will give an overview of the two Web service categories. Based on this overview a category is chosen for realizing the StatusInterface and the Web service technology will be described in depth. Security risks related to Web services will be identified and it will be discussed how the chosen Web service category can mitigate these risks.

The W3C¹³ definition of a Web service is from 2004 and is tightly coupled with specific standards for SOAP based Web services[24]. This definition must be relaxed to include the modern RESTful Web services. Below the relaxed definition is given.

DEFINITION 3.1 A Web service is a software system designed to support interoperable machine-to-machine interaction over a network typically conveyed using HTTP.

Web services fall into two categories.

1. SOAP based Web services
2. RESTful Web services.

In this section the two Web service types and the Windows Communication Foundation framework will be described.

3.3.1 SOAP

SOAP was developed for Microsoft in 1998, and is currently being maintained by the XML Protocol Working Group of the World Wide Web Consortium. In 2003 SOAP version 1.2 became a W3C recommendation. Later it became the underlying layer of WSDL and UDDI.

Web Services Description Language (WSDL) is also XML-based, and describes functionality offered by a Web Service[35, p. 148]. A WSDL-file is machine-readable and describes how to call the Web Service, which parameters it takes and what the service returns.

¹³The World Wide Web Consortium <http://www.w3.org/>

When talking about SOAP and Web Services, there are many other specifications than just SOAP and WSDL. Collectively, all these specifications are referred to as WS-*. The motivation of all these different specification, comes from many different requirements from different groups. For example the WS-Security that apply security for Web services. Then there are others like WS-MetadataExchange that allow retrieval of meta data from a Web Service endpoint. SOAP is still widely used in business to business integration, partly due to the well developed standards which fits most needs.

3.3.1.1 SOAP based Web Services Pros and Cons

- + Required machine readable definition in the WSDL.
- + Can use different transport layer protocols.
- + Supports asynchronous and synchronous messaging.
- + WS-* includes messaging, transaction, compensation and security standards.
- Overhead introduced by the use of XML.
- Complex standards require use of framework.

3.3.2 REST

REpresentational State Transfer (REST) is a resource oriented architecture for communicating, for example over the Internet. It was introduced in Ph.d Roy Fieldings dissertation from 2000[14]. REST is a set of constraints that together form an architectural pattern. For a service to become RESTful it has to conform to the constraints; Statelessness, Addressability, Uniform Interface, Connectedness, Layered System and Cache. These constraints will be discussed in depth in section 3.4. REST was derived by analysing the existing World Wide Web architecture. Therefore the combination of REST and the HTTP protocol appears seamless. However the two are mutually independent and REST as an architectural pattern is not bound to any specific technology or protocol. A RESTful Web service however is the combination of REST architecture applied to Web services using Web standards like HTTP[42]. In other words RESTful Web services is a single application of REST. Because of the seamless integration with HTTP and the relatively simple architectural constraints RESTful Web services are regarded as simple and lightweight when compared to SOAP based Web services.

Since its introduction the REST architectural pattern has seen little change and it remains as described in Roy Fieldings dissertation. The concept of RESTful Web services has also remained unchanged as it is REST architecture integrated with Web standards. Until recently the only development from a software perspective has been the emergence of frameworks supporting REST. Where SOAP based Web services used to be the preferred choice, companies are now turning towards REST. Large Web based companies like Amazon¹⁴, Facebook¹⁵ and Google¹⁶ have mature REST APIs and the trend goes towards integrating back end business to business via RESTful Web services. To cater for this the JBoss community¹⁷ promoted the development of a new series of standards, the REST-*

3.3.2.1 REST-*

REST-* was publically announced at JBoss World 2009. The Website dedicated to the project gives the following description[38].

REST-* is an open source project dedicated to bringing the architecture of the Web to common patterns in middleware technology. REST has a the potential to re-define how application developers interact with traditional middleware services. The REST-* community aims to re-examine which of these traditional services fits within the REST model by defining new standards, guidelines, and specifications. Where appropriate, any end product will be published at IETF.

The four areas REST-* addresses are Workflow/Business Process Management, Messaging, Compensations and Transactions. These are key components in designing more complex business to business interactions. The concept of transactions is actually a REST anti pattern as distributed transaction models require server side session context, and the proposed standards have little to do with the original REST architecture. Roy Fielding wrote this response to the REST-* standards

Bill, if you want people to have an open mind about what you are trying to do, then the respectful thing would be to remove REST from the name of your site.

¹⁴<http://docs.amazonwebservices.com/AmazonS3/latest/dev/Welcome.html>.

¹⁵<http://developers.facebook.com/docs/reference/api/>.

¹⁶https://developers.google.com/custom-search/docs/dev_guide.

¹⁷<http://www.jboss.org/>

Quite frankly, this is the single dumbest attempt at one-sided "standardization" of anti-REST architecture that I have ever seen. It even manages to one-up the previous all-time-idiocy of IBM when they renamed their CORBA toolkit "Web Services" in a deliberate attempt to confuse customers into thinking they had something to do with the Web.

Distributed transactions are an architectural component of non-REST interaction. Message queues are a common integration technique for non-REST architectures. To claim that either one is a component of "Pragmatic REST" is the equivalent of putting a giant Red Dunce Hat on your head and then parading around as if it were the latest fashion statement.

The idea that the community would welcome such a pack of marketing morons as the standards-bearers of REST is simply ridiculous. Just close the stupid site down.

Sincerely,

Roy T. Fielding [15]

Fielding's wish came through. Although the REST-* site is still running the project is almost dead. The last update to the proposed standards were in March 2010 and no further submission have been made since then. The implementation of the REST-* standard also limits itself to the HornetQ (a JBoss product) which implements the REST-* Messaging standard. Although the REST-* standards has little relation with the original REST principles, it did attempt to address important shortcomings as non-repudiation in RESTful Web services. These shortcomings have yet to be solved and limits the REST application in some business areas where transaction, compensation and messaging are essential.

3.3.2.2 RESTful Web services Pros and Cons

- + Architectural style, not a standard.
- + Simple due to good integration with the existing HTTP protocol and other Web standards.
- + Not bound to XML so overhead may be reduced.
- + Good interfaces for exposing data.
- + Good scalability. Inherited scalability from Web standards.
- No standard for implementing transactions.

- No required standard for describing a RESTful services.
- Due to statelessness constraint, state must be kept at client side, which requires fat clients.
- HTTP which RESTful Web services are based on do not per default support asynchronous communication.

3.3.3 Windows Communication Foundation

Windows Communication Foundation (WCF) is a set of API's in the .Net framework used for designing, building applications under SOA (Service-Oriented Architecture). WCF supports both advanced RESTful and SOAP based Web services and includes implementation of several WS-* protocols such as WS-security.

Communication is done via synchronous or asynchronous messages containing data that are send between endpoints. Endpoints contain an URI address and a binding property, where data can be strings, bytestreams and XML. An example could be a client of a service requesting data from that service. Fx REST will do it by sending out a GET request, and data would be returned. Although it is platform independent on the client-side, the server side will be bound to Microsoft[36].

3.3.3.1 Windows Communication Foundation Pros and Cons

- + Client-side is platform independent
- + Flexible (Supports both SOAP and REST)
- + Switch protocols without programming
- + Supports both synchronous and asynchronous communication
- Service relies on the WCF framework and requires that the server supports hosting these services. The .Net compact framework does only provides a subset of WCF. This limited WCF framework cannot host services.

3.3.4 Choice of Web Service technology

In this section the Web service technology for exposing the RemoteAPI, the Browser Interface, and the StatusInterface on the RelayService will be chosen. Thereby taking the first step in fulfilling MH2 and MH3.

The optimal choice for Web services technology would be WCF as the combination with the Azure Service Bus enables publishing the Web service, so it is accessible when a public IP address is not present. This solution is described in [2.1.4 Azure Service Bus and Windows Communication Foundation](#) and the problem with this solution is that only a subset of the WCF framework is supported in the .Net Compact framework¹⁸ and therefore it cannot host services. This results in a loss of design consistency between the 2250 and the RelayService. As a result WCF is disregarded.

Therefore the choice stands between a SOAP based Web service and a RESTful Web service. The SOAP based Web services strengths lie in the WSDL description, the many standards and the good frameworks for stub generation. In contrast the RESTful Web services excel with the its simplicity, scalability and seamless integration with the HTTP standard. The existing RemoteAPI interface is a RESTful Web services and the functional requirement MH2 specifies that we must ensure backward compatibility. Therefore the RemoteAPI interface must be present as a RESTful interface and the logical choice is to also expose the StatusInterface using the same technology.

3.4 REST in depth

This section describes the REST constraints defined in Roy Fieldings dissertation[14] and “Principled design of the modern Web architecture” [16]. Anti-patterns are patterns that may be commonly used but are ineffective and/or counter productive in practice¹⁹. Examples of REST anti patterns are included in this section to illustrate the effects and document their existence. It is worth noting that if a constraint is broken the application is no longer considered as RESTful.

3.4.1 Client Server

Client and server are separated by a uniform interface. The interface gives a loose coupling between the client and the server. The client deals with user interface and session context while the server handles data storage and other back end functions.

¹⁸This is due to size restrictions of the ROM on an embedded device

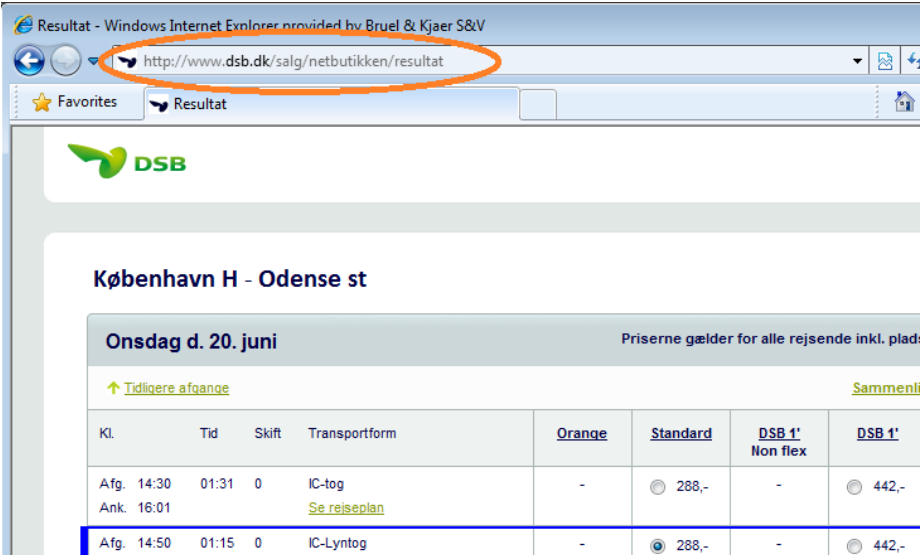
¹⁹Anti-pattern Wikipedia <http://en.wikipedia.org/wiki/Anti-pattern>

3.4.2 Addressability

All resources must be uniquely addressable. The address of the resource must reasonably describe the concept of a resource and can in some cases also describe its relation to other resources. In the World Wide Web URI is used for addressing and this standard is therefore also used by RESTful Web services. This constraint allows us to directly address a resource and use it as an entry point. Furthermore it supports the HTTP caching as it exposes the resources individually. The URI standard has a hierarchical structure and a RESTful Web services should use this to make meaningful addresses and give information about the resource structure.

3.4.2.1 Addressability Anti-pattern

Web sites that do not uniquely identify relevant resources through URI are violating the addressability constraint of REST. An example of such a violation is from the Web application of the Danish State Railways (DSB). One of the features of the Web application is to search for journeys and buy tickets. After performing a search we reach the page illustrated in figure 3.5. The URI for



The screenshot shows a Windows Internet Explorer browser window. The address bar contains the URL `http://www.dsb.dk/salg/netbutikken/resultat`, which is circled in orange. The page content is from DSB (Danish State Railways) and shows search results for the route 'København H - Odense st' on 'Onsdag d. 20. juni'. The results are presented in a table with columns for 'Kl.', 'Tid', 'Skift', 'Transportform', and four price categories: 'Orange', 'Standard', 'DSB 1' Non flex', and 'DSB 1''. The first row shows 'Afg. 14:30' and 'Ank. 16:01' for 'IC-tog' with a price of 288,- for Standard and 442,- for DSB 1'. The second row shows 'Afg. 14:50' for 'IC-Lyntog' with a price of 288,- for Standard and 442,- for DSB 1'. A link 'Se rejseplan' is visible between the two rows.

Kl.	Tid	Skift	Transportform	Orange	Standard	DSB 1' Non flex	DSB 1'
Afg. 14:30	01:31	0	IC-tog	-	288,-	-	442,-
Ank. 16:01			Se rejseplan				
Afg. 14:50	01:15	0	IC-Lyntog	-	288,-	-	442,-

Figure 3.5: Search result page violating the addressability constraints

the result page does not appear to reflect the search and performing additional searches with varying parameters reveal that the URI remain the same. The

conclusion is that the search result as a resource is not addressable and a search result cannot be accessed directly via a link.

3.4.3 Statelessness

Communication between client and server must be stateless. Each request from the client must be self contained so that all information needed to understand the request is contained in the request itself. Server side communication context must not be used to process a request and therefore all session state is kept on the client side. In other words no session or communication context may be stored on the server. However resources exposed and other data on the server can be stateful without violating this constraint.

Fielding argues that

Scalability is improved because not having to store state between requests allows the server component to quickly free resources, and further simplifies implementation because the server doesn't have to manage resource usage across requests. [14, p.79]

In regards to cloud computing and horizontal scaling it gives a further advantage as it is irrelevant what server instance a request is delegated to. This is important in the Azure platform as the request is delegated by the load balancer as described in [3.2.1 Execution Models](#).

3.4.3.1 Statelessness Anti-pattern

A large number of Web sites use cookies and sessions to maintain a session state. This requires additional server side resources for maintaining the session state and is a violation of the statelessness constraint in REST. Resources are freed by performing garbage collection on the stored session after a short idle time. The Web application from DSB discussed in [3.4.2.1](#) contains an example of statelessness anti-pattern. If a user performs a search and tries re-access the results after approximately 20 minutes he or she will be met with the Web page illustrated in figure [3.6](#). It can be difficult to say whether the statelessness anti-pattern is present because the addressability anti-pattern is present or if the addressability anti-pattern is unimportant because of the stateless anti-pattern. It is however clear that their presence result in bad usability for the user.

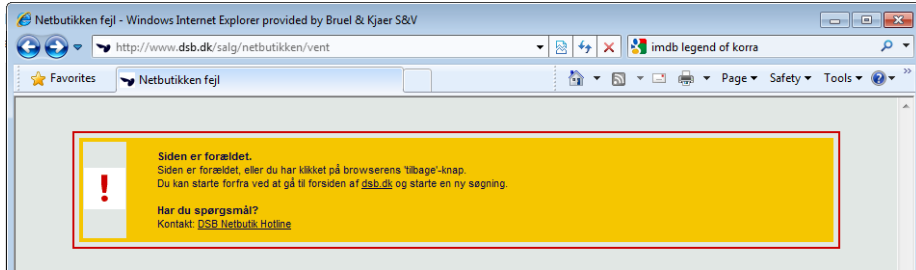


Figure 3.6: Search result page violating the statelessness constraint

3.4.4 Uniform Interface

The REST architecture style applies the software engineering concept of generality and emphasizes uniform interfaces between components. The standardized way of interacting with resources gives a low barrier and decouples the interface from the actual service it provides. The uniform interface requires that information is transferred in a standardized way and it can therefore not be designed for a specific application's needs. The following constraints govern the development of the uniform interface.

Identification of resources

Individual resources are identified in requests. The representation presented to the requester is conceptually separate from the actual resource. For example a resource will often be stored in a database while the representation can be in the XML or JSON²⁰ format.

Manipulation of resources through representations

The representation of a resource, including the meta data attached, includes the information needed to modify or delete the resource.

Self descriptive messages

Messages must include descriptive information on how to process the message itself. For example a HTTP response will include a content type field. Responses must also explicitly or implicitly indicate their cacheability

Hypermedia as the engine of application state

Resources act as entry points to the application. Further transitions are dynamically identified within the hypermedia by the server. Fx if a work flow consist of interaction with a series of resources the client only knows the entry point resources and is thereafter guided by the server which

²⁰JavaScript Object Notation Wikipedia <http://en.wikipedia.org/wiki/JSON>.

will respond with a link to the next resource. This is also evident in the connectedness requirement.

RESTful Web services follow the Create Read Update Delete (CRUD) interface, mapping to the HTTP methods POST, GET, PUT and DELETE. The concepts of safety²¹ and idempotence²² are used to classify and distinguish between methods. A RESTful Web service should be designed so that GET is safe. Changes should not be requested via the GET method. PUT and DELETE methods must be idempotent and POST is neither safe nor idempotent[17, Section 9.1.2].

3.4.4.1 Uniform Interface Anti-pattern

DSB provides us with an example of the uniform interface anti-pattern in their journey search interface. Going through the HTML reveals a semantic misuse of the HTTP methods as seen in figure 3.7. POST is used for the search function but the function is safe and therefore GET should be used. Using POST instead of GET forces the browsers to invalidate their cached representations and will typically present usability issues for the user when navigating backwards and forwards.

```

<body class="clearfix" style="margin:0px; background-color:white;" onload="resizeToIFrame()">
  <div id="Nb4Sogebox" class="content soegeboks">
    <div>
      <form id="id33" method="post" action="?wicket:interface=:25:soegeboks:form::IFormSubmitListener::">
        <div style="width:100%; height:0px; position:absolute;left:-100px;top:-100px;overflow:hidden">
          <div class="standardsogebox">
            <div class="contentbox topimage textcolor" style="padding-top: 13px;">
              <div class="contentbox image2">
                <div class="destinationcontentbox image2">
                  <div class="destinationcontentbox image2" style="padding-bottom: 12px;">
                    <div class="contentbox image1 textcolor" style="padding:7px 0;">
                      <div class="contentbox image2" style="padding:12px 0;">
                        <div>
                          <div>
                        </div>
                      </div>
                    </div>
                  </div>
                </div>
              </div>
            </div>
          </div>
        </form>
      </div>
    </div>
  </div>

```

Figure 3.7: HTML source showing the semantic violation

²¹A safe method is a method where changes, if any, are not requested by the invoker. The invoker cannot be held accountable for any side effects as "he" did not request them.

Example of Safe method: Performing a Google search can be considered safe as you are simply requesting data and not requesting any changes. Google may log your search and therefore a change does occur, but as it was not requested the search can still be considered safe.

²²An idempotent method is a method where applying the method one or more times with the same parameters will yield the same result. By definition a safe method must be idempotent. Example of Idempotent method: Updating your contact details on a Website can be considered idempotent as doing so one or more times will yield the same result.

3.4.5 Connectedness

Relationships between resources should be described as transitions in hypermedia. In RESTful Web services relationships that are navigable forward should be described with links.

3.4.6 Layered System

In REST a series of layers can exist between the client and the server. The client cannot see beyond the initial layer and does not communicate directly with the ultimate receiver. The intermediary layers/components can actively transform message content as they are self-descriptive and their semantics visible to these components. The layered system can be used to encapsulate legacy services and clients and can support scalability by enabling load balancing and caching.

3.4.7 Cache

The cache constraint is added to improve network efficiency. Responses must either implicitly or explicitly declare whether they are cacheable. The client can then optionally reuse the cacheable response for identical requests. Likewise the intermediary components, as described in the previous subsection, can choose to reply with a cached response. HTTP/1.1 specifies three headers in a HTTP response for declaring cacheability Cache-Control, ETag and Vary[17].

Cache-Control

The Cache-Control header can be used to specify a time to live which defines how long a response is valid. The header can also declare that the response is not cacheable

ETag

An ETag is a unique Id for this version of a resource at a specific URI. If used the server will include an ETag in every response and the client can choose to include an ETag header in a HTTP conditional get request. If the clients ETag matches the ETag of the response, the server will respond with the 304 (NOT MODIFIED) status code.

Vary

Multiple representation of a resource can exist on the same URI. For example content negotiation can be used to specify whether the response should be in XML or JSON format. The ETag cannot make this distinction. The Vary header field allows a server to define which other header fields will cause the response to vary.

The conditional GET can also be used by setting a If-Modified-Since header. If the response to the given request hasn't changed, the server will respond with the 304 status code.

3.4.8 Web Service Security

To support machine to machine interaction Web services expose standardized interfaces. In SOAP the WSDL is self describing and provides all information needed to communicate with a service, in REST the entry points along with the self descriptive messages likewise provide information needed to interact with a resource. This makes Web services more vulnerable to attacks when made publicly available. The threats for a Web service can be grouped into the four categories Unauthorized access, Unauthorized alteration of message, Man in the Middle and Denial of Service Attacks^[35].

Unauthorized access

A service is available to an unauthorized participant. Information within a message is viewable by a unintended or unauthorized participant.

Unauthorized alteration of message

An attacker alters a message by adding, removing or modifying the content. The receiver mistakenly accepts the content as being from the originator. In a broad context this category also includes replay attacks and session jacking. This threat is often present in combination with the Man in the Middle threat.

Man in the Middle

An attacker compromises an intermediary node in the layered architecture and positions himself between the consumer and provider and poses as respectively provider and consumer. The attacker can control the exchange of messages and potentially read and/or modify the content of the messages²³.

Denial of Service Attacks

The objective of this attack is to make the service unavailable to legitimate users, by using all the systems resources. The system is made unavailable by flooding it with messages from an attacker. Signed or encrypted messages are especially effective as they require additional processing.

The general security mechanisms that minimizes the risk are described below. These descriptions covers the general concepts. The actual implementation

²³An example of a Man in the Middle attack is a routing detour which by modifying headers directs sensitive messages to an outside location.

varies depending on platform, framework and architecture.

3.4.8.1 Authentication

In distributed systems authentication is the mechanism by which clients and service prove that they are acting on the behalf of a specific user or system. Authentication can be one directional where the clients identity is verified by the provider or bi-directional where both client and provider prove their identity. Proof of identity methods can range from supplying username and password to digital certificates²⁴.

3.4.8.2 Authorization

The authorization mechanism provides access control so that only authentic clients obtains access to protected resources and services. The access control is defined by an authorization policy which typically defines permissions on the basis of roles, groups or privileges. Roles groups and privileges are determined based on identity which is established via authentication, thus authorization is dependent on authentication.

3.4.8.3 Integrity

Messages are sent over computer networks and can potentially be modified as mentioned above. The integrity mechanism ensures that the content is original and has not been modified. In other words that the data received is the same as the data sent. Integrity can be achieved by using digital signature to validate the content of a message. The predominant way of doing this is by using hashing algorithms and digitally signed digest codes.

3.4.8.4 Confidentiality

Confidentiality is the ability to ensure that data is only available to those who have the proper authorization. When messages are sent via untrusted sources confidentiality is ensured by applying encryption to the message. This can be done at a network layer where point to point encryption can be enforced or at an application layer where messages or parts of messages can be encrypted.

²⁴See http://en.wikipedia.org/wiki/Public_key_certificate for explanation of this concept

3.4.9 REST Security

REST itself does not deal with security standards and protocols. The seamless combination of REST and Web standards found in RESTful Web services give a very simple way to mitigate several of the risks discussed above. RESTful Web services can use the HyperText Transfer Protocol Secure (HTTPS) for encryption, HTTPS is a combination of HTTP and SSL/TLS. This ensures server authentication and confidentiality. Encryption happens on a transport layer level and therefore security is guaranteed on a transport level and not on an application level. This is one of the main conceptual differences between security in SOAP based Web services and RESTful Web services²⁵. Combining encryption with the Transport Layer Protocol TCP ensures Integrity. The HTTP Access Authentication specification[20] can be used for authentication, but other standards like openId can be used as well. It is evident that the given confidentiality can be dangerous as it exposes authentication data to everyone.

It is important to note that encrypting on a transport layer makes it impossible for intermediary nodes to read the requests and provide cached responses.

3.5 Chapter Summary

In this chapter the benefits of cloud computing was discussed. The different categories were introduced and other relevant concepts were discussed. Windows Azure was chosen as PaaS provider and the key concepts of Windows Azure were discussed. This discussion will later serve as the foundation for the choices regarding design and implementation in Windows Azure. The concept of Web services and the two types SOAP based and RESTful Web services were introduced. It was chosen that the StatusInterface should be RESTful and the REST architecture was discussed in depth to understand the design requirements. This understanding shall later be used to validate that the design complies with the REST constraints. Security risks in cloud computing as well as for Web services were discussed and it was discussed how respectively Windows Azure and RESTful Web services mitigates these risks.

²⁵WS-Security allows encryption on an application layer, which allows endpoint to endpoint encryption effectively preventing all intermediary nodes from reading encrypted content

Components and Communication

The aim of this chapter is to describe the components in the design, their interfaces and specify the data flow between these components. The overall data flow will be described and it will be specified how individual devices can be addressed thereby fulfilling one of the REST constraints. Three different communication strategies for interaction between RelayClient and RelayService will be described and their pros and cons listed. One communication strategy will be selected based on the pros and cons and used as basis for the protocol between RelayService and RelayClient. Requirements for the protocol will be defined and the protocol specified using a protocol state machine diagram. The message types will be identified and the message structure described.

4.1 Component Design

There are four major components in this solution all introduced in section 2.6. The 2250WebServer, the RelayClient (detailed in chapter 5), the RelayService (detailed in chapter 6), and the Customer as illustrated in figure 4.1. The 2250WebServer is an existing component and as mentioned previously it provides the RemoteAPI and the Browser interfaces. The MH2 requirement specifies that the RelayService should provide the same interfaces as the 2250WebServer. Furthermore the functional requirements dictate that the RelayService offer the StatusInterface, which realizes the functionality described in MH3,

MH4, and NH1. These three interfaces will be used by the Customer. The RelayClient must use the RemoteAPI and Browser interfaces on the 2250WebServer as it forwards HTTP requests to the individual interfaces. On a high level of abstraction the RelayClient must offer an interface (ForwardHttp) which forwards HTTP requests to the 2250WebServer and the RelayService must provide an interface used by the RelayClient to authenticate (2250Authenticate) realizing requirement MH6. The actual design may vary from the representation modelled here. The ForwardHttp and 2250Authenticate interfaces and the communication between RelayClient and RelayService will be introduced in section 4.2 and detailed in section 4.3 where the actual design is chosen.

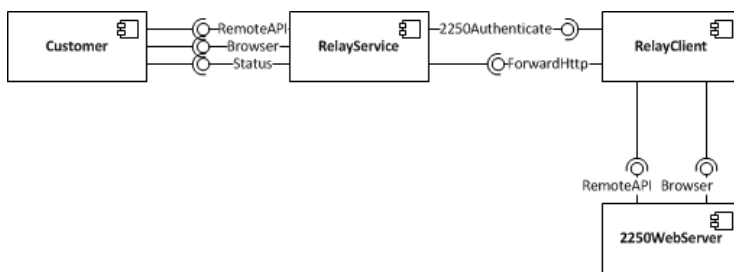


Figure 4.1: Component diagram showing the overall components in the solution and the interfaces they provide and consume.

4.2 Overall Communication

Figure 4.1 shows the interfaces and the uses. This combined with the *Send remote control command* use case provides the basis for describing the communication flow when sending remote control commands.

Following the use case the communication starts with the Customer sending a remote control request to the RelayService. A remote control request is any request for either the RemoteAPI or the Browser interface. These are Web applications and the interfaces communicate via HTTP messages, therefore remote control requests and responses are HTTP messages. From the Customer's point of view the interfaces offered by the RelayService must be identical to the RemoteAPI and the Browser interface offered by the 2250WebServer, as defined in MH2. The implication of this is that the RelayService must appear to be a 2250WebServer from the Customer's viewpoint. This design qualifies RelayService as a reverse proxy¹.

¹A Reverse proxy is a type of proxy server which acts as an intermediary for servers and retrieves resources from these given servers on the behalf of clients

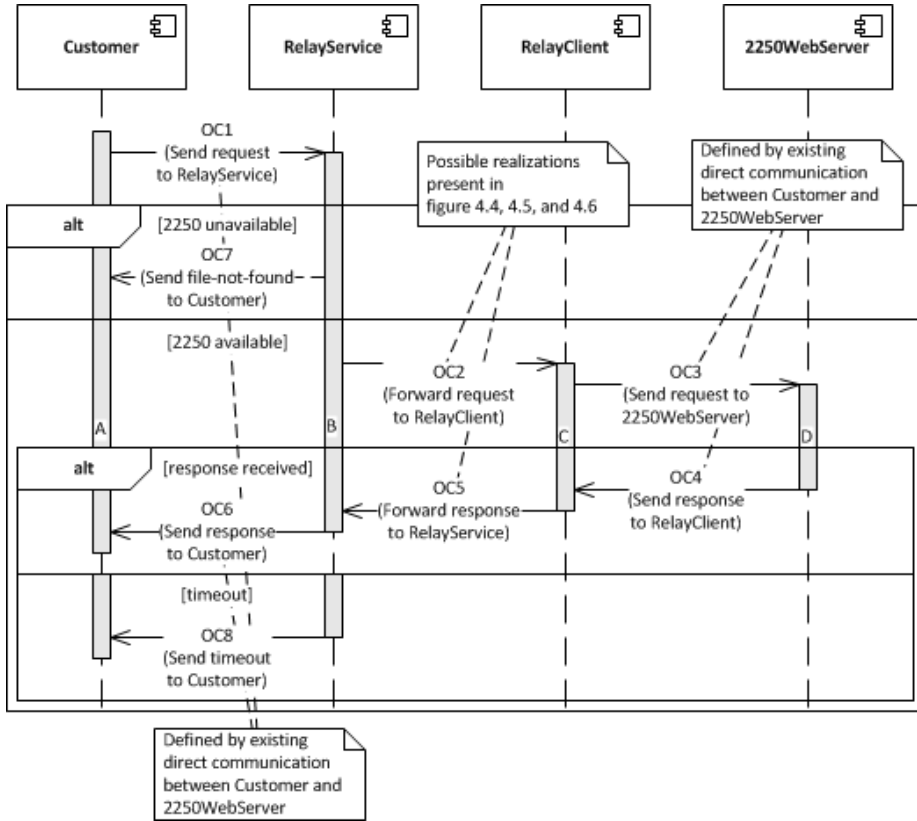


Figure 4.2: Sequence diagram showing the overall control and data flow when a request is relayed.

After the RelayService processes the request it forwards it to the RelayClient. How this is done will be determined in section 4.3. When the RelayClient receives the request it processes it and forwards it to the 2250WebServer. The 2250WebServer returns a HTTP response exactly as it would if it was communicating directly with the Customer. The RelayClient processes this response and forwards it to the RelayService which processes the response and forwards the modified version to the Customer.

4.2.1 Addressable Devices

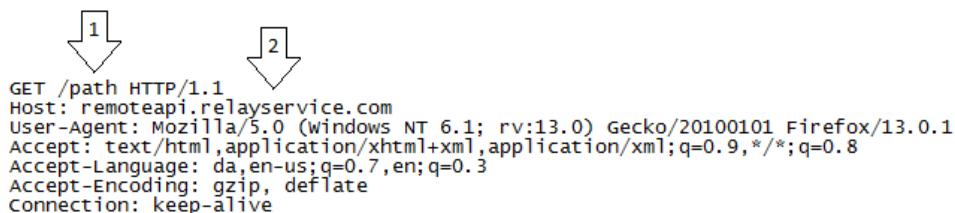
Following requirement NH4, multiple devices may be reachable via the Relay-Service . It acts as a router for the remote control requests and therefore it is necessary to distinguish what device the request is ultimately intended for

so that it can forward the HTTP request to the correct RelayClient as seen in figure 4.2. The MH2 requirement dictates that the HTTP content must not contain additional information and as the HTTP content does not address the device the value identifying the device must be present in the HTTP header. One solution is to make each device uniquely addressable using the URI standard. If the device is regarded as a resource then making it uniquely addressable fulfils the Addressable REST constraint specified in 3.4.2. This design will be used so the RemoteAPI remains RESTful.

The URI standard is transformed and used in a HTTP request by splitting the identifier into a host part and a path part. This is also apparent in figure 4.3 where the address `http://remoteapi.relayservice.com/path` is split into the host "`remoteapi.relayservice.com`" and the path part `/path`.

If the design chosen reference the device in the path for example

`http://remoteapi.RelayService.com/device/<deviceId>` it will cause a path mismatch between a given resource on the reverse proxy and one at the 2250WebServer. This would require that the path is modified by extracting the device identifier and altering the path to the one matching the given resource on the 2250WebServer. This also requires that the HTTP response content is modified so that all relative as well as local absolute links will reference the resources on the reverse proxy. This requires inherent knowledge about the the content of the HTTP messages and how links appear, so that the content can be parsed and links modified. Instead of this it is chosen to maintain the same rela-



The diagram shows an HTTP request with two arrows pointing to specific parts of the request line and headers. Arrow 1 points to the path `/path` in the request line. Arrow 2 points to the host `remoteapi.relayservice.com` in the Host header.

```

GET /path HTTP/1.1
Host: remoteapi.relayservice.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:13.0) Gecko/20100101 Firefox/13.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: da,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
  
```

Figure 4.3: This figure shows a HTTP request sent to `http://remoteapi.RelayService.com/path`. The URI is split into the Path parameter in the HTTP request line(1) and the Host header field(2).

tive path on the reverse proxy and the 2250WebServer. This means that the resource `<device>/index.html` on a device will also be addressed `<RelayService>/index.html`. The implication of this is that the device will be present as a root resource on the RelayService. This will be achieved by having the unique identifier for the device as a sub domain in the address. The result of this design choice will be the given address e.g. `device-123445567.RelayService.com/path`. A further implication of this choice is that all relative links in the HTTP response will have a correct reference. As the RemoteAPI and Browser interface

do not contain any absolute links, it is unnecessary to modify the content of the HTTP responses.

Take a moment to appreciate how simple a solution this is to an otherwise complex problem.

4.3 Communication between RelayClient and RelayService

Previously in the project the communication between the RelayClient and RelayService has been unspecified and only considered on an abstract level. This section will establish the overall communication flow between the two. Three different strategies which are technically possible will be considered, each with its own strengths and weaknesses. When evaluating the strategies two of the criteria are latency, the amount of time a customer has to wait for a command to be processed, and overhead, the amount of extra traffic incurred by this strategy.

In all the proposed strategies it will be the RelayClient who initiates contact to the RelayService. This is because incoming connections cannot always be made to the 2250, which is the core problem in this project.

4.3.1 Polling

One of the simplest solutions to the core problem is a poll strategy. Whenever the customer wants to remote control the device he will send requests to the RelayService. At regular intervals the RelayClient will poll requests from the RelayService and afterwards push responses to the RelayService. This communication flow is further detailed in figure 4.4. One of the strengths with this strategy is the loose coupling between the RelayClient and the RelayService. The RelayService has a well defined interface and it is possible to substitute the RelayService as long as this interface is realized. This strategy fulfils the REST Statelessness constraint and it thereby minimizes the load on RelayService allowing it to handle more RelayClients[4].

To minimize the latency the polling frequency must be high. This in turn increases the overhead. To balance the two the rate of relay messages must be predicted as the ideal situation is when the polling rate and the rate of messages are identical[4]. This relay message rate is impossible to predict and it will only be possible to minimize either latency or overhead. Minimizing one will be at the expense of the other. Another possibility is long-polling. In long-polling the RelayClient requests a relay request from the RelayService. If no new relay

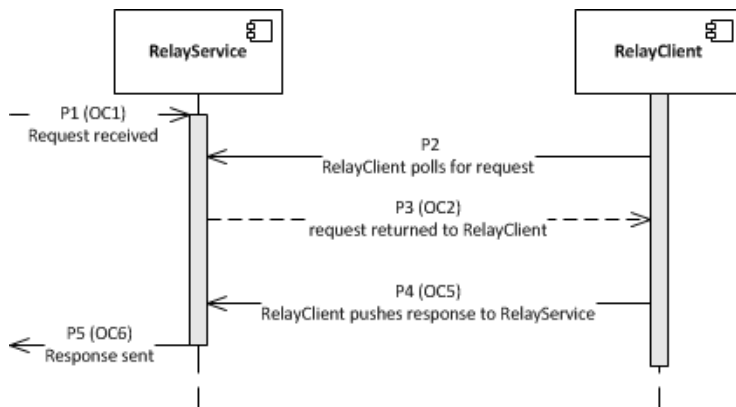


Figure 4.4: Sequence diagram describing the communication flow between Relay and RelayService using the poll strategy.

request is present the connection will be kept alive for a given interval waiting for such a message. Due to the open connection long-polling share more similarities with the Tunnel solution than the Polling solution. Below the pros and cons of the Polling strategy are summarized.

- + Both the ForwardHTTP and the 2250Authentication interfaces are placed on the RelayService. This moves complexity to the RelayService from the RelayClient who will only consume services and not provide any.
- + Simple communication strategy with limited amount of messages exchange.
- + The ForwardHTTP and 2250Authentication interfaces can be RESTful, using standard technology and a clear interface will enable a proprietary RelayService made by customers
- + Fulfils REST Stateless constraint
- The solution will introduce either latency or overhead, most likely a great deal of both.
- High polling rate increases the load on the RelayService drastically.

4.3.2 Tunnel

Another solution is to establish a connection (referred to as tunnel) from the RelayClient to the RelayService and channel data through this tunnel². When the

²The name comes from the similarity to SSH and VPN tunnels. In modern Web development the proposed Tunnel solution is similar to the WebSocket protocol as it provides

RelayClient starts it creates a connection to the RelayService. The connection is created and kept alive whenever the RelayClient can reach the RelayService. If the connection is closed the RelayClient will attempt to re-establish the connection. Whenever the Customer wants to remote control the device he will send requests to the RelayService. The commands are then channelled through the Tunnel to the client application. This communication flow is further detailed in figure 4.5. This established connection between the RelayClient and

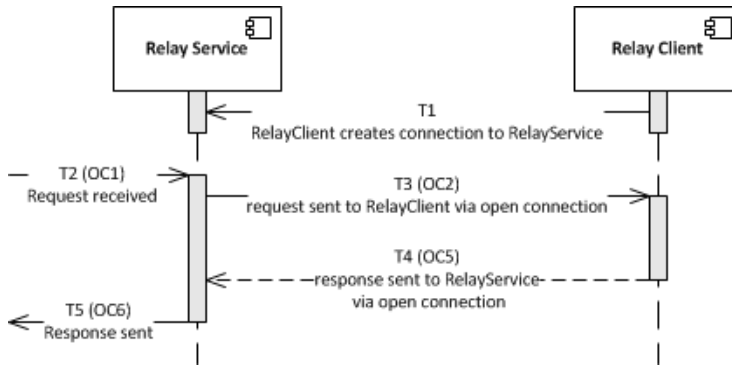


Figure 4.5: Communication flow between RelayClient and RelayService using the tunnel strategy.

the RelayService allows requests to be instantly forwarded to the RelayClient, afterwards responses are instantly sent to the RelayService and thereafter to the Customer. On the other hand establishing a connection and keeping it alive at all times possible induces overhead. On the server side this also violates the REST Statelessness constraint as information for each connection must be maintained. Keeping the connection alive and storing information on the server increases load on the server to such a degree that the server must be scalable in order accommodate multiple connections[4]. The pros of cons of the Tunnel solution are summarized below.

- + Simple communication strategy with limited amount of messages exchange.
- + Messages are instantly transferred to the RelayClient thereby creating a minimum of latency.
- Violates REST Statelessness constraint
- Connection between RelayClient and RelayService must be kept alive, this creates overhead.

bidirectional communication between client and server.

4.3.3 Push Notification

The Push Notification solution is inspired by the Push Notification Services for modern smart phones which was introduced in [2.1.3 Microsoft Push Notification Service](#). It can be considered a hybrid of the Poll and Tunnel solution as concepts of both are present. The idea is to have a standard service that forwards notifications to the remote device. For this to be possible a Push Notification Client must establish and maintain a connection to a Push Notification Server. The RelayClient can then request an endpoint on the Push Notification Server via the Push Notification Client and provide this endpoint to the RelayService. The RelayService can via this endpoint send notifications to the RelayClient via the Push Notification Server and Client. This allows the RelayService to signal to the RelayClient that a remote control request has been received and the RelayClient can fetch this request. The actual flow of this is described in figure 4.6. The Push Notification is a very generic solution that can be used not

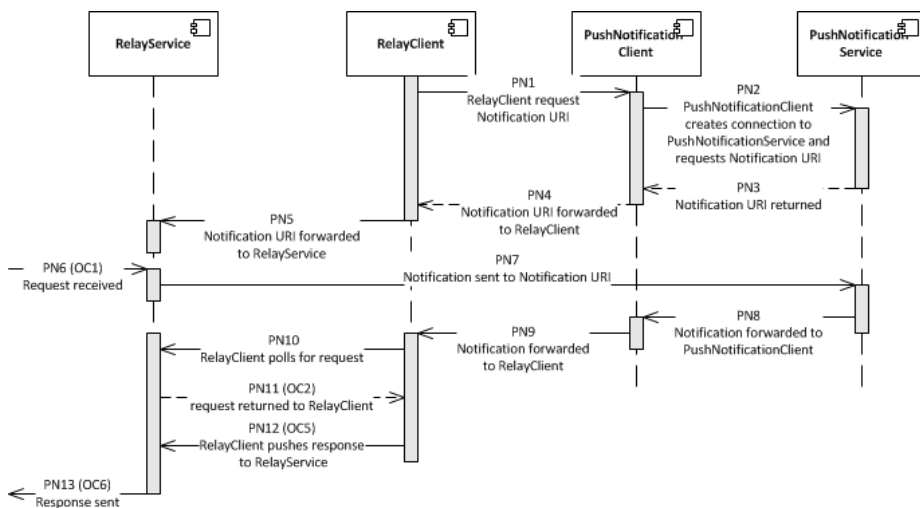


Figure 4.6: Communication flow between Client Application, RelayService, PushNotificationClient, and PushNotificationService using the Push Notification strategy.

only by the RelayClient but also by any other application on the 2250 which requires notifications from a remote third party. It does however require that a connection is established and kept alive for notifications to be passed through and the solution introduces a great deal of complexity. Below the pros and cons of the solution are summarized.

- + Notifications are instantly transferred to the RelayClient thereby creating

a minimum of latency on notifications.

- + Generic solution that can be used by other applications on the 2250.
- + The interface for polling and pushing the remote control requests and responses can be standardized.
- + Possible to separate RelayService and Push Notification Service so that RelayService is situated at the customer's data center while Push Notification Server is hosted in the cloud.
- Violates REST Stateless constraint
- After notification is received the remote control request must be fetched. This extra step increases latency.
- Complex communication strategy with more message needed to be exchanged.
- Complex solution which requires more programming resources to be implemented.
- Notification connection must be kept alive, this creates overhead.

4.3.4 Choice of Communication Strategy

The Push Notification strategy provides the most generic solution and could be used for other applications on the 2250 device. However genericness is already offered in rich measure when the RelayService combined with the RelayClient provide access to Web applications on a device. Furthermore the Push Notification requires additional messages sent back and forth. This adds complexity and increases latency when compared with Tunnel solution. The Tunnel solution offers the lowest latency. It does induce overhead by keeping the connection alive but this is not notable when compared to the overhead induce by the Poll strategy or the Push Notification strategy which also requires a connection kept alive for pushing notifications. The Tunnel solution does require communication state on the RelayService which increases the load on the service. This must be mitigated by ensuring the RelayService is scalable. Therefore the Tunnel solution has been chosen. In the following section this solution will be realized with a detailed communication protocol.

4.4 RelayService and RelayClient Detailed Communication Protocol

In the previous section the Tunnel solution was chosen as communication strategy. This strategy combined with the functional requirements and B&K wishes form the basis of the requirements for the communication protocol. Below the communication requirements (CR) are listed.

- CR1 Based on requirement MH6 the protocol must support 2250 authentication to the RelayService.
- CR2 To maintain a constant connection the protocol must give the means to keep the connection alive. To minimize computing on the 2250 following requirement NFR5 the keep alive must be initiated by the RelayService.
- CR3 B&K has requested that the protocol must initially determine the protocol version so that it can be upgraded at a later time. The RelayClient is the prime mover in the initiating sequence and should therefore also initiate the ProtocolNegotiation.
- CR4 Once authenticated it must be possible to signal the counterpart that the connection should be disconnected.
- CR5 The protocol must support that the remote control request and responses exchange. The exchange of remote control request and responses must happen asynchronously allowing request to be handled concurrently thereby improving throughput. This follows from requirement NH2.

It is possible to use existing standards to realize the communication requirements. One possibility is to create and maintain a HTTP connection from the RelayClient to the RelayService. The main arguments against choosing this protocol are: additional overhead introduced by the HTTP headers, violation of the Client Server paradigm which HTTP is based on, and the relay of HTTP requests requires filtering of HTTP messages based on their content and headers when other messages are based on HTTP as well. This introduces a great deal of complexity while reducing the flexibility of the communication.

Instead a proprietary message protocol will be developed. Based on communication specifications it is possible to define the 2250Authentication and Forward-Http interfaces introduced in section 4.1. These interfaces are shown in figure 4.7. The interfaces describe the functionality the respective components must support. The 2250Authenticate interfaces must supply following functionality;

Authenticate, NegotiateProtocol, RespondRelay and Disconnect. The ForwardHttp interface must supply the following functionality; KeepAlive, RequestRelay and Disconnect.

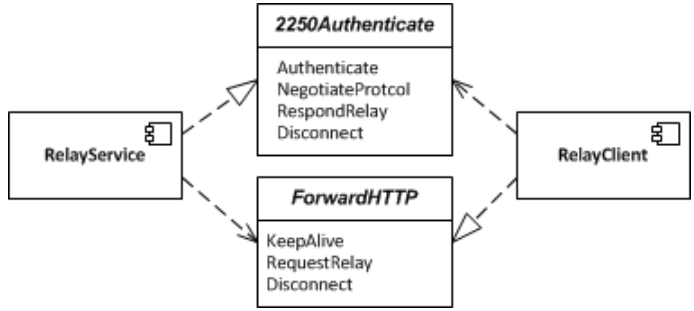


Figure 4.7: Component diagram showing the RelayService and RelayClient components and a detailed view of the interfaces that they provide to and consume from each other. Realization of the ForwardHttp and 2250Authentication interfaces present in figure 4.1

The protocol state machine in figure 4.8 describes the communication protocol design from the RelayClients perspective. This protocol version will be specified as 1.0. The protocol will use TCP as transport layer, which ensures ordering and reliability. Initially the transport level connection is established by the Relay-Client so messages can be exchanged. If the connection fails the state machine goes into its final state. If the connection is successful a ProtocolNegotiationRequest is sent to the RelayService which sends a ProtocolNegotiationResponse. These messages are exchanged to negotiate the protocol version thereby realizing CR3. If the protocol version matches this protocol version (1.0) an AuthenticationRequest message is sent, an AuthenticationResponse message is received and the state machine enters the Protocol negotiated state, otherwise it reaches the final state.

Upon reaching the protocol negotiated request the protocol state machine proceeds to the Authenticated state if the authentication was successful otherwise it reaches the final state. In the Authenticated state KeepAliveRequest and RelayRequest can be received. KeepAliveRequests are modelled as synchronous message exchange where the state machine enters the Keeping Connection Alive state and first leaves this state when the RelayClient sends a KeepAliveResponse. The RelayRequest and RelayResponses are modelled asynchronously, meaning that multiple RelayRequest can be received before a RelayResponses are sent. In other words the RelayClient can handle multiple simultaneous requests.

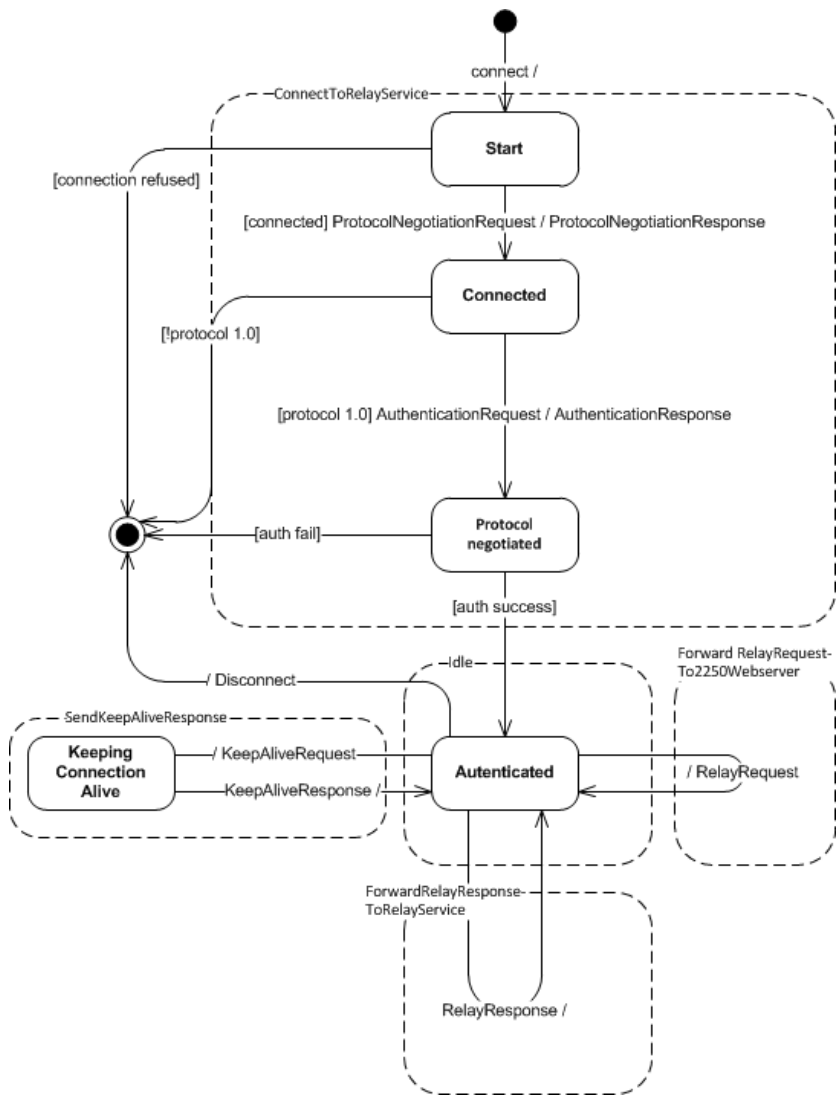


Figure 4.8: Protocol state machine specifying the communication between RelayClient and RelayService from the point of view of the RelayClient

4.4.1 Packet Types

To support the protocol specification a proprietary packet specification is needed. The protocol state machine in figure 4.8 combined with the interface descriptions

Seq. Msg. Id	Packet Type	Description
T1	(Connect)	This functionality is included in the TCP layer
T1	ProtocolNegotiation-RequestPacket	Packet containing list of proposed protocols
T1	ProtocolNegotiation-Response-Packet	Packet containing chosen protocol. The chosen protocol should be one of the proposed protocols
T1	AuthenticationRequest-Packet	Packet containing authentication credentials
T1	AuthenticationResponse-Packet	Packet containing authentication response indicating success
	KeepAliveRequestPacket	Ping message
	KeepAliveResponsePacket	Pong answer
T3	RelayRequestPacket	Packet containing a Relay Request
T4	RelayResponsePacket	Packet containing a Relay Response
	DisconnectPacket	Packet indicating a disconnect by either side

Table 4.1: Packet Types

in figure 4.7 can be used to defer the packet types needed in this communication protocol. These packet types are described in table 4.1

4.4.2 Packet Structure

The packet structure consists of a header and a payload. The TCP protocol ensures that the communication is reliable and ordered. This makes a "start of header" and "checksum" redundant information in our packet. A payload size is needed to determine the content length of a packet, this value will be referred to as PayloadSize. A value describing the packet type is needed so the packet type can be identified by reading the header, this value will be referred to as PacketType. As mentioned earlier in this section the communication protocol specifies that the RelayClient can receive a new Relay Request before it has responded to the current. To allow concurrency in the handling of requests a token is needed to uniquely identify the request/response pairs, this value is referred to as ConversationId. There is no need to further wrap the content of the packet as the content length can already be used to determine the end of a packet. Figure 4.9 depicts a packet which fulfils the combination of requirements. This is the packet structure chosen.

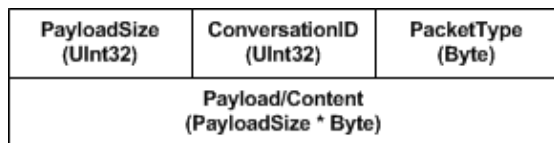


Figure 4.9: Diagram showing the structure of a packet sent between Relay-Service and RelayClient. Non UML.

The Payload content will vary depending on the packet type. An example of package content is a RelayRequestPacket. The packet content will contain a raw byte representation of the HTTP request. The PacketType of the packet will be the a byte value uniquely identifying the RelayRequestPacket, the ConversationId will be a unsigned 32 bit integer uniquely identifying the conversation and the PayloadSize will be the length of the HTTP request in bytes including the Header and HTTP request line.

4.5 Chapter Summary

In this chapter the main components in the design and their interfaces were described. The overall data flow between the components was detailed and it was chosen to address each device in the host part of a URL thereby allowing the relative path to remain identical for resources on the RelayService and the 2250 Web Server. Three communication strategies Poll, Tunnel and Push Notification were introduced and it was chosen to use the Tunnel strategy mainly due to its low latency. A detailed protocol for the Tunnel strategy has been described in a protocol state machine diagram and message types as well as message structure has been defined.

RelayClient

In this chapter the design and implementation of the RelayClient will be introduced and discussed. According to NFR1 the RelayClient implementation is bound to the Windows CE platform and the .Net Compact Framework. The overall behaviour of the RelayClient will be defined. Thereafter the different components will be introduced and key classes in the component will be described in detail. During this description the design patterns used will be introduced. The detailed behaviour of the RelayClient will be specified based on the overall behaviour, the components and the key classes. Key points and special considerations in the implementation will be discussed. The combination of the previous chapter and the present establishes the design, implementation and interactions of the RelayClient.

5.1 Behaviour

The purpose of the RelayClient is quite simple. The RelayClient must:

1. Connect to the RelayService. Based on requirement MH10 and the protocol specified in figure 4.8.
2. Receive relay requests from the RelayService and forward these requests to the 2250WebServer. Based on communication flow specified in figure 4.2 and the protocol in figure 4.8.

3. Receive relay responses from the 2250WebServer and forward these responses to the RelayService. Based on communication flow specified in figure 4.2 and the protocol in figure 4.8.

The communication between the RelayClient and the RelayService was defined in the protocol state machine in figure 4.8 page 78. The communication between RelayClient and 2250WebServer needs not be explicitly defined as it is a simple HTTP client server request response communication.

The behaviour of the RelayClient is specified in figure 5.1. The RelayClient starts by initializing. During this phase it will retrieve the credentials needed to connect to the RelayService, the RelayService address, and the RelayService port. After the device is initialized it will attempt to connect to the RelayService. In the ConnectToRelayService state in figure 5.1 all communication from the Initial state to the Protocol negotiated in the protocol state machine figure 4.8 takes place. If this fails it enters the Sleep state and waits for a given period before it enters the ConnectToRelayService state and retries to connect. Thereby fulfilling MH10 for the RelayClient.

If the connection is successful it enters the Idle state. This is the equivalent of the Authenticated state in the protocol. In the Idle state four things can happen.

1. A KeepAliveRequest can be received from the RelayService. In that case the RelayClient should send a KeepAliveResponse to the RelayService and re-enter the Idle state.
2. A RelayRequest can be received from the RelayService. In that case the RelayClient should forward the RelayRequest to the 2250Webserver and re-enter the Idle state.
3. A RelayResponse can be received from the 2250WebServer. In that case the RelayClient should forward the RelayResponse to the RelayService and re-enter the Idle state.
4. The connection can disconnect or time out. In that case the RelayService should enter the ConnectToRelayService state and attempt to establish a connection. Thereby requirement MH10.

5.2 Design

The RelayClient consists of four software components; Communication, Client-Communication, CustomHttp and RelayClient. The four component and inter-

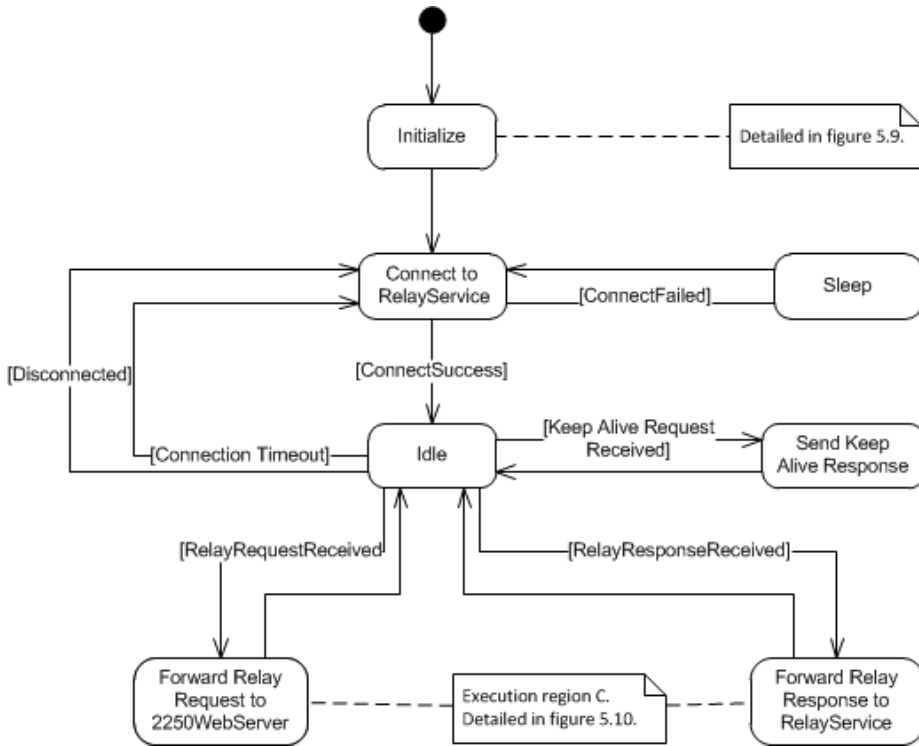


Figure 5.1: State machine defining the behaviour of the RelayClient.

relationships are illustrated in figure 5.2. These components will be specified in this section.

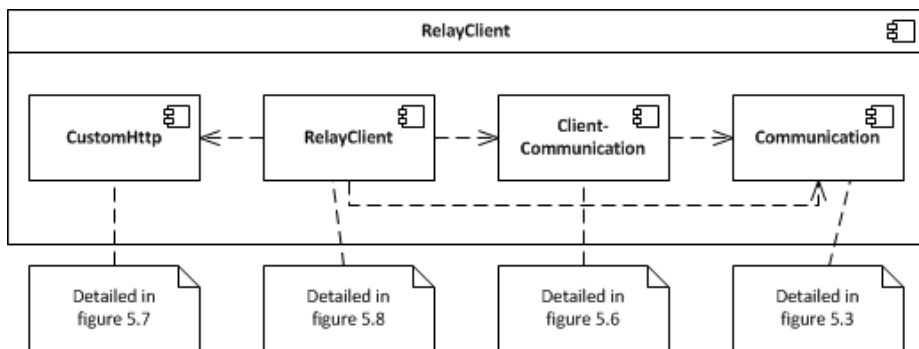


Figure 5.2: Component diagram showing the components used in the Relay-Client application.

5.2.1 Communication

The Communication component is the core component in the communication between the RelayClient and RelayService. The component contains the classes needed by both the RelayClient and RelayServer. The class diagram in figure 5.3 shows an overview of majority of classes in this component. Of these classes the most interesting are the ExtendedTcpClient, the TcpTunnel, the TcpTunnelPacket and the CommunicationFactory which will be described in detail.

5.2.1.1 ExtendedTcpClient

The ExtendedTcpClient is described in figure 5.4. As shown in the figure it realises the IExtendedTcpClient interface. The IExtendedTcpClient interface defines methods for connecting, disconnecting, and writing to the connection. The referenced delegates ConnectionClosedHandler and DataReceivedHandler are public events¹ allowing other classes to subscribe to this event. The use of delegates and events in this way is a modern *c#* architectural variant² of the Observer pattern[37]. The Observer pattern is described in depth in Gamma et al., p. 293 [22]. Using delegates decouples the Observer from the Observable and allows events from the Observable to reach the Observer with the Observable only having a function reference and not a reference to the Observer. These references must be freed whenever the Observer or the Observable are no longer used to prevent memory leaks. This is done by having the relevant classes implement the IDisposable interface and remove the reference when disposed.

5.2.1.2 TcpTunnel

The TcpTunnel is an abstract class which realizes the ITcpTunnel as shown in figure 5.5. The TcpTunnel's main objective is to parse raw data into packets and send packets via an IExtendedTcpClient subclass. The TcpTunnel registers itself with the ConnectionClosedHandler and the DataReceivedHandler events on the IExtendedTcpClient subclass so that it receives events when data is received or the connection is closed. The ITcpTunnel defines methods for writing via the tunnel and closing the tunnel. The two delegates TcpTunnelPacketReceivedHandler and the TcpTunnelChangedHandler allows other objects to register for packet received events and tunnel status changed events. Effectively allowing registered objects to be notified when a message is received. The use of this is further detailed in 5.2.5 Detailed Behaviour.

¹An event is a specific implementation of a delegate with a more limited access so that different observers cannot overwrite or cancel each others registration

²Another modern variant of this pattern is the Reactive Extensions (Rx) Framework (<http://msdn.microsoft.com/en-us/data/gg577609.aspx>).

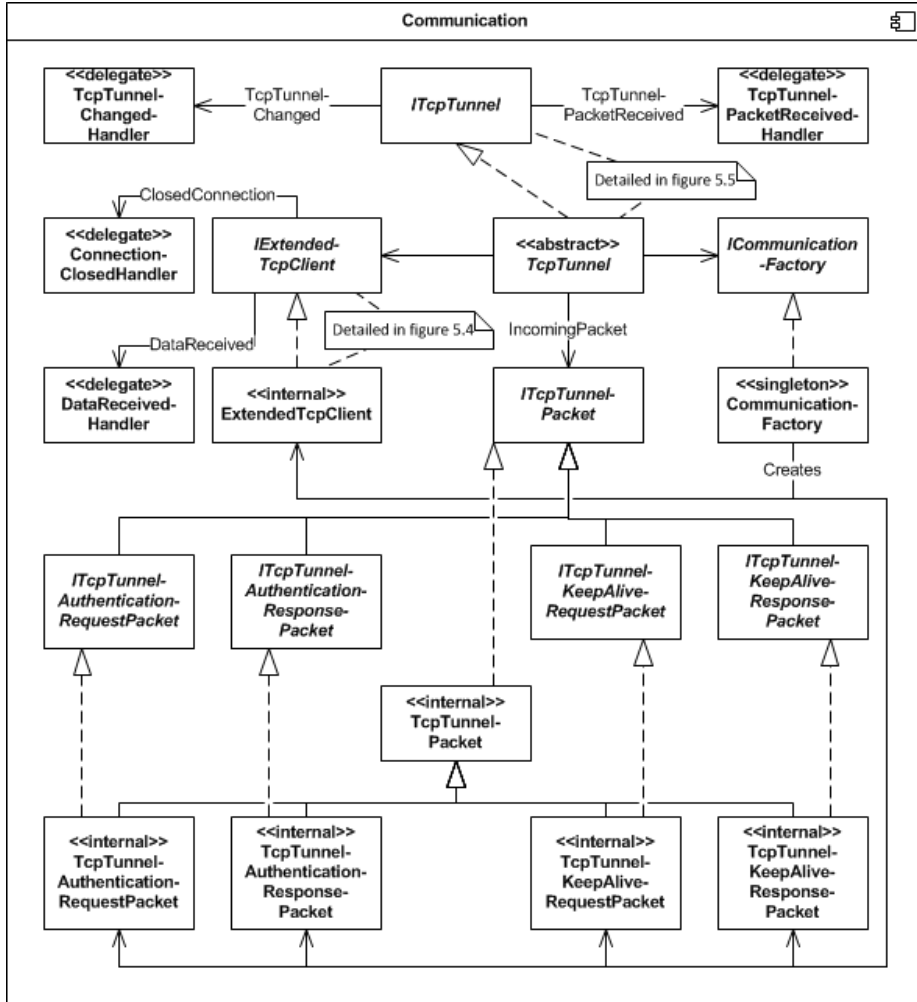


Figure 5.3: Class diagram showing an excerpt of classes and interfaces in the Communication component. The classes and interfaces can be grouped into the following types; delegates, tcp tunnel packets, factory classes and interfaces, the extended tcp client class and interface and the tcp tunnel abstract class and interface.

5.2.1.3 TcpTunnelPacket

The TcpTunnelPacket realizes the ITcpTunnelPacket and represents a packet fulfilling the packet definition from 4.4.2 Packet Structure. All the specific

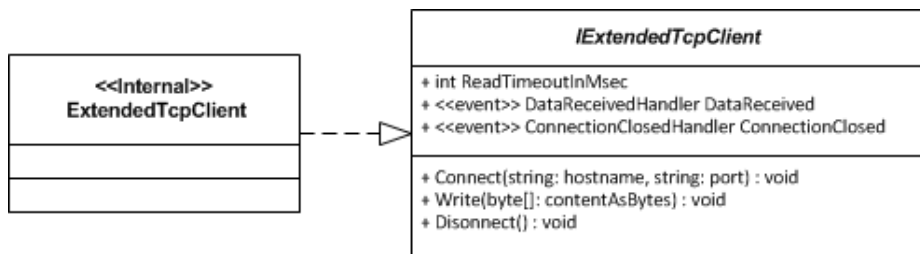


Figure 5.4: Extract of the Communication component showing the `ExtendedTcpClient` class which realises the `IExtendedTcpClient` interface.

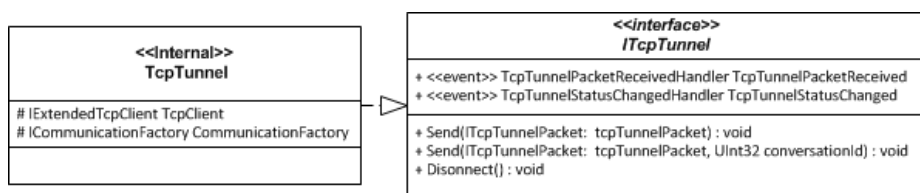


Figure 5.5: Extract of the Communication component showing the `TcpTunnel` class which realises the `ITcpTunnel` interface.

packet implementations inherit from this base class. An example of this is the `TcpTunnelAuthenticationRequestPacket` which is a subclass of `TcpTunnelPacket` but also realises the `ITcpTunnelAuthenticationRequestPacket` interface. The different packets are realizations of the packets described in [4.4.1 Packet Types](#).

5.2.1.4 CommunicationFactory

The classes in the Communication component are instantiated using the abstract factory pattern. The abstract factory defines an interface for instantiating a family of related objects without specifying their concrete subclasses [22, p. 87]. This pattern is used to create a system that is independent of how objects are created, composed and represented, to reveal just interfaces of a class library and to constrain which objects that can be used together. This allows a more loose coupling and will provide great benefit if the protocol between `RelayClient` and `RelayService` should change. In the Communication component the `ICommunicationFactory` represents the abstract factory and the `CommunicationFactory` represents the concrete factory.

The `CommunicationFactory` is a singleton. The singleton is a design pattern used to restrict the number of instantiations of a class to one. This pattern is

used because multiple factories of the same instance would introduce unnecessary overhead as a single factory is sufficient. This is an especially valid point for the RelayClient where resources are limited. The singleton pattern is described in [22, p. 127].

5.2.2 ClientCommunication

A class diagram for the ClientCommunication component is shown in figure 5.6. The ClientCommunication component provides a client implementation in the Tunnel solution. It uses the Communication component for handling basic communication while it implements the actual protocol. The component consists of two classes and two interfaces. The ClientCommunicationFactory, the TcpTunnelClient, IClientCommunicationFactory and ITcpTunnelClient. The abstract factory pattern is again used to create objects in this component. The ClientCommunicationFactory implements the IClientCommunicationFactory interface and inherits from the CommunicationFactory so this single factory can be used for all creation of communication objects. The TcpTunnelClient implements the ITcpTunnelClient interface and is a specialization of the TcpTunnel class. The TcpTunnel registers with the TcpTunnelPacketReceived event. When a packet is received the TcpTunnelClient processes the logic and triggers events via the inherited TcpTunnelChanged delegate. This enables generic use of this client.

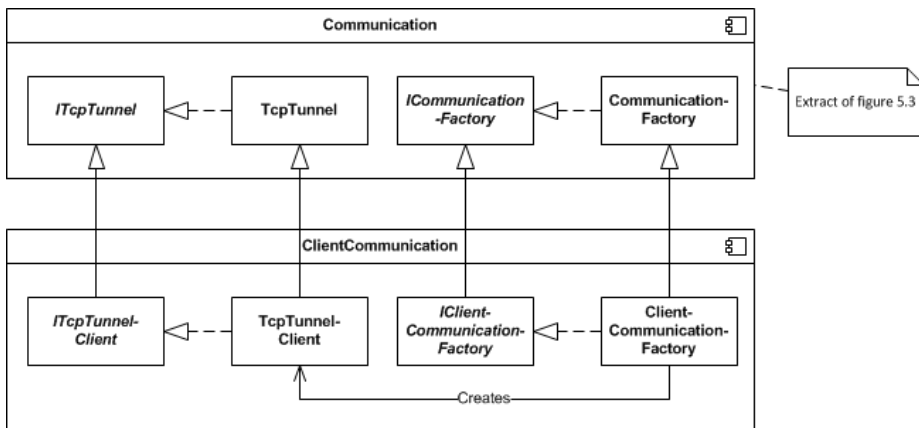


Figure 5.6: Class diagram showing classes and interfaces in the ClientCommunication component together with an extract of the Communication component

5.2.3 CustomHttp

A class diagram for the CustomHttp component is shown in figure 5.7. The CustomHttp component is used to parse and represent HTTP requests and responses. The subclasses of IHttpClient provides functionality to send HTTP requests and receive their responses. It provides asynchronous send and receive, where the combination of the HttpResponseReceived event and a unique Id for each request ensures that responses can be matched to the corresponding requests. The IHttpRequest and IHttpResponse interfaces define functionality for respectively converting IHttpRequest to a System.Net.HttpWebRequest³ and converting from a System.Net.HttpWebResponse⁴ to a IHttpResponse.

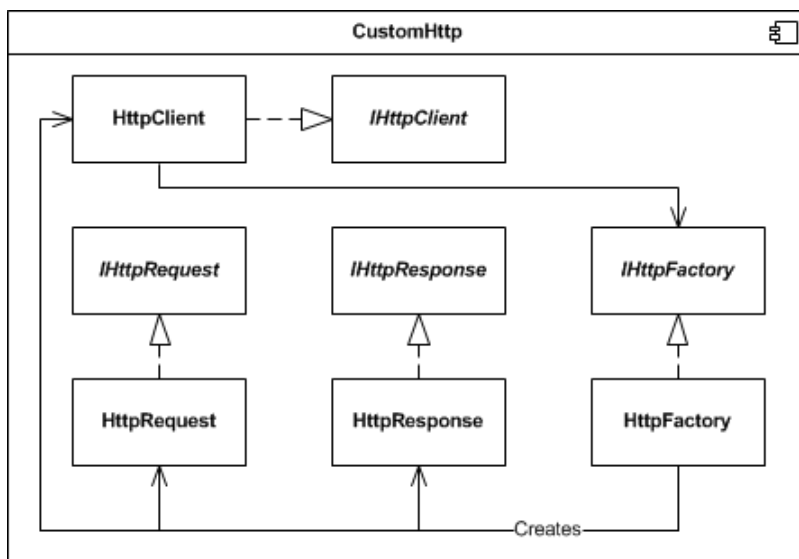


Figure 5.7: Class diagram showing classes and interfaces in the CustomHttp component.

5.2.4 RelayClient

A class diagram for the RelayClient component is shown in figure 5.8. The RelayClient component handles the behaviour and configuration of our client software. The Configuration class represents a configuration, specifying username, password and server details for the Tunnel connection. The configuration

³The format in which a request is sent to the 2250WebServer.

⁴The format responses from the 2250WebServer are received in.

is read by an `IConfigurationReader` subclass. The `RelayClient` realizes the `IRelayClient` interface which defines the two methods; `Initialize` and `Run`. The `Initialize` method takes a `Configuration` as argument and initializes the `RelayClient` for a connection. The `Run` method starts a connection and restarts the connection if it is disconnected. The `RelayClient` object controls the behaviour of the application and realizes the specific behaviour specified in [5.1 Behaviour](#). The loose coupling between the `RelayClient` and the `ITcpTunnelClient` subclass adds flexibility so that the behaviour can be changed by creating a new subclass of `IRelayClient` or the connection can be changed by providing a different subclass of `ITcpTunnelClient` to the `RelayClient`.

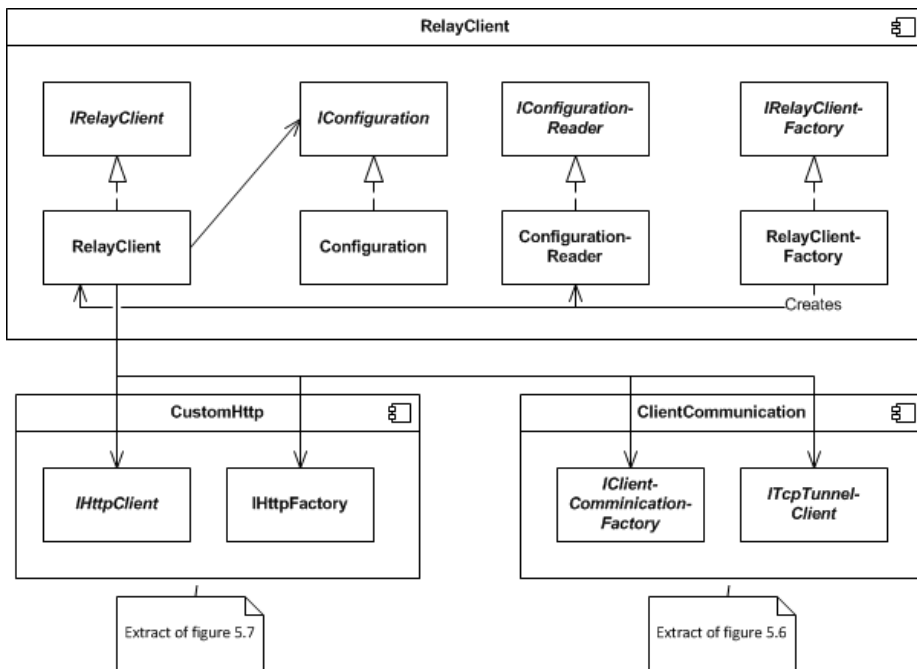


Figure 5.8: Class diagram showing classes and interfaces in the `RelayClient` component.

5.2.5 Detailed Behaviour

In this subsection the behaviour of the `RelayClient` object and its use of other classes will be described in detail using sequence diagrams. This will give an overview of the program logic.

The initialization sequence described in [figure 5.9](#) gives an overview of the ini-

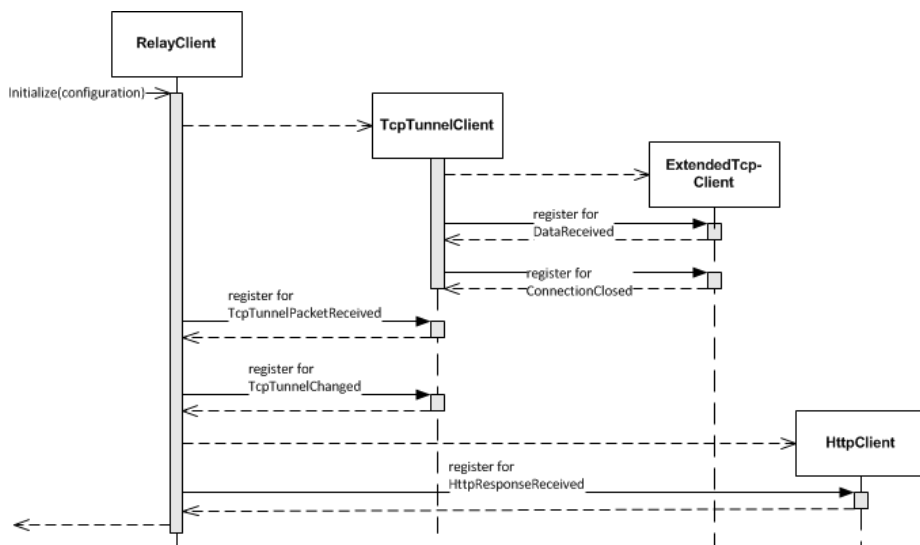


Figure 5.9: Initialization sequence for the RelayClient. The sequence diagram corresponds to the Initializing state in figure 5.1.

tialization state where objects are created and register to relevant events and delegates⁵. The RelayClient creates the TcpTunnelClient, which during its initialization creates an ExtendedTcpClient object and register for DataReceived and ConnectionClosed events at the given object. When the TcpTunnelClient is initialized the RelayClient registers for TcpTunnelPacketReceived and TcpTunnelChanged events at the TcpTunnelClient. Afterwards the RelayClient creates a HttpClient and registers for HttpResponseMessageReceivedEvents. The registration of events gives an indication of the information flow in the application. An example of such information flow is present in figure 5.10.

Data is received by the ExtendedTcpClient and this is forwarded via the DataReceived event. The TcpTunnelClient is a subclass of TcpTunnel which receives this data and translates it into packages. Once a complete package is received the TcpTunnelPacketReceived event is triggered. The TcpTunnelClient is also registered for this event and upon triggering it determines the packet type and chooses the appropriate action according to the protocol state machine. If the appropriate action causes a status change for the tunnel connection the TcpTunnelChanged event is triggered. An example of such an action is the arrival of a TcpTunnelAuthenticationResponsePacket which either indicates that the authentication succeeded or failed. Figure 5.10 shows a sequence in which the

⁵The diagram abstracts away from the fact that the individual classes are created using the abstract factory pattern.

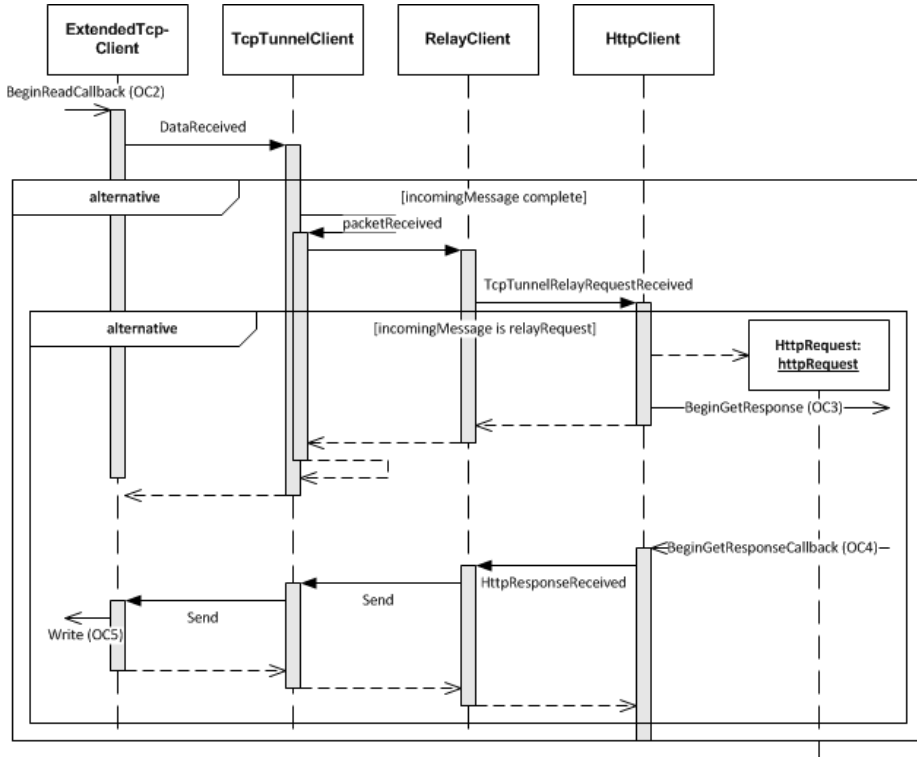


Figure 5.10: Relay Sequence showing how a request is processed and forwarded and how a response is received and forwarded. This Sequence diagram corresponds to execution region C in figure 4.2.

message received is a relay request. The RelayClient is registered to receive all message events. If the message is a relay request the request is forwarded to the HttpClient which asynchronously performs the request on the 2250WebServer. When the response is received the HttpResponseReceived event is triggered by the HttpClient. The RelayClient receives the event and forwards it via the TcpTunnelClient and ExtendedTcpClient.

5.3 Implementation

This section will present key points in the implementation of the chosen design. The three things that stand out in the implementation of the RelayClient are the special considerations due to the embedded platform, how the RelayClient

can address not only the 2250WebServer but also other local devices and how the application can be integrated in the existing BasicEnv solution.

5.3.1 Quirks of The Embedded Platform

There are some special considerations when implementing on the embedded platform. The .Net Compact framework is limited and this requires additional implementation and may require additional workarounds.

Apparently there is client side limit on the number of concurrent network connections to a host. Determining this consumed a great deal of time which is why it is mentioned here. This plays a role in the forward of requests (OC3) to the 2250WebServer from the RelayClient . This limit is not unique for the platform and a similar limitation exist on regular platforms however with a larger limit than 2 which is the default maximum on the .Net Compact Framework⁶. Having a limit on the number of connections in itself is not a problem but rather a clever design decision to prevent flooding. However the System.Net.HttpWebRequest class is used to communicate with the 2250WebServer and this combination creates trouble. In some situations the HttpWebRequest class implementation has some issues accepting that a connection should not be kept alive, even when this is indicated in the HTTP header. Two connections which are kept alive effectively blocks any other requests to the same host. The solution to this is to force the connections to close after a given idle period⁷ as seen below.

```
httpWebRequest.ServicePoint.MaxIdleTime = 50;
```

To reduce the number of active and awaiting connections and to ensure a minimum of load a semaphore in the HttpClient is used to restrict the number of active requests to two. Any requests beyond that will wait until the semaphore is released. This prevents the framework from discarding the requests due to limitations on the number of simultaneous connections to the specific service point.

5.3.2 RelayClient a Generic Proxy

In 4.2.1 [Addressable Devices](#) it was defined how the HTTP requests and responses could and should be modified to ensure that the devices were addressable. It was determined that by identifying the device in a subdomain the

⁶This specific limit is based on section 8.1.4 in *Hypertext Transfer Protocol – HTTP/1.1* [17].

⁷Default maximum idle time is 100 seconds. The example code sets it to 0.05 seconds.

relative paths would still match and need not be modified. The host field does however still state that the HTTP request is meant for the RelayService as RelayService acts as a reverse proxy⁸. It would be sufficient to simply change this value to "localhost" as the RelayClient will always be present on the 2250 itself. However changing the host field does provide a unique opportunity. As mentioned previously the 2250 is a versatile device and can be used in many setups. [2.6.1 Device/2250](#) describes how it in environmental setups can be connected to a GRPS router and/or a weather station and that it in other setups may have other hardware attached or available over the local network. By allowing the host field to specify to whom the RelayClient will forward the request to these devices can now be reached on the local network. This means that the RelayClient functions as a regular proxy and gives endless opportunities. Therefore this solution is chosen, even though the benefits are not utilized in this project. Because the addressing of device takes place on the RelayService the modification of host name should also happen there. The only change in implementation is that the HttpClient should not always connect to the "localhost" but instead use the value in the HttpRequest host field as target.

5.3.3 Integrating in BasicEnv

The RelayClient must be integrated in the existing BasicEnv solution, so that it runs at start up. As this is a proof of concept prototype it is chosen to have a minimum integration. At the point where other processes are started in the BasicEnv software an additional code section has been added which starts the RelayClient as a separate process. This level of integration is sufficient for determining the fulfilment of requirement NFR5 page 20. In the long term this should be substituted with a better integration which for example allows the RelayClient to shut down and start up automatically when the software is updated.

5.4 Chapter Summary

In this chapter the design and implementation of the RelayClient has been discussed. The overall behaviour of the RelayClient has been introduced and the four components Communication, ClientCommunication, CustomHttp and RelayClient have been described. Key classes and interfaces in the different components have been detailed. Based on this and the overall design the detailed behaviour of the initialization sequence and the detailed behaviour when a relay request is received has been defined. Noteworthy points in the implementation have been introduced. It was defined how HTTP requests needed additional

⁸Having a mismatch in the host header field will cause the server to ignore the request.

workarounds to close a connection, how a semaphore has been introduced to limit the number of connections, how the solution has been made so that it can forward requests to devices on the local network as well and how the RelayClient has been integrated with the existing BasicEnv software.

RelayService

The design and implementation options on the cloud platform relevant to this project were described [3.2 Windows Azure](#). The selections were postponed until overall design choices were made. In chapter [4 Components and Communication](#) the design decisions for the systems as a whole were made. In this chapter the combination of background knowledge from Windows Azure and the overall design choices will be combined and the design and implementation of the RelayService including the StatusInterface will be specified.

6.1 Choosing an Execution Model

In [3.2.1 Execution Models](#) the different execution models supplied by Windows Azure were discussed. It was discussed how and why solutions could be split into background and foreground processes hosted on respectively WebRoles and WorkerRoles. One of the main arguments for this was scalability. The opposing design where all processes are present on the same instance will be considered to better understand this argument.

The RelayService has three tasks it must address.

1. Handling requests for the relay interface and send responses. As defined in [4.2 Overall Communication](#).

2. Handling requests for the StatusInterface and send responses. As defined in [4.1 Component Design](#).
3. Creating and maintaining connections with RelayClients. Based on the combination of MH10 and the specified protocol defined in [figure 4.8](#).

In the case where foreground and background processes are not separated a single role will handle all three tasks. When all tasks are handled by a single instance, communication between instances is unnecessary and the design choice provides a benefit. However consider if the load increases and additional instances are spun up. The RelayClient will be connected to a single instance. Due to the load balancing described in [3.2.1 Execution Models](#) requests for a specific device would not necessarily reach the instance connected to the device. Therefore the request must be forwarded between the instances and the initial benefit of no communication between instances are lost. Also consider the case where increased load by the background processes would cause the foreground processes to become unresponsive. This would violate the MH7 requirement (page 17) of always sending a response to the customer. Separation of the foreground and background processes mitigates this risk and provides a good design where individual roles have well defined responsibility and scope. The RelayService is separated into the WebRole RelayServiceFrontend and the worker role RelayServiceBackend. The RelayServiceFrontend receives requests for the exposed interfaces from the Customer. The RelayServiceBackend handles connections and communication with RelayClients. To process an actual request communication between the front end and back end is needed. This communication will be discussed in the following subsection.

6.1.1 Choosing a Messaging system

The options for communication between instances in Windows Azure has been described in [3.2.2 Messaging](#). The choice of direct network connection is not viable. When multiple instances are running each WebRole must be connected to each WorkerRole. The cost of maintaining a high number of connections is too high. The choice therefore stands between the Azure Queue, the Service Bus Queue and the Service Bus Topic/Subscription. The message data exchanged between the front end and back end will represent HTTP messages. The messages should be processed immediately and need not be stored for longer periods of time. The actual size of HTTP messages is not known, but allowing larger messages is more desirable. It is also relevant that messages always have an intended receiver. Messages from the front end must reach the back end instance that has a connection to the specific device. Messages from the back end must reach the front end instance that send the original request so that it can forward the response to the customer.

Considering this the Service Bus Topic/Subscriptions have been chosen. It has the advantage that messages can be sent to central topic and simply be addressed using an Id as correlation filter allowing a fast hashtable look up. This provides a great deal of simplification in the design process. In comparison choosing a queue based solution would require that each message receiver created its own queue to ensure that it will read only the messages and all the messages intended for it. This also requires multiple outbound queue connections. Again this would give rise to a situation where the number of references increase dramatically as the number of instances increase similar to the number of connections in the Direct Connection.

6.1.2 RelayServiceBasic Design

To accommodate the new design the RelayService component is split into the RelayServiceBackend and the RelayServiceFrontend. The basic functionality is moved into a third component called RelayServiceBasic. The RelayServiceBasic consists of classes specifying the content exchanged between the front end and back end together with the classes used for exchanging the content. The diagram in figure 6.1 shows the new structure of the component where both the RelayServiceFrontend and the RelayServiceBackend use the RelayServiceBasic to exchange data over the Service Bus.

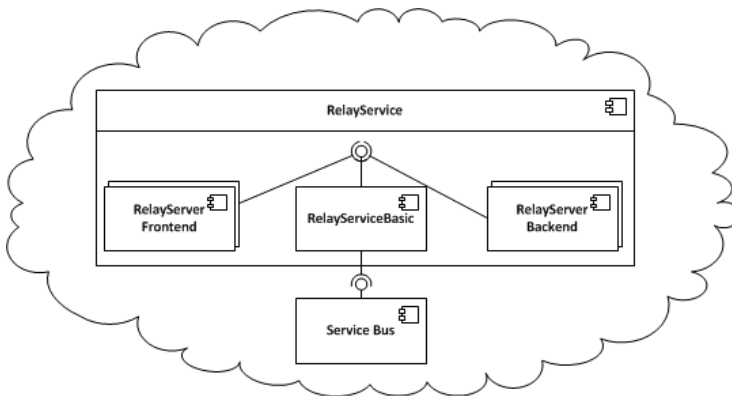


Figure 6.1: Component diagram showing the relationship between Service Bus, RelayServiceFrontend, RelayServiceBackend, and RelayServiceBasic. The components are placed within the same cloud platform.

The class diagram showing classes relevant for content exchanged via the Service Bus is shown in figure 6.2. The content exchanged is represented by the generic IRelayResponse and IRelayRequest interfaces. In this application

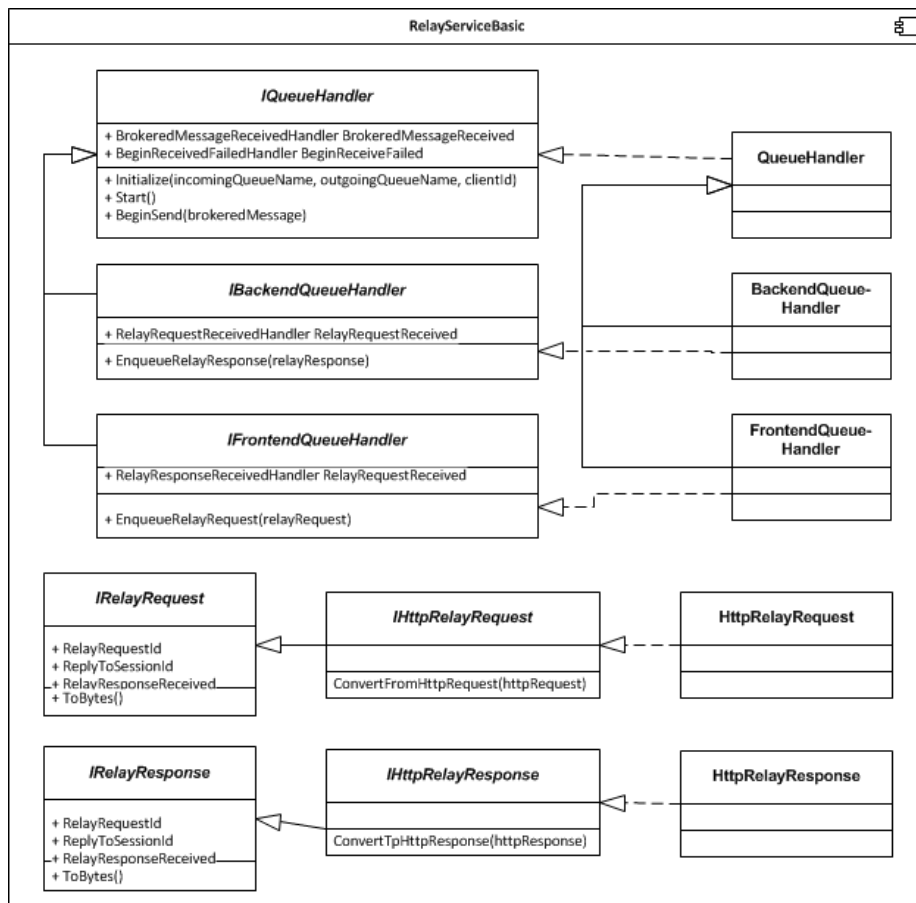


Figure 6.2: Class diagram showing the relevant interfaces and classes for relay messages in the RelayServiceBasic component.

the `IRelayRequest` will always represent a HTTP request and the `IRelayResponse` will always represent a HTTP response. To represent this specialization the `IHttpRequest` and `IHttpRelayRequest` interfaces are used. The `IHttpRequest` defines the `ConvertFromHttpRequest` which enables conversion from `System.Net.HttpRequest`¹ to `IHttpRequest`. The `IHttpRelayResponse` defines the `ConvertToHttpResponse` which enables conversion from an `IHttpRelayResponse` to `System.Net.HttpResponse`². These conversions are defined because the object received when a Customer send a request is `System.Net.HttpRequest` and the object send to the Customer as a response is of

¹This is the format that the request is originally in.

²This is the format used to send a response to the Customer.

the type `System.Net.HttpRequest`. The `IHttpRequest` and `IHttpRequestResponse` are realised by the respective classes `HttpRequest` and `HttpRequestResponse`.

The `RelayServiceFrontend` uses the `IFrontendQueueHandler` interface to communicate via the Service Bus. The `IFrontendQueueHandler` defines the method `EnqueueRelayRequest` and the event `RelayResponseReceived`. The `IFrontendQueueHandler` is a specialisation of the `IQueueHandler` which defines general methods and properties for queue handling. The `IBackendQueueHandler` likewise inherits from `IQueueHandler`. This interface is used by the back end to communicate via the Service Bus. The `IBackendQueueHandler` defines the method `EnqueueRelayResponse` and the event `RelayRequestReceived`.

The `IFrontendQueueHandler` and `IBackendQueueHandler` are realised by the respective classes `FrontendQueueHandler` and `BackendQueueHandler`. The two classes both inherit from `QueueHandler` which realises the `IQueueHandlerInterface`. The `QueueHandler` class contains the actual logic used for connecting to the Service Bus via the Topic/Subscription strategy.

The `QueueConnector` uses the `clientId` provided in the `Initialization` method to create a subscription. The `clientId` will be used as a `CorrelationFilter` meaning that all messages with a `CorrelationId` matching the `clientId` will be read by this specific `QueueConnector` and forwarded via the `FrontendQueueHandler` or `BackendQueueHandler`. The use of `CorrelationId` as a filter ensures a faster dispatching of messages as the recipients can be looked up in a hash table in constant time.

6.2 RelayServiceBackend Design and Implementation

The components related to the back end design are shown in figure 6.3. The top component in the back end design is the `RelayServiceWorkerRole`. It is a worker role as described in 3.2.1.3 [WorkerRole](#) and is responsible for starting the background processes. The `RelayServiceBackend` component is responsible for accepting incoming connections from `RelayClients` and handling the collection of connections. The `RelayServiceBasic`, `CustomHttp` and `Communication` components have already been described in respectively 6.1.2 [RelayServiceBasic Design](#), 5.2.3 [CustomHttp](#) and 5.2.1 [Communication](#). The `ServiceCommunication` component provides a server implementation in the Tunnel implementation. It uses the `Communication` component for basic communication while it implements the actual protocol. It is the server equivalent of the `ClientCommunication` described in 5.2.2 [ClientCommunication](#). The `Authentication`

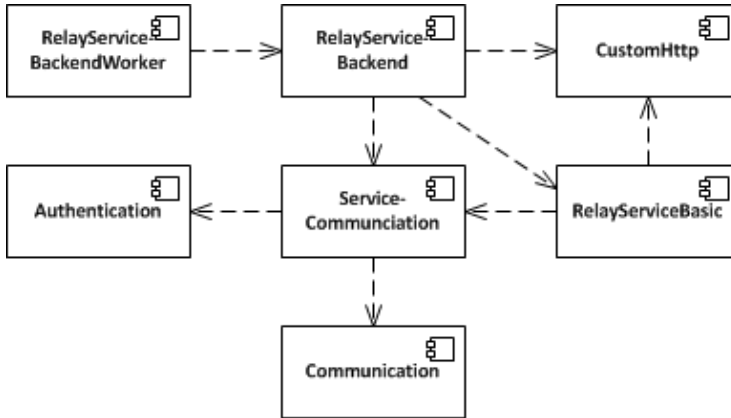


Figure 6.3: Component diagram showing the relations between relevant components from the back end design point of view.

component provides functionality for authenticating the RelayClient when it initiates a connection.

The class diagram in figure 6.4 shows the most important classes in the back end design. The RelayServiceWorkerRole class implements the worker role functionality. It is responsible for creating an IRelayServerController for each possible endpoint³. The RelayServerController realizes the IRelayServerController and handles incoming connections from RelayClients. When a connection is established the RelayController creates an IRelayServer (via the factory) and initializes it. The RelayServer realizes the IRelayServer interface. It contains the logic defining how the RelayService handles a single Tunnel connection as well as a reference to an IBackendQueueHandler which it uses to receive relay requests and send relay responses. The Tunnel connection is realized via the TcpTunnelServer class which realises the ITcpTunnelServer interface and inherits from TcpTunnel. An IAAuthenticator is passed to the TcpTunnelServer from the RelayServer. This IAAuthenticator class authenticates the credentials supplied by the RelayClient⁴.

The sequence diagram in figure 6.5 shows the control and data flow when an IRelayRequest is received. The data flow in the RelayService follows a structure similar to the one in RelayClient and is event based. Initially the RelayServer registers with the RelayRequestReceived event at the IBackendQueueHandler.

³In this case an endpoint equals a given port on which a RelayClient can communicate with the RelayService.

⁴In the proof of concept prototype this is realized but the Authenticator class which connects to a database and compares the query results with the provided credentials

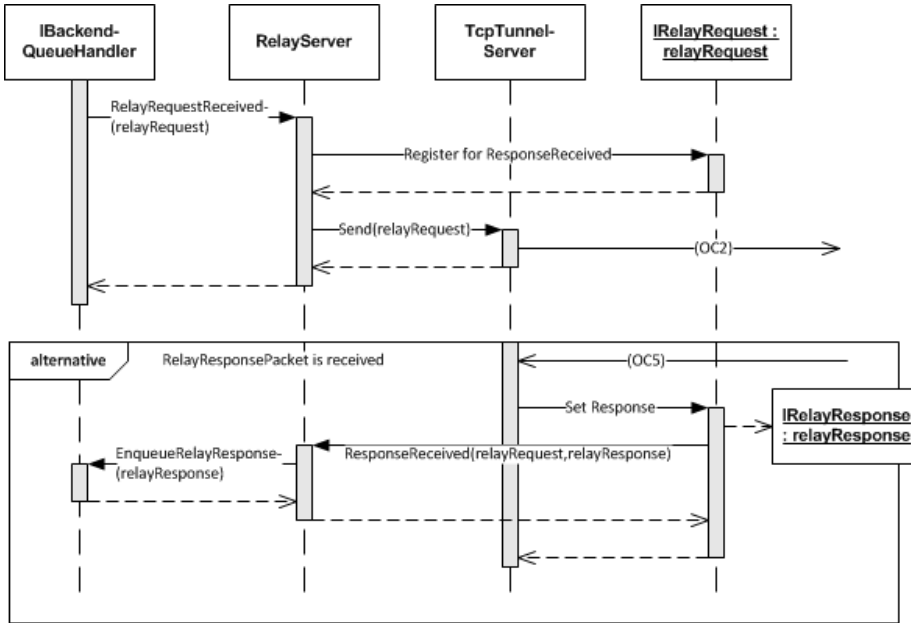


Figure 6.5: Sequence diagram showing the control logic and data flow for the back end when a relay request is received. Contained in Execution region B figure 4.2.

6.2.1 RelayServiceBackend Implementation

One of the implications of the design specified in the figure 6.5 is that the `TcpTunnelServer` must save the `IRelayRequests` it sends in a collection so that it can match the responses with the given requests. In the implementation the `TcpTunnelServer` has a `List` called `_unrespondedMessages` which is precisely such an collection.

6.3 StatusInterface Design and Implementation

The `StatusInterface` aims to provide customers with an overview of a 2250 status independent of its online status, thereby fulfilling requirement MH3 (page 17). The `StatusInterface` is based on the resources identified in 2.6 Domain Analysis. The five resources which will be present in this interface are `Customer`, `Registration`, `Device`, `Browser Interface` and `RemoteAPI`. The relation between these resources can be seen in figure 6.6. The `Device` resource will be represented by the status as described in 2.6.9 Status. According to the REST Addressable

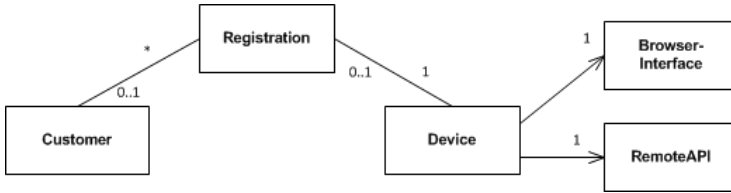


Figure 6.6: Class diagram showing the resources in the StatusInterface and their relations.

constraint the resources must be uniquely addressable. In [4.2.1 Addressable Devices](#) a design was chosen so that the 2250WebServer was addressable and the RemoteAPI remained RESTful. As a result of this the Browser Interface also remains addressable. The three new resources will be addressed using the definition described in [table 6.1](#). The hierarchical address structure shows that a Registration belongs to a Customer.

Resource	URI
Customer	/api/customers/<userId>
Registration	/api/customers/<userId>/RegisteredDevices/<deviceId>
Device	/api/devices/<deviceId>

Table 6.1: Address for StatusInterface resources.

The StatusInterface is realized using an ASP.NET MVC 3 design which follows the Model-View-Controller pattern. The class diagram in [figure 6.7](#) shows the structure of the solution. The CustomerController controls requests for the Customer and the Registration resources while the DevicesController controls requests for the Device resource. When a requests is made to the Web server hosting the StatusInterface, predefined functions are invoked in the respective controllers based on registered routes. For example if /api/devices/1234553 is requested at the Web server the function Device in the DeviceController with the parameter deviceId=1234553 will be invoked.

It is possible to provide a filter restricting access to the functions in a controller based on HTTP method. If the Uniform Interface constraint ([page 61](#)) is honoured and the HTTP methods POST, GET, PUT and DELETE maps to CRUD, the filtering can be used to define what type of operations can be made on a given resource via the controller.

The individual resources are represented in the model by the classes DeviceResource, RegistrationResource and CustomerResource. These are accessed via their respective managers DeviceManager, RegistrationManager, and Customer-

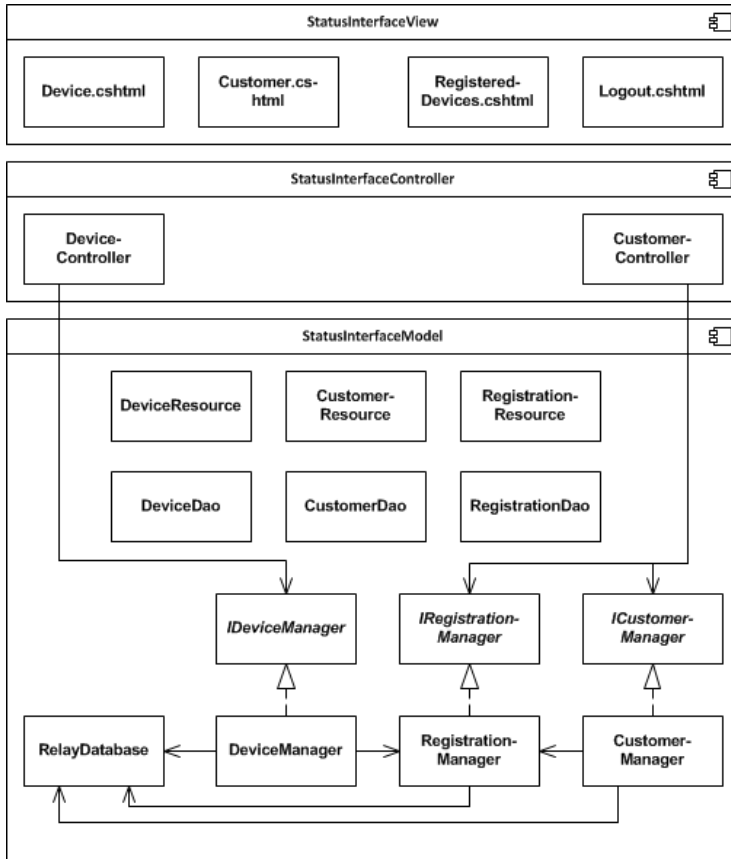


Figure 6.7: Class diagram of the StatusInterface.

Manager and the data access objects DeviceDao, RegistrationDao, and CustomerDao. Following the REST constraints the managers only allow a subset of CRUD operations on the resources. Function calls to the Controllers will respond with representations of the DeviceResource, RegistrationResource and CustomerResource objects. The Accept header in the HTTP request is used to determine the actual representation sent. This implementation supports text/xml, application/json and if they are not supplied it will send a text/html response. The text/html response is a regular view which allows users to interact with the resources from a browser as described in [2.7 Mockups](#). Screen shots of the text/html representation of the resources can be found in [Appendix B](#).

Basic HTTP authentication is required to protect the resource and authorization is determined based of these credentials on a per user basis. A Customer can

only see his own CustomerResource and only see Registrations made by himself. The Devices are likewise protected so that only Devices which are registered by the Customer can be viewed by the Customer.

To improve performance of the StatusInterface, caching is integrated in the implementation. Caching using the HTTP/1.1 standard was discussed in [3.4.7 Cache](#). In the current domain and with the user interaction specified by the use cases it is hard to predict when a resource is updated or changed. The CacheControl header, which states how long a response is valid, is uneffective in this case. Therefore the ETag header, which uniquely determines entity versions, will be used to introduce caching. As mentioned in [3.4.7 Cache](#) the Vary field can be used to describe what other header fields in a HTTP request will cause the response to vary. It was determined that the resource representations will depend on the Accept header, therefore the Vary will be defined to Accept to signal that a different Accept header may result in a different response. As an ETag for the Device resource the time for last update of the resource will be used while a hashed value of a customers registrations will be used as ETag for the Customer resource.

This sums up the design of the StatusInterface. Throughout the design it has been specified how the solution uses the HTTP standard and how this relates to REST. Table [6.2](#) sums up how the StatusInterface fulfils the Statelessness, Uniform Interface, Connectedness and Addressable Rest constraints.

Constraint	Compliance
Client Server (page 58)	HTTP as used fits the Client Server paradigm with the devision of responsibilities as specified in the REST constraint
Statelessness (page 60)	No communication state is stored on the StatusInterface.
Addressable (page 59)	Each resource will have a URI and is therefore addressable.
Uniform Interface (page 61)	Resources are accessed and manipulated using HTTP methods and the content is self descriptive
Connectedness (page 63)	Associations between resources will be represented by links.
Layered System (page 63)	The design complies with the Layered System constraint as this is in built in the HTTP communication.
Cache (page 63)	Caching is supported by the use of the ETag and Vary fields in the HTTP response

Table 6.2: Table showing compliance with REST constraints.

6.3.1 Choosing Storage Type

In [3.2.3 Data Management](#) the different storage types in Windows Azure were introduced. Earlier in this section the resource representation and their class representations have been introduced, these will serve as basis for judging what storage type will be the optimal choice. The three resource that needs to be present in the data storage are the customer, the device and the registration. The two resources customer and device will be accessed independently. This combined with the one to many relationship between customer and registration and the one to one relationship between registration and device results in separation of customer and device into different tuples that should have a relation between them.

The data model requires limited amount of storage. The number of customers and registrations can not exceed the number of devices. The devices are representations of physical hardware and the number of such devices will be lower than 100k (high estimate). Because a limited amount of data is stored for each customer, device and registration and the number of devices and customers will be relatively small, the storage needed will be limited as well. The 150GB limit on SQL Azure will not be an issue. Furthermore the storage model will not likely change or need to be dynamic for the status information. Combining all these observations leads to the conclusion that the benefits of Azure Table are not needed in this project and instead the SQL Azure will be chosen because of the server side processing, the relations between tuples and the charging model.

The implications of this choice in regards to Brewers CAP theorem will be discussed in [7.4 Consistency, Availability, Partition-Tolerance and Scalability](#).

6.4 Frontend Design and Implementation

The class diagram in figure [6.8](#) show the important classes in the front end design and their relations. The SubdomainModule implements the IHttpModule interface. An IHttpModule is a class that can be used to catch all HTTP requests and filter them. In the SubdomainModule this is used to specify what action should be taken by the front end based on the subdomain. This is shown in the sequence diagram in figure [6.9](#). If the subdomain matches the one specified in [4.2.1 Addressable Devices](#) to access the RemoteAPI and Browser Interface the module invokes the ProcessRequest method on the RemoteControlHandler class. The RemoteControlHandler processes the request wrapping it as an IHttpRelayRequest and changing the host name to the appropriate host as specified in [5.3.2 RelayClient a Generic Proxy](#). After this the IHttpRelayRequest is forwarded to the RelayServiceBackend via the IFrontendQueueHandler. The RemoteCon-

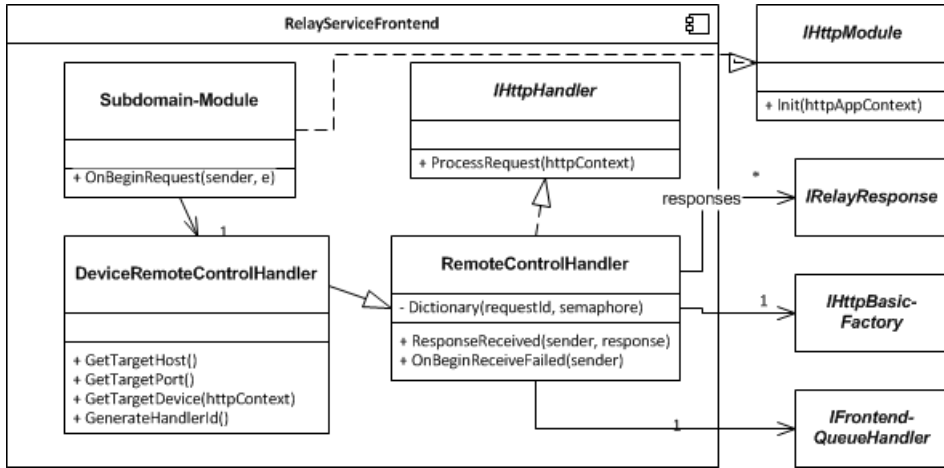


Figure 6.8: Class diagram showing the classes and interface relevant for the front end RelayService to function as a reverse proxy.

rolHandler creates a semaphore stores this in a Dictionary that maps from the request id to the semaphore and afterwards waits a certain time period for the semaphore to be freed. If the semaphore is not freed within the given time period the request times out and a 404 is sent to the Customer. If the IFrontendQueueHandler receives an IRelayResponse this triggers the IRelayResponseReceived event. The ResponseReceived method is invoked on the RemoteControlHandler. The RemoteControlHandler looks up the semaphore based on the Id of the relay request. The semaphore is freed and the response is sent to the Customer. In this design each specific handler represent an entry point to a different host. In the current setup the only addresses host is the 2250WebServer and therefore the only handler present is the DeviceRemoteControlHandler which inherits from the RemoteControlHandler. For the DeviceRemoteControlhandler the host will be "localhost" as all requests are targetted at the local 2250WebServer.

6.5 Chapter Summary

In this section the design and implementation of the RelayService have been specified. The RelayService was split into a WebRole and a WorkerRole based on the nature of the processes present in the RelayService. The Service Bus Publish/Subscription has been chosen for communication between the front end and back end processes. The classes needed for this communication have been separated into the RelayServiceBasic component. The RelayServiceBackend representing the WorkerRole has been defined and the behaviour specified via

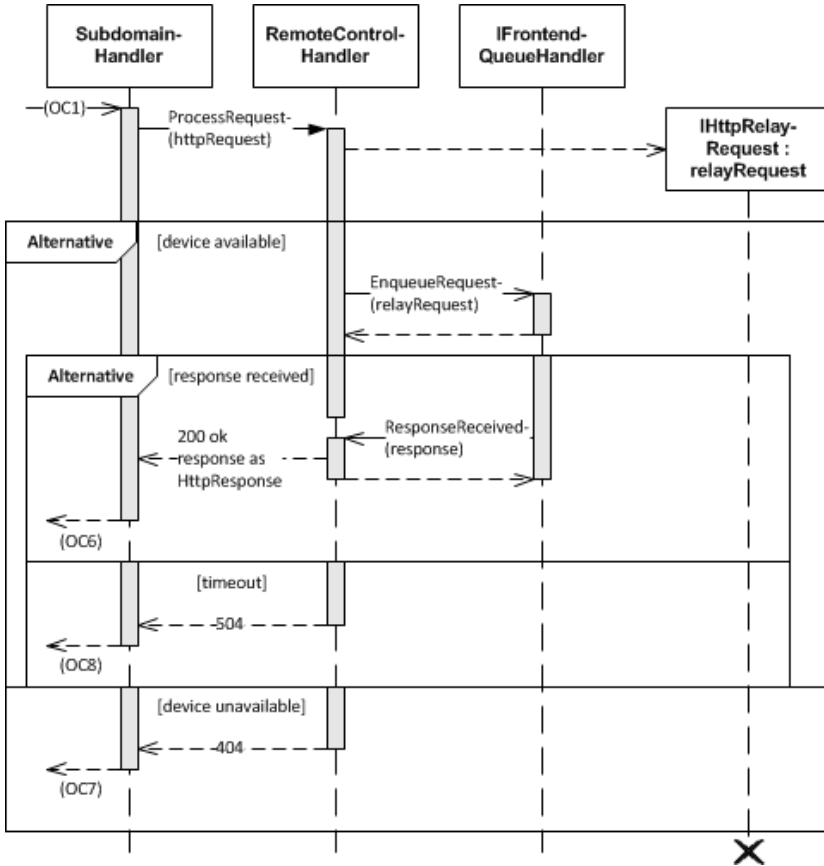


Figure 6.9: Sequence diagram showing the control logic and data flow for the front end when a relay request is received. Contained in Execution region B figure 4.2.

sequence diagram. In relation to defining the StatusInterface design and implementation the SQL Azure was chosen as storage type. The RESTful StatusInterface was specified and it was discussed how it satisfies the REST constraints. Last but not least the remaining front end design and implementation has been specified and the behaviour detailed.

Discussion

This chapter will describe how the implementation is validated and describe the acceptance tests used to evaluate the design. The performance acceptance tests will determine how the solution functions over time as well as estimate the increased latency introduced by the solution. Security in the application will be discussed and future work in this area identified. It will be discussed what changes should be made to the solution if it were to be hosted locally instead of in the cloud. The design will be evaluated in relation to Brewers CAP theorem and finally the solution as a whole will be evaluated.

7.1 Validating the Solution

White box unit tests and black box integration test have been implemented to validate the prototype throughout the development process. The unit tests have been focussed on testing object creation in the factories and the object conversions. Via the reverse proxy and the StatusInterface, black box integration tests have been conducted to test the integration between the different components in the project and their behaviour when combined. Both unit tests and integration tests can be found in Appendix C. To validate the functionality of the prototype versus the requirements, acceptance tests have been made. In this section acceptance tests for functionality and acceptance tests for performance will be described.

7.1.1 Functionality Acceptance Test

Acceptance tests determine whether the requirements of the project has been met. These tests will be performed to determine to what extent the RelayService fulfils the functional requirements. The acceptance tests are based on the use cases. Section 2.5 identified how the use cases cover the functional requirements, therefore successfully going through a use case ensures that the covered requirements are met. The tests themselves are manually performed using the Browser Interface and the text/html representation of the StatusInterface. The results of these tests can be seen in table 7.1.

The non-functional requirements are based on other metrics, specified in the requirements themselves. Table 7.2 shows and describes how the non-functional requirements are met. The requirement NFR3 (page 20 requires additional processing to evaluate. This will be done in the following section where the performance is measured.

7.1.2 Performance Acceptance Test

In this section the latency and stability performance metrics of the reverse proxy functionality will be evaluated. The stability test will determine how the RelayService functions over time. In the Browser Interface the GUI is continuously updated by sending HTTP requests via the AJAX technology. Over a time period of 24 hours it will be measured how many of the requests which are processed by the RelayService are successful (200 status code) and how many that are not. This is done by implementing a log on the RelayService which identifies the status code and saves this in a database. Due to the fact that the requests are enumerated it is possible to determine the percentage of requests that reach the RelayService and are processed. The results of this test are presented in table 7.3.

Establishing and understanding the latency introduced by the RelayService compared to a direct connection is a complex matter. The latency in the direct connection (*DLatency*) consists of the time it takes for the request and response to travel over the network (*NTime1*) and the time it takes for the 2250WebServer to process the request (*PTime*).

$$DLatency = NTime1 + PTime \quad (7.1)$$

The latency for the RelayService (*RSLatency*) consist of the time it takes for the request and response to travel between RelayService and Customer(*NTime2*),

Requirement	Status	Note
MH1 (Communication is facilitated by central service)	✓	Tested manually be realizing use case <i>Send Remote Control Command</i> .
MH2 (Expose existing Web interfaces as is)	✓	Tested manually be realizing use case <i>Send Remote Control Command</i> .
MH3 (Status information is available even when device is offline)	✓	Tested manually be realizing use cases <i>Update 2250 Status</i> and <i>See 2250 Status</i> .
MH4 (Overview of registered 2250)	✓	Tested manually be realizing use case <i>See 2250 Overview</i> .
MH5 (Access authentication on 2250)	✓	Tested manually be realizing use case <i>Send Remote Control Command</i> .
MH6 (2250 authenticates to RelayService)	✓	Tested manually be realizing use case <i>Authenticate 2250</i> .
MH7 (Response always sent by RelayService)	✓	Tested manually be realizing use case <i>Send Remote Control Command</i> .
MH8 (Customer is limited to seeing his own 2250s)	✓	Tested manually be realizing use case <i>See 2250 Overview</i> .
MH9 (Customer authenticates to RelayService)	✓	Tested manually be realizing use case <i>Authenticate Customer</i> .
MH10 (RelayService and RelayClient constantly seek connection)	✓	Tested manually be realizing use case <i>Authenticate 2250</i> .
NH1 (Customer can register unregistered 2250)	✓	Tested manually be realizing use case <i>Register 2250</i> .
NH2 (RelayService can handle concurrent requests)	✓	Tested manually be realizing use case <i>Send Remote Control Command</i> .
NH3 (RelayService handles multiple active Customers)	⚠	Not sufficiently tested. Large scale tests are needed to validate this requirement.
NH4 (RelayService handles multiple active 2250s)	⚠	Not sufficiently tested. Large scale tests are needed to validate this requirement.
NH5 (Encrypted communication)	✗	Functionality not implemented.
NH6 (Software automatically switches to direct connection between customer and 2250 whenever possible)	✗	Functionality not implemented.

Table 7.1: Fulfilment of functional requirements.

Requirement	Status	Note
NFR1 (RelayClient runs on 2250 platform)	✔	All <i>must have</i> requirements are fulfilled with the RelayClient running on the 2250 device.
NFR2 (RelayService not hosted in-house)	✔	Windows Azure is used to host the RelayService.
NFR3 (Max. 5 seconds added latency compared to direct connection between Customer and 2250)	⚠	Not sufficiently tested. This requirement will be addressed in the following section.
NFR4 (Minimize bandwidth)	⚠	Not tested. The communication between RelayClient and RelayService is designed to minimize overhead. However to validate this requirement the RelayClient should run concurrently with the NMT streamer software which utilizes a great portion of the available bandwidth.
NFR5 (Minimal CPU and memory usage)	✔	The RelayClient is integrated into the BasicEnv software and the <i>must have</i> requirements are fulfilled with both applications running concurrently.
NFR6 (Prototype does not compromise existing software on device)	✔	The RelayClient is integrated into the BasicEnv software and the <i>must have</i> requirements are fulfilled with both applications running concurrently.

Table 7.2: Fulfilment of non-functional requirements

	Successful	Unsuccessful	Unprocessed	Total
Number	41844	6	0	41850
Percentage	99.99%	0.01%	0.00%	100%

Table 7.3: Stability test

the time it takes to process the request at the front end (FERQ), the time it takes to process the request at the front end (FERS), the time it takes to send response and request via the Service Bus (SB), the time it takes for the back end to process the request (BERQ), the time it takes for the back end to process the response (BERS), the time it takes to send the request and response via the tunnel (TunnelTime), the time it takes to process the request at the RelayClient (RCRQ), the time it takes to process the request at the RelayClient (RCRS), the time it takes to send the request to the 2250WebServer (NTime3) and the time it takes for the 2250WebServer to process the request (PTime).

$$\begin{aligned}
 RSLatency = & NTime2 + FERQ + FERS + SB + BERQ + BERS \\
 & + TunnelTime + RCRQ + RCRS + NTime3 + PTime \quad (7.2)
 \end{aligned}$$

To simplify this the processing time in RelayService, the RelayClient and the send time in the Service Bus will be combined in the RSPTIME variable. Furthermore the assumption will be made that NTime3 is negligible as this is a local connection on the same device.

$$RSLatency = NTime2 + RSPTIME + TunnelTime + PTime \quad (7.3)$$

Based on this the extra latency introduced can be written as

$$\begin{aligned}
 IncreasedLatency &= RSLatency - DLatency \\
 IncreasedLatency &= NTime2 + RSPTIME + TunnelTime - NTime1 \quad (7.4)
 \end{aligned}$$

From this it is clear that the added latency cannot be attributed to the processing time alone, but also varies with the network time. The increased latency will be greater the closer the Customer and 2250 are compared to the RelayService. For example if the 2250 and the Customer are both in Alaska and the RelayService is hosted in Northern Europe the direct connection allows lower network latency while the RelayService requires that data is sent from Alaska to Northern Europe to Alaska and then back again, effectively ramping up the latency. This complicates the assessment of fulfilment of requirement NFR3 as it varies with network connections and locations. The increased latency will be determined in a single set up where the 2250 and the Customer is on the same local network placed in Denmark while the RelayService is hosted in Northern Europe. The latency will be measured as a round trip time by using the Firefox Firebug plugin. This is placed on the Customer side and if it increases latency it will do so equally for both tests. The requests are HTTP GET requests performed with the AJAX technology in the Browser Interface. The increased latency is determined by sorting the two sets based on their latency and subtracting one from the other. The data is available in Appendix D.

The histogram of the data combined with the a bell curving describing a normal distribution based on the variance and mean of the data is shown in figure 7.1. Based on this diagram the assumption is made that the IncreasedLatency follows a normal distribution. Based on this assumption it is possible to predict that 99.5% of the requests will have an increased latency below 1.8 seconds.

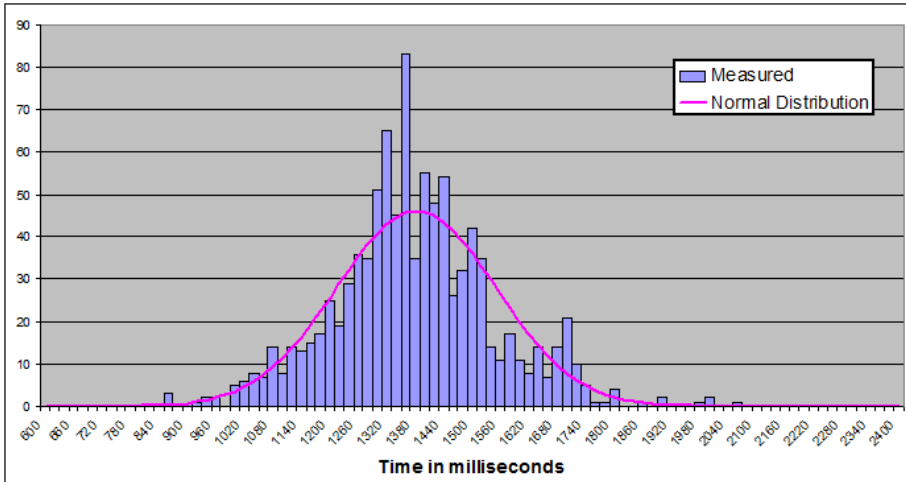


Figure 7.1: Histogram of measured IncreasedLatency and Normal Distribution plotted together. Non UML.

Because the Customer and 2250 is situated on the same network it will be assumed that $NTime2 = TunnelTime$. $NTime1$ and $NTime2$ can be determined as the connect, send and receive measurements combined for the respective direct and relay measurements. Based on this the $RSPTTime$ can be isolated, this value is static and identifying this allows to estimate increased latency as a function of how close the customer is to the $RelayService(NTime2)$, how close the 2250 is to the $RelayService(TunnelTime)$, and how close the Customer is to the 2250 ($NTime1$). A combined histogram and bell curve for the normal distribution using the measured mean and variance is show in figure 7.2. Based on this diagram the assumption is made that the $RSPTTime$ follows a normal distribution. Based on this assumption it is possible to predict that 99.5% of the requests will have an increased latency below 1.66 seconds.

Based on this number it is possible to conclude that the remaining 3.34¹ seconds are satisfactory to contain network latencies and requirement NFR3 can therefore be considered as fulfilled.

¹The 5 seconds in the requirement subtracted the 1.66 seconds

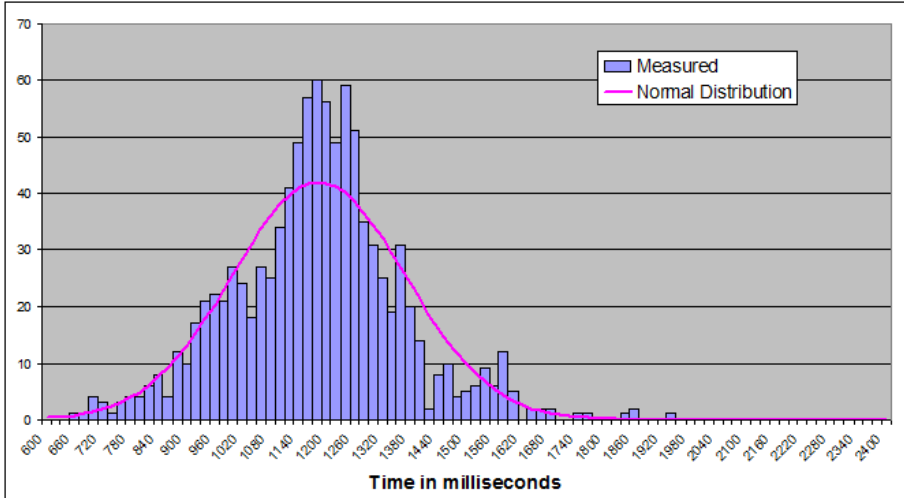


Figure 7.2: Histogram of measured RSPTime and Normal Distribution plotted together. Non UML.

7.2 Security

In [3.4.8 Web Service Security](#) security issues related to Web services were discussed. In the same subsection it was also discussed what security mechanisms could be used to mitigate these risks. Before determining the level of security in the proposed solution, the security in the existing set up with a direct connection will be discussed.

The direct connection has a direct connection between Customer and 2250Web-Server. The protocol used is HTTP and the data is unencrypted. To authenticate users the HTTP basic access authentication is used but the 2250 does not authenticate itself to the customer. In theory a malicious user could pose as a 2250 to other customers. This solution can only be considered secure when customer and 2250 are on the same local network where it is guaranteed that no eavesdroppers are present.

In the proposed solution all traffic is relayed over the RelayService. The StatusInterface uses the HTTP basic access authentication on the RelayService while requests to the RemoteAPI and Browser Interface are validated on the 2250WebServer. The RelayClient authenticates itself to the RelayService using the proprietary protocol previously defined. Currently the RelayService does not authenticate itself to the Customer or the RelayClient. Like in the

direct connection solution the traffic is unencrypted. Even though using the RelayService provides the same level of security as the direct connection it is still unacceptable. If this solution should ever become commercially viable SSL should be used to encrypt traffic and to provide authentication of the RelayService. Another way to increase security is to require authentication with the reverse proxy before access to the 2250WebServer is granted. There are several ways of doing this:

1. Introduce state to the use of the reverse proxy based on sessions or tokens. These strategies compromises respectively the REST architecture and requirement MH2 (page 17).
2. Use the Proxy-Authentication and Proxy-Authorization headers specified by the HTTP protocol. This violates the MH2 requirement.

When the additional security measures are implemented the communication to and the use of the Web services can be regarded safe and commercially viable. The security issues regarding cloud computing will be discussed in the following section.

7.3 A RelayService Without Cloud

In this section it will be discussed why and how the RelayService can be realized without utilizing a cloud computing platform.

The most prominent security risks related to cloud computing are Unauthorized User Access and Malicious Insider, Geographical Data Location, Data Segregation and Availability. Even though Availability is important to strive after the other three risks are the most relevant in this solution. Microsoft attempts to mitigate these threats, as described in [3.2.4 Risk Mitigation in Windows Azure](#) and for a majority of cloud users the risk mitigation will be sufficient. One of the major issues however is the Geographical Data Location where some customers may be unsatisfied with any data being stored in other countries or even simply remote locations.

One way to avoid these risks is to use an alternative to cloud computing and host the RelayService locally. This strategy will remove the benefits of cloud computing but is technically possible. Two elements in the RelayService are cloud specific. The worker role used to instantiate RelayServerControllers and the use of the Service Bus. The WorkerRole can be removed by creating a class with a main method which at runtime creates RelayServerControllers with specific

IP endpoints. The use of the Service Bus is limited to the RelayServiceBasic component. The interaction between the front end and back end happens via the use of the IFrontendQueueHandler and IBackendQueueHandler. The only class that use the Service Bus directly is the QueueHandler. The classes FrontendQueueHandler and BackendQueueHandler inherits from this class. The flexible solution provided by the abstract factory pattern enables creating new subclasses of IFrontendQueueHandler and IBackendQueueHandler as well as a new concrete IRelayServiceBasicFactory will allow a complete transition from the Service Bus messaging to another message strategy. This could be a direct connection between the front end and back end or a simple queue implementation if they coexist on the same server.

This shows how flexible the solution actually is and how easily the benefits of cloud computing can be traded for local hosting and increased security.

7.4 Consistency, Availability, Partition-Tolerance and Scalability

Throughout the design process scalability has been an important factor in the design. This scalability has been supported by the two major technologies RESTful Web services and Windows Azure. The two technologies are complementary and in all aspects independent. Scalability in RESTful Web services come from the Cache constraint combined with the Layered System constraint. This allows clients and intermediary nodes to use cache responses. Intermediary nodes can however only send responses if the request and responses are unencrypted. In the StatusInterface caching is supported by supplying the ETag header field. The scalability provided by Windows Azure comes from having multiple instances of a given role, this is supported by the Statelessness constraint in REST which makes it unimportant what instance handles a given request. This design choice is what allows the Windows Azure load balancer to function.

In relation to horizontal scaling presented in [3.1.2 Scalability](#) it is important which units there should be scaled, when they should be scaled, and how they are scaled. The identified scaling units are the WebRole, the WorkerRole, SQL Azure, and the Service Bus. The scalability of these units are described in table [7.4](#)

Due to the software design and because the front end is stateless the decision between C. A. or P. for the StatusInterface bounds in the choice of data storage. The choice fell upon the SQL Azure storage which can be classified as Consistent and Partition-Tolerant. This choice was made not because Consistency was weighted over Availability but rather because of the differences in functionality

Scale Unit	When and how
WebRole	When: When CPU load or memory usage exceeds maximum, or when number of threads exceeds maximum. How: Request additional instance in cloud manager or enable auto-scaling on the specified parameters as explained in 3.2.1 Execution Models .
WorkerRole	When: When CPU load or memory usage exceeds maximum, or when number of tcp connections are exhausted How: Request additional instance in cloud manager or enable auto-scaling on the specified parameters as explained in 3.2.1 Execution Models .
SQL Azure	When: Load on database is too high. How: Using SQL Azure federations where the database is spread out on multiple federations as explained in 3.2.3.3 Azure SQL Database .
Service Bus	When: Number of messages exceeds 2,000 messages per second. How: Create new Service Bus namespace and new Service Bus

Table 7.4: Scalability matrix

and pricing between the two different storage models.

The other aspect is what the reverse proxy can be categorised as. In chapter [4 Components and Communication](#) it was decided to maintain a constant connection between the RelayClient and the RelayService. When the RelayService was split into a back end and front end the connection was between the RelayClient and a single instance of the back end. This single connection and at any rate the single device classifies this as Available and Consistent. This is due to the very nature of the domain and is as such unchangeable.

If a database node fail occurs the StatusInterface can reach a situation where requests will be ignored to preserve Consistency. If the 2250 fails or is offline the requests will not reach the device and no response will be sent. The result is the same in both scenarios, no useful response is received. The remote proxy should not and can not be used for real time control of the 2250 and the StatusInterface need not be up at all times (if it did the choice of data storage would be different). Therefore the fact that a useful response is not received is acceptable for the solution and the choices made between C, A, and P, valid.

7.5 Evaluating Solution

Based on the performance tests it is evident that the round trip time will increase when using the RelayService as mediator, compared to a direct connection between the Customer and 2250. The increased time will especially be significant compared to a direct connection when the customer and 2250 are situated with a short network distance and the RelayService is situated with a longer network distance as evident in equation 7.4. The increased round trip time makes the solution unsuitable for real time controlling of the device. But for other applications where time delay is less significant the solution is still valid and it will meet the NFR3 requirement.

Another limitation is the acceptable size of HTTP request and responses. The solution uses the Service Bus to communicate between the front end and back end. The Service Bus has limited message sizes as described in [3.2.2 Messaging](#). The maximum message size is 256KB and therefore the solution cannot be used to transfer large files. However the interfaces that currently exist on the 2250WebServer do not define resources of such large size. The solution is suitable for the intended use, but any large data transfers require workarounds.

The proposed proof of concept prototype meets all the *must have* requirements specified in [2.2 Functional Requirements](#) as well as all the non-functional requirements from [2.3 Non-Functional Requirements](#) which have been tested. The solution provides access to 2550WebServers that are situated behind firewalls or do not have a public IP address. This allows customers to access their devices from anywhere. Furthermore the solution allows customers to get a quick overview of their devices and the status of these. As an added feature, the RelayClient can act as a forward proxy and allow customers to access other network devices on the local network. This only requires a minimum of added implementation. Based on the fulfilment of requirements and the implementation quality the proof of concept prototype has been a success. In [8.3 Future Work](#) it will be discussed how the solution can be evolved from the prototype state to a viable product.

Conclusion

This chapter contains the findings of and the overall conclusion for this project as well as the future work. The chapter will compare the proof of concept solution with the vision as well as the the problem statement and describe the objectives. Based on these findings the overall conclusion will be made. The findings and overall conclusion will be put into perspective by a section describing future work.

8.1 Findings

In chapter 1 [Introduction](#) the vision and problem statement were defined. The vision of this project is defined as the following:

The vision of this project is to provide a proof of concept prototype where the 2250 is remote controlled without it having a public IP address. Creating devices that can be initialized and remote controlled from anywhere in the world potentially reduces the time spent on configuring NMTs by B&K staff and provides added value for customers by providing remote control and access of a 2250. This furthermore enables B&K to decouple the physical interface on the 2250 from the remote control interface enabling B&K to provide faster and better ways of remote controlling a device off-site.

In [1.5 The Problem](#) the two major problems were identified as:

1. Providing an overview and status of a registered 2250s for a customer
2. Providing access to 2250 potentially behind firewalls/routers and with inbound ports closed.

This thesis has described how both problems can be addressed by introducing a central service hosted on the Windows Azure platform. The service provides the StatusInterface as a RESTful Web service. This allows customers to get an overview of registered devices, and provides reverse proxy functionality effectively allowing a customer to access a device placed anywhere. In [1.5.2 Thesis Definition](#) a set of objectives were chosen to quantify the different aspects of the development and design of such a solution. (Objectives fulfilled: 1).

[2.1 State of the Art](#) detailed the functionality of the existing solution and the functionality of other state of the art solutions. The 10 *must have* 6 *nice to have* and 6 non-functional requirements were identified based on the described functionality as well as the user scenarios introduced in [1.5 The Problem](#). Seven use cases were designed which cover all *must have* requirements. (Objectives fulfilled: 2, 3).

Chapter [3 Technology Analysis](#) specified the advantages of cloud computing, defined scalability, introduced the CAP theorem and identified the *Unauthorized User Access and Malicious Insider*, *Geographical Data Location*, *Data Segregation*, and *Uptime* security risks. Windows Azure was chosen as PaaS provider for the RelayService in [3.1.4 Choice of Cloud Computing Category and Cloud Provider](#). Windows Azure provides a platform designed to facilitate scalability and the different design options as well as risk mitigation were discussed. Windows Azure does mitigate the majority of risks to a degree that is acceptable by B&K and their customers. Out of SOAP, RESTful and WCF Web services RESTful architecture was chosen for the new StatusInterface to keep a uniform architecture across platforms. RESTful Web services mitigate identified Web service security risks by utilizing the underlying Web standards. (Objectives fulfilled: 4, 5, 6, 7).

In chapter [4 Components and Communication](#) it was specified how addressing the device in the host field allowed the relative path to a resource to remain the same on the device and the reverse proxy. The Tunnel strategy was chosen as communication strategy between RelayClient and RelayServer and a proprietary protocol for the communication was detailed. (Objectives fulfilled: 8, 9).

In chapter 5 [RelayClient](#) the behaviour of the RelayClient prototype has been defined. The implementation, and design including the *Abstract Factory*, *Singleton* and *Observer* patterns. A implementation was chosen were the RelayClient could act as a forward proxy. (Objectives fulfilled: 10, 11).

In the chapter 6 [RelayService](#) it was chosen to separate the RelayService into a front end (WebRole) and a back end (WorkerRole) to increase scalability. The Service Bus topic/subscription was chosen as a message system for communication between the two. The StatusInterface has been developed honouring the REST constraints. SQL Azure storage was chosen as persistent memory for the StatusInterface due to the increased functionality and more suitable pricing model. (Objectives fulfilled: 9, 10, 11).

In the chapter 7 [Discussion](#) security, use of cloud computing, combination of cloud computing and REST, CAP theorem related to the prototype, and evaluation of design were discussed. It was determined that the security offered by the prototype was insufficient and it was specified how it could be improved. It was also determined how the RelayService could be hosted in-house to mitigate cloud computing security risks. Furthermore it was established that REST architecture facilitates the horizontal scaling in Windows Azure. In relation to the CAP theorem it was established that the reverse proxy focusses on CA while the StatusInterface focusses on CP. The design evaluation showed strengths as well as shortcomings of the prototype, one of the most evident shortcomings is the increased latency making it unfit for time critical remote controlling. The two most evident pros are that it allows access to device regardless of firewalls and inbound port settings, and that it facilitates a new user interface independent of the physical appearance of the device. Based on acceptance tests it is concluded that all *must have* requirements have been met. Some *nice to have* and non-functional requirements were not tested as they required long term testing or multiple 2250s or customers (in the thousands). (Objectives fulfilled: 12, 13).

8.2 Overall Conclusion

Based on the findings in this thesis it can be concluded that it is possible to relay traffic via a central service and facilitate communication when both parties have private IP addresses or are behind firewalls. It can also be concluded that it is possible to maintain a status for each device on the central service so that customers can get a quick overview. The solution does need further work to be commercially viable which will be discussed in [8.3 Future Work](#), the aspect regarding a new remote control interface will also be discussed briefly. As a whole it can be concluded that the proof of concept prototype and the project as a whole is a success.

8.3 Future Work

The future work describes the remaining development necessary before the solution can be regarded as commercially viable.

As discussed in [7.2 Security](#) the security in the prototype is at an unsatisfactory level. Implementing encryption and two way authentication will be one of the steps in convincing users of the security of the solution and becoming commercially viable.

The RelayClient is currently a proof of concept prototype. This client has strict requirements in regards to memory and CPU use as requirement NFR5 expresses. While the current RelayClient is fully functional it is a requirement that the code should be revised and optimized even further. In combination with this the RelayClient should be integrated better in the BasicEnv to ensure on the fly shut down and start up of the RelayClient when software and firmware upgrades are in progress.

In [7.5 Evaluating Solution](#) it was established that the increased latency made the solution unfit for time critical remote controlling. One way to reduce load on the RelayService, provide a faster round trip time and to enhance the user interface would be to create a new REST client for accessing the RemoteAPI and the StatusInterface. This REST client should provide a GUI that is separated from the GUI on the 2250 and be designed for remote controlling via a RESTful Web service and take into account the strengths and weaknesses such a design has. This could also provide value for the customers as mentioned in [1.4 Vision](#).

To validate the solution large scale and long term tests should be designed to stress test the solution as well as verify that it behaves correctly over a longer period of time. This would validate NH3, NH3 and NFR4 which could not be established in [7.1.1 Functionality Acceptance Test](#) and [7.1.2 Performance Acceptance Test](#). Keep in mind that the RelayClient will be running for long time periods and the 2250 may be placed far from the office location. A crash of the RelayClient would require physical presence to restart.

After the above issues have been addressed it is time to consider how this solution fits in the B&K roadmap. Designing the extend of the use and the further focus areas is not easy in an environment with multiple stakeholders and conflicting interests. The software design itself is beautifully isolated from the rest of the software and admirably simple. From a technical point of view there is nothing to hinder for the application being integrated into other B&K products like the Lan-XI modules.

APPENDIX A

Use Cases

A.0.1 Use Case: Authenticate 2250

Authenticate 2250	
Actor	2250
Preconditions	None
Postconditions	None
Success End Condition	2250 has received positive authentication response
Failed End Condition	2250 has received negative or no authentication response
Main path (M)	<ol style="list-style-type: none"> 1. 2250 sends authentication request to relay service 2. RelayService receives authentication request 3. RelayService processes authentication request 4. RelayService sends positive authentication response 5. 2250 receives authentication response
Extensions	<ol style="list-style-type: none"> 2a RelayService does not receive authentication request 2a1 2250 waits a given interval and restarts at step 1 4a RelayService sends negative response 5a 2250 does not receive authentication response 5a1 2250 waits a given interval and restarts at step 1

A.0.2 Use Case: Authenticate Customer

Authenticate Customer	
Actor	Customer
Preconditions	None
Postconditions	None
Success End Condition	Customer has received positive authentication response
Failed End Condition	Customer has received negative or no authentication response
Main path (M)	<ol style="list-style-type: none"> 1. Customer sends authentication request to system 2. RelayService receives authentication request 3. RelayService processes authentication request 4. RelayService sends positive authentication response 5. Customer receives authentication response
Extensions	<ol style="list-style-type: none"> 2a RelayService does not receive authentication request 4a RelayService send negative response 5a Customer does not receive authentication response

A.0.3 Use Case: Register 2250

Register 2250	
Actor	Customer
Precondition	Customer is authenticated
Postcondition	
Success End Condition	2250 is registered to Customer
Failed End Condition	2250 is not registered to Customer
Main path (M)	<ol style="list-style-type: none"> 1. Customer sends register request to RelayService 2. RelayService receives register request 3. RelayService verifies the content of the register request 4. RelayService registers 2250 to customer 5. RelayService sends positive register response to customer 6. Customer receives register response
Extensions	<ol style="list-style-type: none"> 2a RelayService does not receive register request 3a RelayService cannot verify the content of the register request <ol style="list-style-type: none"> 3a1 RelayService sends negative register response to customer 6a Customer does not receive authentication response

A.0.4 Use Case: See 2250 Status

See 2250 status	
Actor	Customer
Precondition	Customer is authenticated.
Postcondition	
Success End Condition	Customer has received 2250 status
Failed End Condition	Customer has not received 2250 status
Main path (M)	<ol style="list-style-type: none">1. Customer requests to see status of a 22502. RelayService receives request3. RelayService validates that customer owns the 22504. RelayService responds with 2250 status5. Customer receives response

A.0.5 Use Case: Unstable Network

APPENDIX B

Screenshots of StatusInterface text/html Representation

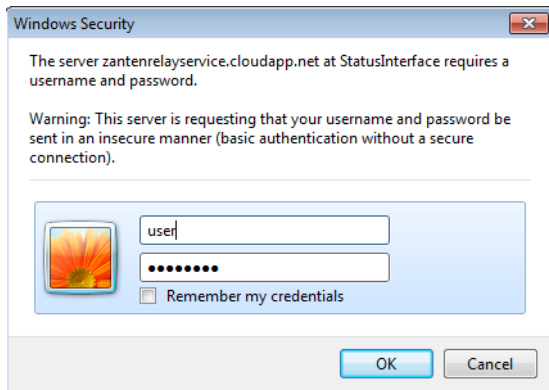


Figure B.1: Screen shot of the StatusInterface login. Non UML.

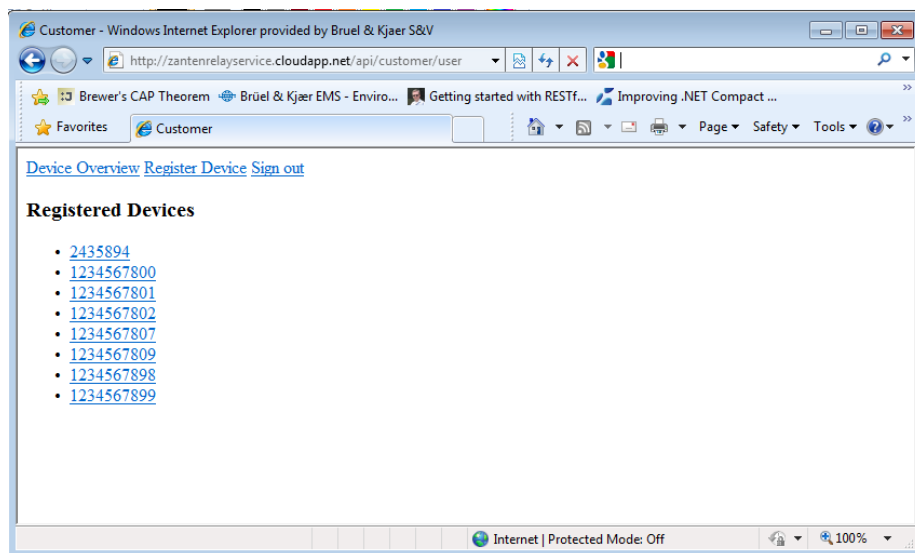


Figure B.2: Screen shot of the StatusInterface showing an overview of the customer and his registered devices. Non UML.

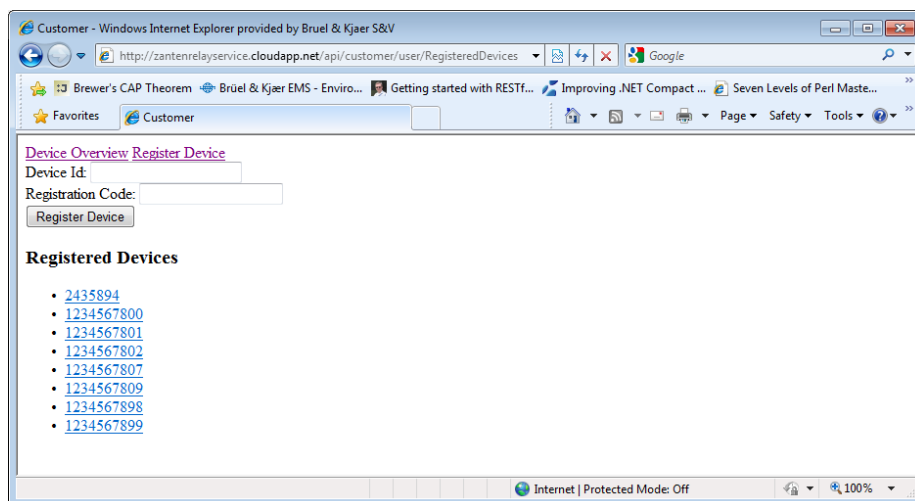


Figure B.3: Screen shot of the StatusInterface showing the registered devices for a customer and allowing the customer to register a new device. Non UML.

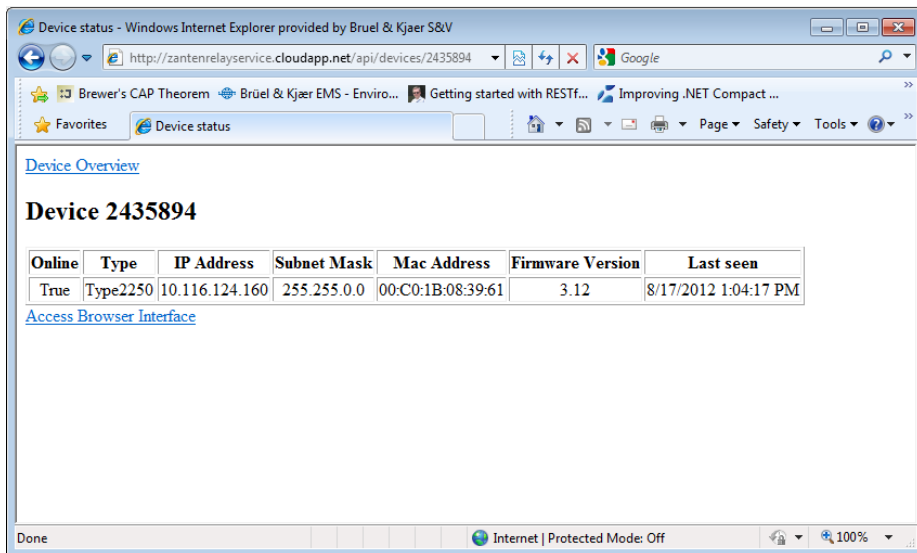
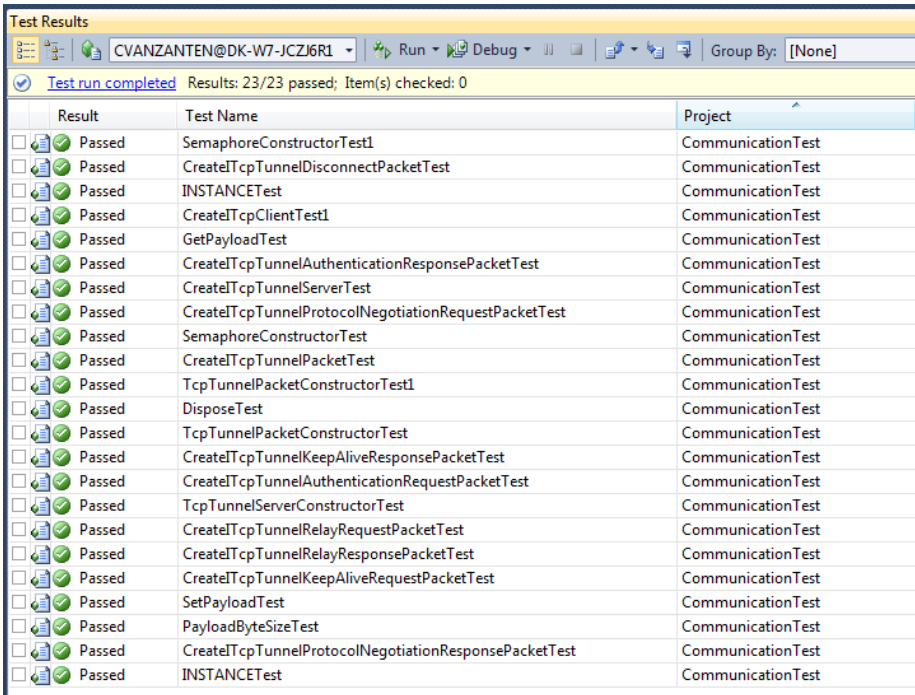


Figure B.4: Screen shot of the StatusInterface showing the status of a single device. Non UML.

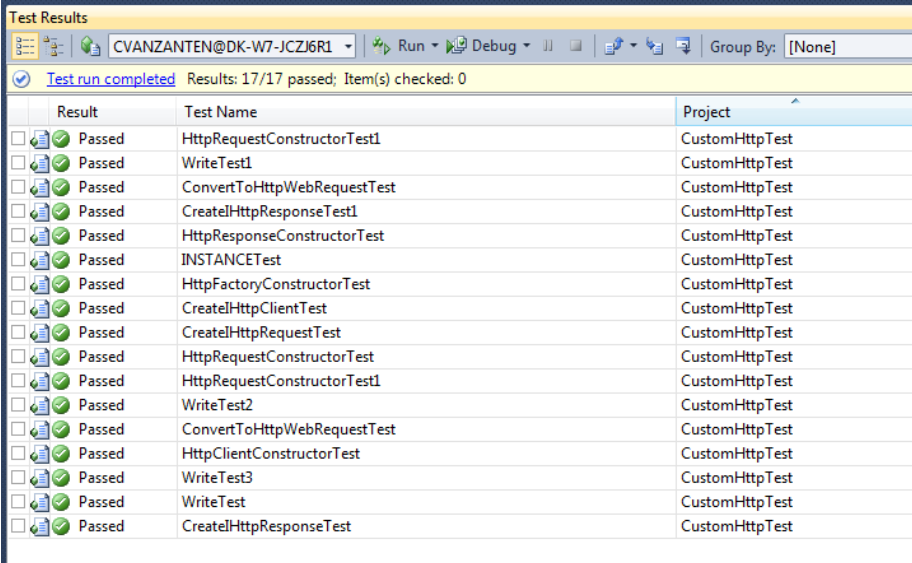
APPENDIX C

Unit and Integration Tests



Result	Test Name	Project
Passed	SemaphoreConstructorTest1	CommunicationTest
Passed	CreateITcpTunnelDisconnectPacketTest	CommunicationTest
Passed	INSTANCETest	CommunicationTest
Passed	CreateITcpClientTest1	CommunicationTest
Passed	GetPayloadTest	CommunicationTest
Passed	CreateITcpTunnelAuthenticationResponsePacketTest	CommunicationTest
Passed	CreateITcpTunnelServerTest	CommunicationTest
Passed	CreateITcpTunnelProtocolNegotiationRequestPacketTest	CommunicationTest
Passed	SemaphoreConstructorTest	CommunicationTest
Passed	CreateITcpTunnelPacketTest	CommunicationTest
Passed	TcpTunnelPacketConstructorTest1	CommunicationTest
Passed	DisposeTest	CommunicationTest
Passed	TcpTunnelPacketConstructorTest	CommunicationTest
Passed	CreateITcpTunnelKeepAliveResponsePacketTest	CommunicationTest
Passed	CreateITcpTunnelAuthenticationRequestPacketTest	CommunicationTest
Passed	TcpTunnelServerConstructorTest	CommunicationTest
Passed	CreateITcpTunnelRelayRequestPacketTest	CommunicationTest
Passed	CreateITcpTunnelRelayResponsePacketTest	CommunicationTest
Passed	CreateITcpTunnelKeepAliveRequestPacketTest	CommunicationTest
Passed	SetPayloadTest	CommunicationTest
Passed	PayloadByteSizeTest	CommunicationTest
Passed	CreateITcpTunnelProtocolNegotiationResponsePacketTest	CommunicationTest
Passed	INSTANCETest	CommunicationTest

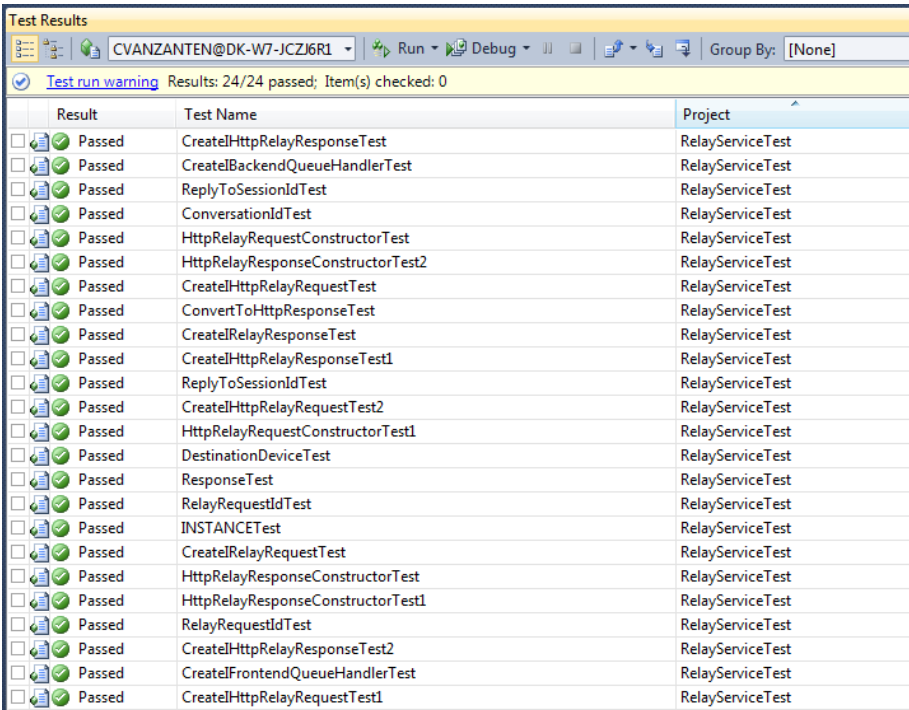
Figure C.1: Screen shot of the Communication tests. Non UML.



The screenshot shows a 'Test Results' window with a toolbar at the top containing icons for search, refresh, and execution (Run, Debug). The toolbar also includes a 'Group By' dropdown menu set to '[None]'. Below the toolbar, a status bar indicates 'Test run completed' with 'Results: 17/17 passed; Item(s) checked: 0'. The main area is a table with three columns: 'Result', 'Test Name', and 'Project'. Each row represents a test case, all of which are marked as 'Passed' with a green checkmark icon. The 'Project' column for all tests is 'CustomHttpTest'.

	Result	Test Name	Project
<input type="checkbox"/>	Passed	HttpRequestConstructorTest1	CustomHttpTest
<input type="checkbox"/>	Passed	WriteTest1	CustomHttpTest
<input type="checkbox"/>	Passed	ConvertToHttpWebRequestTest	CustomHttpTest
<input type="checkbox"/>	Passed	CreateHttpResponseTest1	CustomHttpTest
<input type="checkbox"/>	Passed	HttpResponseConstructorTest	CustomHttpTest
<input type="checkbox"/>	Passed	INSTANCETest	CustomHttpTest
<input type="checkbox"/>	Passed	HttpFactoryConstructorTest	CustomHttpTest
<input type="checkbox"/>	Passed	CreateHttpClientTest	CustomHttpTest
<input type="checkbox"/>	Passed	CreateHttpRequestTest	CustomHttpTest
<input type="checkbox"/>	Passed	HttpRequestConstructorTest	CustomHttpTest
<input type="checkbox"/>	Passed	HttpRequestConstructorTest1	CustomHttpTest
<input type="checkbox"/>	Passed	WriteTest2	CustomHttpTest
<input type="checkbox"/>	Passed	ConvertToHttpWebRequestTest	CustomHttpTest
<input type="checkbox"/>	Passed	HttpClientConstructorTest	CustomHttpTest
<input type="checkbox"/>	Passed	WriteTest3	CustomHttpTest
<input type="checkbox"/>	Passed	WriteTest	CustomHttpTest
<input type="checkbox"/>	Passed	CreateHttpResponseTest	CustomHttpTest

Figure C.2: Screen shot of the CustomHttp tests. Non UML.



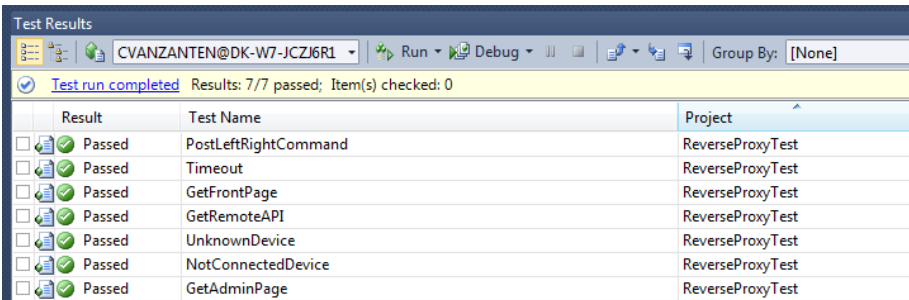
Test Results

CVANZANTEN@DK-W7-JCZJ6R1 Run Debug Group By: [None]

Test run warning Results: 24/24 passed; Item(s) checked: 0

Result	Test Name	Project
Passed	CreateHttpRelayResponseTest	RelayServiceTest
Passed	CreateBackendQueueHandlerTest	RelayServiceTest
Passed	ReplyToSessionIdTest	RelayServiceTest
Passed	ConversationIdTest	RelayServiceTest
Passed	HttpRelayRequestConstructorTest	RelayServiceTest
Passed	HttpRelayResponseConstructorTest2	RelayServiceTest
Passed	CreateHttpRelayRequestTest	RelayServiceTest
Passed	ConvertToHttpResponseTest	RelayServiceTest
Passed	CreateRelayResponseTest	RelayServiceTest
Passed	CreateHttpRelayResponseTest1	RelayServiceTest
Passed	ReplyToSessionIdTest	RelayServiceTest
Passed	CreateHttpRelayRequestTest2	RelayServiceTest
Passed	HttpRelayRequestConstructorTest1	RelayServiceTest
Passed	DestinationDeviceTest	RelayServiceTest
Passed	ResponseTest	RelayServiceTest
Passed	RelayRequestIdTest	RelayServiceTest
Passed	INSTANCETest	RelayServiceTest
Passed	CreateRelayRequestTest	RelayServiceTest
Passed	HttpRelayResponseConstructorTest	RelayServiceTest
Passed	HttpRelayResponseConstructorTest1	RelayServiceTest
Passed	RelayRequestIdTest	RelayServiceTest
Passed	CreateHttpRelayResponseTest2	RelayServiceTest
Passed	CreateFrontendQueueHandlerTest	RelayServiceTest
Passed	CreateHttpRelayRequestTest1	RelayServiceTest

Figure C.3: Screen shot of the RelayServiceBasic tests. Non UML.



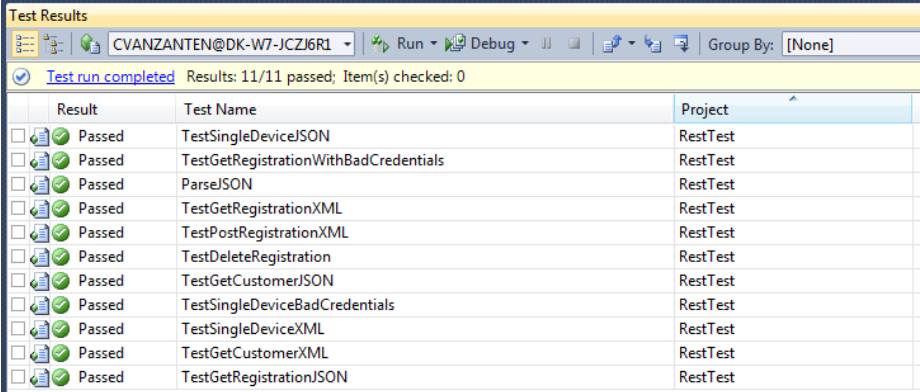
Test Results

CVANZANTEN@DK-W7-JCZJ6R1 Run Debug Group By: [None]

Test run completed Results: 7/7 passed; Item(s) checked: 0

Result	Test Name	Project
Passed	PostLeftRightCommand	ReverseProxyTest
Passed	Timeout	ReverseProxyTest
Passed	GetFrontPage	ReverseProxyTest
Passed	GetRemoteAPI	ReverseProxyTest
Passed	UnknownDevice	ReverseProxyTest
Passed	NotConnectedDevice	ReverseProxyTest
Passed	GetAdminPage	ReverseProxyTest

Figure C.4: Screen shot of the reverse proxy tests. Non UML.



The screenshot shows a 'Test Results' window with a toolbar at the top containing icons for search, refresh, run, debug, and other actions. Below the toolbar, a status bar indicates 'Test run completed' and 'Results: 11/11 passed; Item(s) checked: 0'. The main area is a table with columns for 'Result', 'Test Name', and 'Project'. All 11 tests listed are marked as 'Passed' and belong to the 'RestTest' project.

	Result	Test Name	Project
<input type="checkbox"/>	Passed	TestSingleDeviceJSON	RestTest
<input type="checkbox"/>	Passed	TestGetRegistrationWithBadCredentials	RestTest
<input type="checkbox"/>	Passed	ParseJSON	RestTest
<input type="checkbox"/>	Passed	TestGetRegistrationXML	RestTest
<input type="checkbox"/>	Passed	TestPostRegistrationXML	RestTest
<input type="checkbox"/>	Passed	TestDeleteRegistration	RestTest
<input type="checkbox"/>	Passed	TestGetCustomerJSON	RestTest
<input type="checkbox"/>	Passed	TestSingleDeviceBadCredentials	RestTest
<input type="checkbox"/>	Passed	TestSingleDeviceXML	RestTest
<input type="checkbox"/>	Passed	TestGetCustomerXML	RestTest
<input type="checkbox"/>	Passed	TestGetRegistrationJSON	RestTest

Figure C.5: Screen shot of the StatusInterface tests. Non UML.

APPENDIX D

Performance Test Data

Below is a excerpt of the measured data. The remaining data is available digitally.

No.	DNS lookup	Blocked	Connecting	Sending	Waiting	Receiving
1	0	0	0	0	444	10
2	0	0	0	0	440	20
3	0	0	0	0	440	20
4	0	0	0	0	440	20
5	0	0	0	0	440	20
6	0	0	0	0	440	20
7	0	0	0	0	450	10
8	0	0	10	0	440	10
9	0	0	0	0	440	20
10	0	0	10	0	430	20
11	0	0	0	0	440	20
12	0	0	0	0	440	20
13	0	0	0	0	440	20
14	0	0	0	0	440	20
15	0	0	0	0	440	20
16	0	0	0	0	440	20
17	0	0	0	0	440	20
18	0	0	10	0	440	10
19	0	0	0	0	450	10
20	0	0	0	0	450	10
21	0	0	0	0	440	20
22	0	0	0	0	440	20
23	0	0	0	0	440	20
24	0	0	0	0	440	20
25	0	0	0	0	450	10
26	0	0	10	0	440	10
27	0	0	0	0	440	20
28	0	0	0	0	440	20
29	0	0	0	0	450	10
30	0	0	10	0	440	10
...

Table D.1: Data from direct connection in sorted order.

No.	DNS lookup	Blocked	Connecting	Sending	Waiting	Receiving
1	0	0	40	0	1210	50
2	0	0	40	0	1220	50
3	0	0	40	0	1240	40
4	0	0	40	0	1290	40
5	0	0	40	0	1300	50
6	0	0	40	0	1310	50
7	10	0	40	0	1320	40
8	0	0	40	0	1330	40
9	10	0	40	0	1350	50
10	0	0	50	0	1360	40
11	0	10	40	0	1360	40
12	0	0	50	0	1370	40
13	0	0	40	0	1370	50
14	10	0	40	0	1380	40
15	10	0	40	0	1380	40
16	0	0	40	0	1390	40
17	0	0	40	0	1390	50
18	0	0	110	0	1320	50
19	0	0	40	0	1400	40
20	0	0	50	0	1385	50
21	0	0	40	0	1400	50
22	0	0	40	0	1410	40
23	0	0	40	0	1410	50
24	0	0	50	0	1410	40
25	0	0	40	0	1410	50
26	0	0	40	0	1420	40
27	0	0	40	0	1410	50
28	0	0	40	0	1420	50
29	0	0	40	0	1440	40
30	0	0	50	0	1420	50
...

Table D.2: Data from RelayService connection in sorted order.

Bibliography

- [1] Michael Armbrust et al. “A view of cloud computing”. In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: [10.1145/1721654.1721672](https://doi.org/10.1145/1721654.1721672). URL: <http://doi.acm.org/10.1145/1721654.1721672>.
- [2] Walter Wayne Berry et al. *Inside SQL Azure*. URL: <http://social.technet.microsoft.com/wiki/contents/articles/1695.inside-sql-azure.aspx>.
- [3] André B. Bondi. “Characteristics of scalability and their impact on performance”. In: *Proceedings of the 2nd international workshop on Software and performance*. Association for Computing Machinery, 2000, pp. 195–203. ISBN: 158113195X.
- [4] Engin Bozdag, Ali Mesbah, and Arie van Deursen. “A comparison of push and pull techniques for AJAX”. English. In: *WSE 2007: NINTH IEEE INTERNATIONAL SYMPOSIUM ON WEB SITE EVOLUTION, PROCEEDINGS*. Ed. by Huang, S and DiPenta, M. 9th IEEE International Symposium on Web Site Evolution, Paris, FRANCE, OCT 05-06, 2007. IEEE Comp Soc; PCOST; FAU; Queens Univ. 10662 LOS VAQUEROS CIRCLE, PO BOX 3014, LOS ALAMITOS, CA 90720-1264 USA: IEEE COMPUTER SOC, 2007, 15–22. ISBN: 978-1-4244-1450-5.
- [5] Brüel and Kjær. *Hand-held Analyzer Types 2250 and 2270: Product Data*. User Manual, Brüel and Kjær. Version BP 2025 – 18. 2012.
- [6] Brüel and Kjær. *Noise Monitoring Terminal Types 3639-A, 3639-B and 3639-C with Hand-held Analyzer Type 2250*. User Manual, Brüel and Kjær. Version English BE 1818 – 15. 2012.
- [7] Eric Brewer. “CAP Twelve Years Later: How the “Rules” Have Changed”. English. In: *COMPUTER* 45.2 (2012), 23–29. ISSN: 0018-9162.

- [8] Eric Brewer and Armando Fox. “Harvest, yield, and scalable tolerant systems”. In: *Proceedings of the 7th Workshop on 15 Hot Topics in Operating Systems*. Iee, 1999, pp. 174–178. ISBN: 0769502377.
- [9] Eric A. Brewer. *Towards Robust Distributed Systems*. Keynote at ACM Symposium on the Principles of Distributed Computing. Slides available online (12 pages). 2000. URL: <http://www.cs.berkeley.edu/~brewer/cs262b-2004/P0DC-keynote.pdf>.
- [10] Julian Brown. *Brewer’s CAP Theorem, The kool aid Amazon and Ebay have been drinking*. URL: <http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>.
- [11] Bruel and Kjær. *Brüel & Kjær Company Website*. <http://www.bksv.com/AboutUs/AboutBruelAndKjaer.aspx>.
- [12] Andrew J. Brust. *NoSQL and the Windows Azure platform Investigation of an Unlikely Combination*. White paper, Microsoft Corporation. Available online (28 pages). 2011. URL: <http://download.microsoft.com/download/9/E/9/9E9F240D-0EB6-472E-B4DE-6D9FCBB505DD/WindowsAzureNoSQL%20WhitePaper.pdf>.
- [13] Brad Calder et al. “Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency”. English. In: *SOSP 11: PROCEEDINGS OF THE TWENTY-THIRD ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*. 23rd ACM Symposium on Operating Systems Principles (SOSP 2011), Cascais, PORTUGAL, OCT 23-26, 2011. ACM SIGOPS; INESC ID. 1515 BROADWAY, NEW YORK, NY 10036-9998 USA; ASSOC COMPUTING MACHINERY, 2011, 143–157. ISBN: 978-1-4503-0977-6.
- [14] Roy T. Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. PhD thesis.
- [15] Roy. T Fielding. *Re: [rest-discuss] REST-**. URL: <http://tech.groups.yahoo.com/group/rest-discuss/message/13266>.
- [16] Roy T. Fielding and Richard N. Taylor. “Principled design of the modern Web architecture”. In: *ACM Trans. Internet Technol.* 2.2 (May 2002), pp. 115–150. ISSN: 1533-5399. DOI: 10.1145/514183.514185. URL: <http://doi.acm.org.globalproxy.cvt.dk/10.1145/514183.514185>.
- [17] Roy T. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: <http://www.w3.org/Protocols/rfc2616/rfc2616.html>.
- [18] Martin Fowler. *UML Distilled Third Edition: A Brief Guide to the Standard Object Modelling Language*. Pearson Education Limited, 2004. ISBN: 0321193687.

- [19] Armando Fox et al. “Cluster-based scalable network services”. In: *SIGOPS Oper. Syst. Rev.* 31.5 (Oct. 1997), pp. 78–91. ISSN: 0163-5980. DOI: [10.1145/269005.266662](https://doi.org/10.1145/269005.266662). URL: <http://doi.acm.org/10.1145/269005.266662>.
- [20] J. Franks et al. *HTTP Authentication: Basic and Digest Access Authentication, RFC 2617*. 1999. URL: <http://www.w3.org/Protocols/rfc2616/rfc2616.html>.
- [21] Svend Gade. *A Brüel and Kjær History Lesson Short Version*. URL: http://www.blznz.com/news/2008/10/07/Br\C3\%BCe1_Kj\C3\%A6r_History_Lesson_Short_4921.html.
- [22] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., 1995. ISBN: 0201633612.
- [23] Seth Gilbert and Nancy Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. In: (2002). URL: www.gartner.com/id=685308.
- [24] H. Haas, A. Brown, and Group. *Web Services Glossary*. 2004. URL: <http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211>.
- [25] Jay Heiser and Mark Nicolett. “Assessing the Security Risks of Cloud Computing”. In: (2008). DOI: [G00157782](https://doi.org/10.1145/157782). URL: www.gartner.com/id=685308.
- [26] LogMeIn Inc. *LogMeIn Security: An In-Depth Look*. White paper, LogMeIn. Available online (22 pages). 2012. URL: https://secure.logmein.com/welcome/documentation/EN/pdf/common/LogMeIn_SecurityWhitepaper.pdf.
- [27] Jason. *BizCloud Overview of Top 10 Security Threats of Cloud Computing*. URL: <http://bizcloudnetwork.com/bizcloud-overview-of-top-10-security-threats-of-cloud-computing>.
- [28] John Joyner. *Windows Azure Web, Worker, and VM roles demystified*. URL: <http://www.techrepublic.com/blog/networking/windows-azure-web-worker-and-vm-roles-demystified/4017>.
- [29] Nicholas Kolakowski. *Microsoft Windows Azure Downtime Blamed on Leap Year Bug*. URL: <http://www.eweek.com/c/a/Enterprise-Applications/Microsoft-Windows-Azure-Downtime-Blamed-on-Leap-Year-Bug-707169/>.
- [30] Josts Engineering Co Ltd. *Brüel & Kjær announces the acquisition of Australian company Lochard Ltd. with the intention of providing customers with world-class Environment Management Solutions*. URL: <http://www.indiaprwire.com/pressrelease/defense/2009062828362.htm>.
- [31] Microsoft. *Push Notifications Overview for Windows Phone*. URL: [http://msdn.microsoft.com/en-us/library/ff402558\(v=vs.92\).aspx](http://msdn.microsoft.com/en-us/library/ff402558(v=vs.92).aspx).

- [32] Microsoft. *Service Level Agreements*. URL: <https://www.windowsazure.com/da-dk/support/legal/sla/>.
- [33] Valery Mizonov and Seth Manheim. *Windows Azure Queues and Windows Azure Service Bus Queues - Compared and Contrasted*. URL: <http://msdn.microsoft.com/en-us/library/windowsazure/hh767287.aspx>.
- [34] MKNZ. *Azure Storage, BASE and ACID*. URL: <http://convective.wordpress.com/2009/07/13/azure-storage-base-and-acid/>.
- [35] Michael P. Papazoglou. *Web services: Principles and Technology*. Pearson Education Limited, 2007. ISBN: 9780321155559.
- [36] Murali Manohar Pareek. *WCF (Windows Communication Foundation) Introduction and Implementation*. 2008. URL: <http://www.codeproject.com/Articles/30374/WCF-Windows-Communication-Foundation-Introduction>.
- [37] Doug Purdy and Jeffrey Richter. *Exploring the Observer Design Pattern*. URL: <http://msdn.microsoft.com/en-us/library/ee817669.aspx>.
- [38] *Rest-**. URL: <http://www.jboss.org/reststar>.
- [39] Jeffrey Richter. *Understanding Cloud Storage*. URL: <http://www.windowsazure.com/en-us/develop/net/fundamentals/cloud-storage/>.
- [40] Aaron Skonnard. *A Developer's Guide to Service Bus in Windows Azure platform AppFabric*. White paper, Microsoft Corporation. Available online (50 pages). 2009. URL: <http://go.microsoft.com/fwlink/?LinkID=150834>.
- [41] Ishpreet Singh Virk and Raman Maini. "Cloud Computing: Windows Azure Platform". In: *Journal of Global Research in Computer Science* 3.1 (2012), pp. 74–76. ISSN: 2229-317x.
- [42] Erik Wilde and Cesare Pautasso, eds. *REST: From Research to Practice*. 1st Edition. Springer, Aug. 2011. ISBN: 9781441983022. URL: <http://amazon.com/o/ASIN/1441983023/>.