Cloud connectivity for embedded systems

ERIK ELDH

KTH Information and
Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

Cloud connectivity for embedded systems

Erik Eldh

Master of Science Thesis

Communication Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

25 February 2013

Examiner: Professor Gerald Q. Maguire Jr.

Abstract

Deploying an embedded system to act as a controller for electronics is not new. Today these kinds of systems are all around us and are used for a multitude of purposes. In contrast, cloud computing is a relatively new approach for computing as a whole. This thesis project explores these two technologies in order to create a bridge between these two wildly different platforms. Such a bridge should enable new ways of exposing features and doing maintenance on embedded devices. This could save companies not only time and money while dealing with maintenance tasks for embedded systems, but this should also avoid the needed to host this maintenance software on dedicated servers – rather these tasks could use cloud resources only when needed. This thesis explores such a bridge and presents techniques suitable for joining these two computing paradigms together.

Exploring what is included in cloud computing by examining available technologies for deployment is important to be able to get a picture of what the market has to offer. More importantly is how such a deployment can be done and what the benefits are. How technologies such as databases, load-balancers, and computing environments have been adapted to a cloud environment and what draw-backs and new features are available in this environment are of interest and how a solution can exploit these features in a real-world scenario. Three different cloud providers and their products have been presented in order to create an overview of the current offerings.

In order to realize a solution a way of communicating and exchanging data is presented and discussed. Again to realize the concept in a real-world scenario.

This thesis presents the concept of cloud connectivity for embedded systems. Following this the thesis describes a prototype of how such a solution could be realized and utilized. The thesis evaluates current cloud providers in terms of the requirements of the prototype.

A middle-ware solution drawing strengths from the services offered by cloud vendors for deployment at a vendor is proposed. This middle-ware acts in a stateless manner to provide communication and bridging of functionality

between two parties with different capabilities. This approach creates a flexible common ground for end-user clients and reduces the burden of having the embedded systems themselves process and distribute information to the clients. The solution also provides and abstraction of the embedded systems further securing the communication with the systems by it only being enabled for valid middle-ware services.

Sammanfattning

Att använda ett inbyggt system som en kontrollenhet för elektronik är inget nytt. Dessa typer av system finns idag överallt och används i vidt spridda användningsområden medans datormolnet är en ny approach för dator användning i sin helhet. Utforska och skapa en länk mellan dessa två mycket olika platformar för att facilitera nya tillvägagångs sätt att sköta underhåll sparar företag inte tid och pengar när det kommer till inbyggda system utan också när det gäller driften för servrar. Denna examensarbete utforskar denna typ av länk och presenterar för endamålet lämpliga tekniker att koppla dem samman medans lämpligheten för en sådan lösning diskuteras.

Att utforska det som inkluderas i konceptet molnet genom att undersöka tillgängliga teknologier för utveckling är viktigt för att få en bild av vad marknaden har att erbjuda. Mer viktigt är hur utveckling går till och vilka fördelarna är. Hur teknologoier som databaser, last distrubutörer och server miljöer har adapterats till molnmiljön och vilka nackdelar och fördelar som kommit ut av detta är av intresse och vidare hur en lösning kan använda sig av dessa fördelar i ett verkliget scenario. Tre olika moln leverantörer och deras produkter har presenterats för att ge en bild av vad som för tillfället erbjuds.

För att realisera en lösning har ett sett att kommunicera och utbyta data presenterats och diskuterats. Åter igen för att realisera konceptet i ett verkligt scenario.

Denna uppsats presenterar konceptet moln anslutbarhet för inbyggda system för att kunna få en lösning realiserad och använd.

En mellanprograms lösning som drar styrka ifrån de tjänster som erbjudas av molnleverantörer för driftsättning hos en leverantor föreslås. Denna mellanprogramslösnings agerar tillståndslöst för att erbjuda kommunikation och funktions sammankoppling mellan de två olika deltagarna som har olika förutsätningar. Denna approach skapar en flexibel gemensam plattform för olika klienter hos slutanvändaren och minskar bördan hos de inbyggdasystemet att behöva göra analyser och distrubuera informationen till klienterna. Denna lösning erbjuder också en abstraktion av de inbyggdasystemen för att erbjuda ytterligare säkerhet när kommunikation sker med de inbyggdasystemet genom

att den endast sker med giltiga mellanprogram.

Acknowledgements

This master's thesis was performed at Syntronic Software Innovations AB in Kista, Sweden.

I would like to thank my advisor at the company Eric Svensson, my thesis employer David Näslund, and my examiner Professor Gerald Q. Maguire Jr. for input and support during this thesis project.

Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

AES	Advanced Encryption Standard
ACID	Atomicity, Consistency, Isolation, and Durability
API	Application Programming Interface
AZ	Availability Zone
BLOB	Binary Large Object
DaaS	Data as a Service
CRC	Cyclic Redundancy Check
DNS	Domain Name System
DSA	Digital Signature Algorithm
HDR	High Replication Data-store
HMAC	Hash-based Message Authentication Code
HPC	High Performance Computing
HTTP	Hyper Text Transfer Protocol
IaaS	Infrastructure as a Service
IP	Internet Protocol
MIB	Management Information Base
NMC	Network Management Center
OData	Open Data

OS	Operating System
PaaS	Platform as a Service
PC	Personal Computer
REST	Representational State Transfer
RSA	Rivest Shamir Adleman
RTOS	Real-time operating system
SaaS	Software as a Service
SDK	Software Development Kit
SNMP	Simple Network Management Protocol
SRAM	Static Random Access Memory
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTL	Time-To-Live
VM	Virtual Machine
XML	Extensible Markup Language
WAS	Windows Azure Storage
WSDL	Web Services Description Language
WSGI	Web Server Gateway Interface
WSN	Wireless Sensor Networks

Chapter 1

Introduction

Cloud computing has over the last few years become a major platform for companies due to its ability to reduce costs and because this paradigm leads to a managed IT infrastructure that can be used to dynamically provision and dimension services. The cloud consists of both hardware and software provided by a data center for which a customer pays only for the resources that they use. Cloud computing exploits virtual machines (VMs) running on clusters of computers. These computers can either be on site or at a hosting provider. The latter solution enables the customer to tailor the number of the virtual machines on the fly as a function of load. This creates a flexible environment that can be used to address different scenarios and phases of an application's usage (deployment, maintenance, and support). The users of this flexibility ranges from high performance computing (HPC) (for example a customer can rent 100 or more VMs to do processing for a period of a few hours) to dynamically scaling the numbers of computers that are used to filter and process a company's e-mail [1, 2].)

In 2009, while researching cloud computing the consulting firm McKinsey found 20 different definitions of the concept. Thus what is perceived as cloud computing can differ between different providers and companies [3]. The United States of America's National Institute of Standards and Technology (NIST) describes the cloud computing as a model for an on-demand pool of networked computing resources which can be deployed rapidly and with minimal interaction [4].

Embedded systems have been deployed in various scenarios to act as controllers. Such systems are quite prevalent today. These systems have different designs, capabilities, and usage. Connecting these systems to the Internet has been done to a varying degrees, but in most cases these systems have only been connected to internal networks. Enabling these systems to securely function when used as Internet enabled devices requires consideration of the

embedded system's often limited capabilities [5]. However, the performance of embedded systems has increased since this earlier paper was published in 2004. Today, the extension of embedded systems to support secure re-programming has been examined by Mussie Tesfaye in his recent Master's thesis [6]. Today an increasing fraction of these embedded systems are being connected to the Internet and form an *Internet of things*. Modern appliances are designed and manufactured with the intent that the resulting appliance will be Internet enabled. Building this capability during development gives the designer an opportunity to address concerns that are difficult to address when adding Internet connectivity to already deployed embedded systems [7, 8].

1.1 Problem context

Cloud computing has become very prevalent and at the same time the number of Internet connected devices is rapidly increasing. These Internet connected devices are not only personal computers (PCs), but increasingly include the computers in cars, lamp posts, the bank card in your wallet, and so forth. Today all of these things are getting Internet connectivity in one way or another. Additionally, new and different products are being created every day. The task of managing all of these products when they are deployed in unison leads to a new scenario which could benefit from cloud computing. A flexible system that can be tailored for all sorts of different uses over time suggests a future where one might expect the lamp post outside your house to inform the maintenance service when it is not working properly – by running diagnostics, reporting statistics about the number of daylight hours/light levels/..., etc. More importantly managers can remotely update the firmware and applications running on an embedded system without requiring physical interaction [9]. In some ways we are moving back to the mainframe and terminal model of timeshared computing, but with the mainframe being a logical service deployed in the cloud and a thin client realized as an embedded system. This evolution also means that a networked embedded device can now have capabilities based upon carrying out operations in the cloud and not simply being restricted to its own local resources.

Syntronic Software Innovations AB has an embedded systems platform called Midrange [10]. The purpose of the thesis project is to explore the possibilities of using a cloud based solution to manage this platform. Limitations of what is possible will heavily depend on the hardware and network connectivity of the Midrange platform. Communication must be set up in a secure manner in an environment based on rapidly deployed servers and

platforms. Creating a cloud based manager can save the company costs as the cloud based solution eliminates the need for a local dedicated server. At the same time this solution can enable a company to manage its deployed products remotely. A goal of this thesis project is to create a generic solution in order to give the company a base to work from, while providing a flexible solution that is able to adapt to different deployment scenarios as requested by customers. Handling problems such as a server crash in the cloud can be recovered from swiftly by detecting a faulty VM and starting up a replacement in the cloud. This new VM can assume the responsibilities of the crashed VM. Additionally, this approach avoids the need for the company to support dedicated hardware or even legacy hardware, while greatly scaling up the company's ability to support very large numbers of deployed systems.

1.2 Structure of this thesis

This thesis is divided into a literature study chapters (Chapter 1 and 2) providing an introduction to the cloud computing paradigm and the context of the thesis. Work related to this thesis such as earlier implementations on the target platform concerning the topic of this thesis, are presented and how they fit in. A scenario for management and usage is presented. This scenario will be utilized in subsequent chapters.

Chapter 4 presents the topics that will be reviewed and the expected outcome.

Chapters 5 and 6 propose how communication should be done and a summary of cloud providers is given in order to highlight concerns when choosing a provider.

Chapter 7 presents an analysis of several cloud providers and their environments by evaluating an example implementation and its deployed environment.

Chapter 8 presents the conclusions drawn during this thesis and suggest some future work.

Chapter 2

Background

This chapter gives an introduction to cloud computing and its benefits. The chapter also introduces the embedded system which is the intended target of this thesis project. The functionality desired of a solution is also explored in order to give an overview of the project.

2.1 Cloud computing

As discussed in Chapter 1, cloud computing is based on using virtual machines (VMs) to deliver different sorts of functionality to a customer (who might be an individual, a company, a government agency, etc.). This chapter will review a number of different cloud offerings that are currently available and how these offerings are divided into sub categories. Examples of the services offered by these different providers will be described.

2.1.1 The cloud

According to Sun Microsystems (now part of Oracle) the deployment of clouds can be divided into three different scenarios [2]:

- A *private cloud* is, as the name implies, a cloud used by a single company. This cloud is either entirely hosted and operated by the company itself or can be located at a co-hosting facility. Reasons for considering and choosing this approach for a cloud are the control and security of the platform itself (as viewed by to the company or a hired operator).
- A *public cloud* differs from a private cloud, in that a public cloud is deployed on a shared infrastructure. The cloud thus shares resources and hardware with different customers. These resources may be located in the

same data center and may be operated by a third party. Here the security aspects of using a cloud becomes more apparent as multiple entities are sharing a common infrastructure. Securing the services provided in such a way as to limit the threat of information leakage should be considered a priority. Some of the ways this information can leak are described by Victor Delgado in his Master's thesis [11].

- A *hybrid cloud* is a mix of both a private and public cloud. A company can run its own cloud on its own hardware in-house, while utilizing as necessary resources such a computing power and storage resources in a public cloud operated by a third party.

The service models provided by a cloud vendor are presented in table 2.1 and are described as follows: [4]

- Infrastructure as a service (IaaS) is based upon providing VMs running guest operating systems, together with providing storage capacity, servers, and more specific solutions such as load balancers. Companies offering such solutions include Amazon's Web Services (AWS), Microsoft's Azure, and GoGrid.

- Platform as a service (PaaS) is offered to customers who wish to deploy their own applications to be run in the cloud. These applications run on VMs with varying levels of control, but the underlying structure remains control led by the hosting provider. Example of PaaS usage includes using a cloud based web server or a Java virtual machine [2]. PaaS should be compared to the longer term and less flexible alternative of renting a co-host location to house dedicated hardware to run applications. The list of companies providing PaaS partially overlaps those providing IaaS (such as AWS and Azure) together with offerings such as Google's App Engine and Salesforce Force.com [12].

- Software as a service (SaaS) offers specific software based solutions running in the cloud. Customers can choose from applications such as email and other collaboration tools to be used by thin clients and/or end users. A simple example is the Gmail email service offered by Google and used by users via their web browsers. Another example is Microsoft's Office 365 – a web based variant of its Office suite. These offerings utilize the servers in the cloud as a back end for the specific software application, while providing the user's only an interface, rather than each of the users using a full featured application running locally.

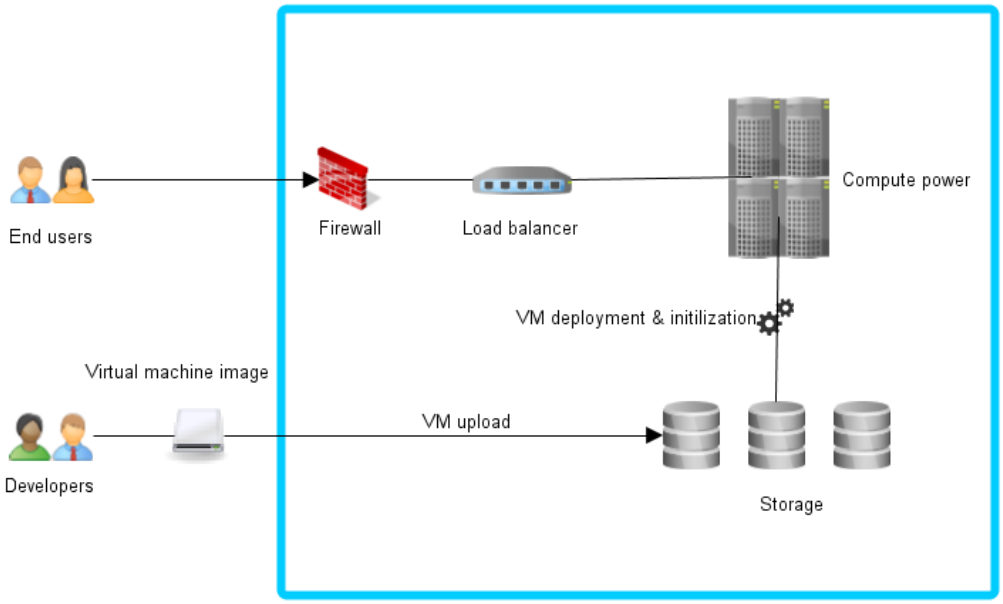

Figure 2.1: An example topology of a IaaS provider.

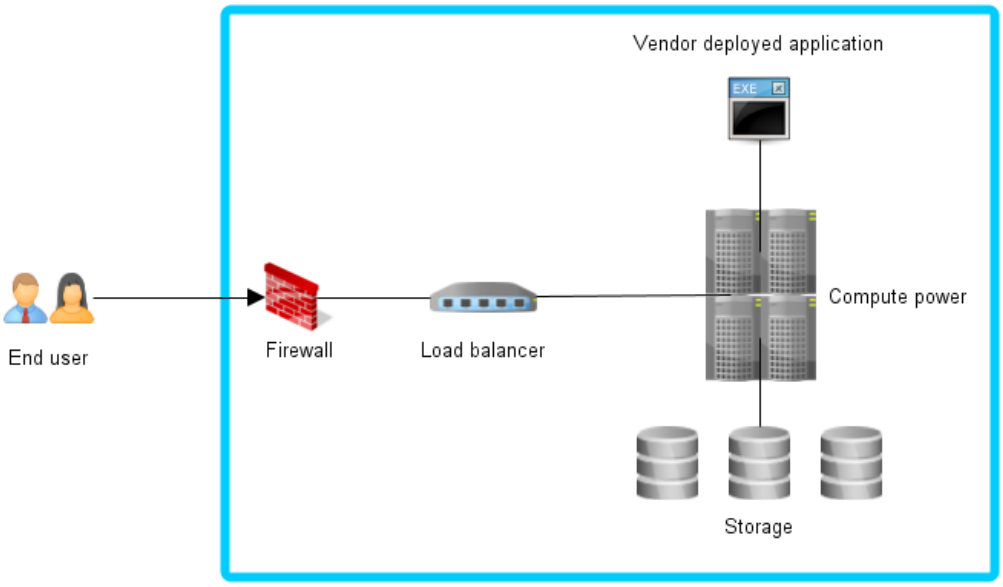

Figure 2.2: An example topology of a SaaS provider.

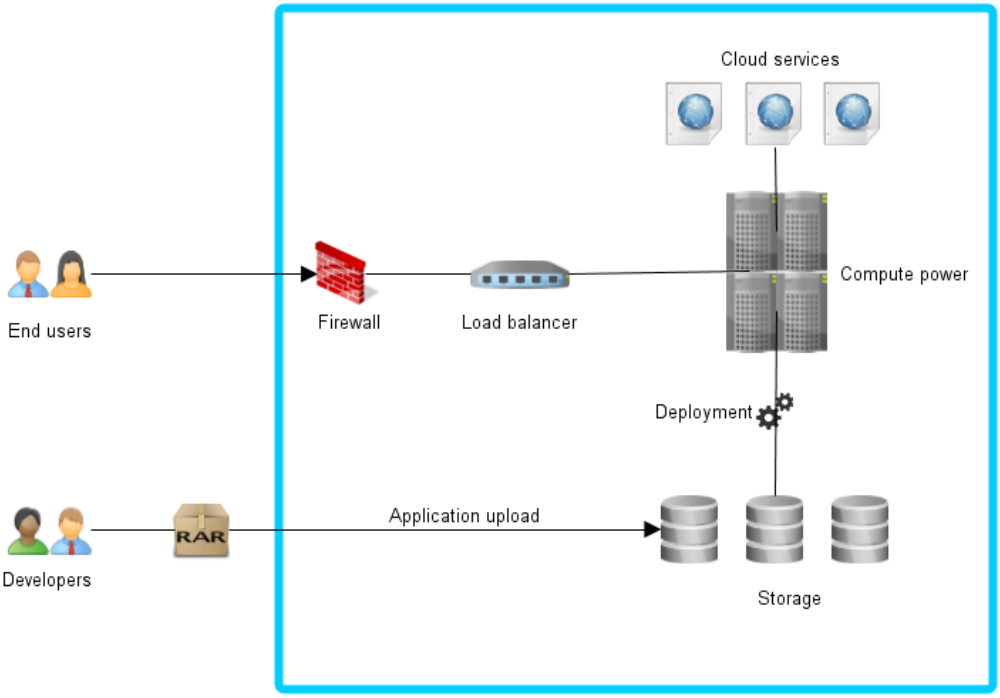

Figure 2.3: An example topology of a PaaS provider.

Table 2.1: Some of the services offered by cloud providers.

Service	**Providers & products**
SaaS	Salesforce, Office 360, Dropbox
PaaS	Windows Azure, Google App Engine, Amazone AWS
IaaS	Amazone AWS, Rackspace, Windows Azure

The relationship between these service models are shown in Figure 2.1; where each layer of service is stacked upon a virtulization layer running on servers located in data-centers. The topology and deployment scenarios for each service model are presented in Figures 2.2, 2.3, and 2.4.

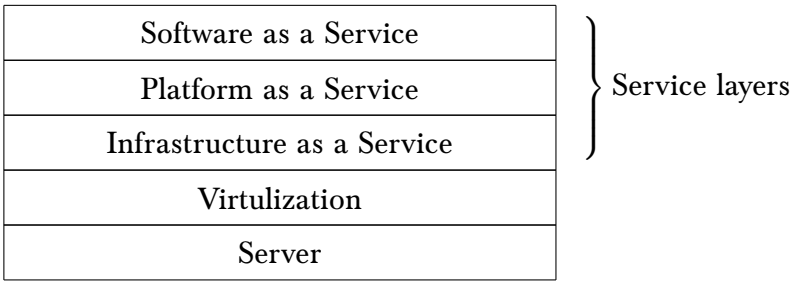

Figure 2.4: Cloud service stack.

From a technological and business viewpoint the acronym CLOUD has been proposed to summarize the benefits and possibilities of cloud computing. The acronym is dissected as follows: [13]

Common Infrastructure The infrastructure running the cloud.

Location-independence The location of the physical data-centers and distribution nodes are unknown to the user (i.e., there is not dependence on physical location).

Online connectivity The resources are accessible over a network.

Utility pricing The pricing of the resources is directly linked to their usage.

on-Demand Resources Resources are provided on demand, i.e. supplies the required resource when you need it and only for that the cloudperiod of time.

2.1.2 Service providers

Service providers of cloud based solutions provide a lot of different products and utilize many different techniques. In [12] Li, et al. have summarized these solutions highlighting the following areas concerning the deployment of a web service:

1. Elastic compute cluster,
2. Persistent storage,
3. Intra-cloud network, and
4. Wide-area delivery network.

The relationship between the cloud client and the VMs running in the cloud is presented in Figure 2.5. Each of the functionalities will be presented in the following sections.

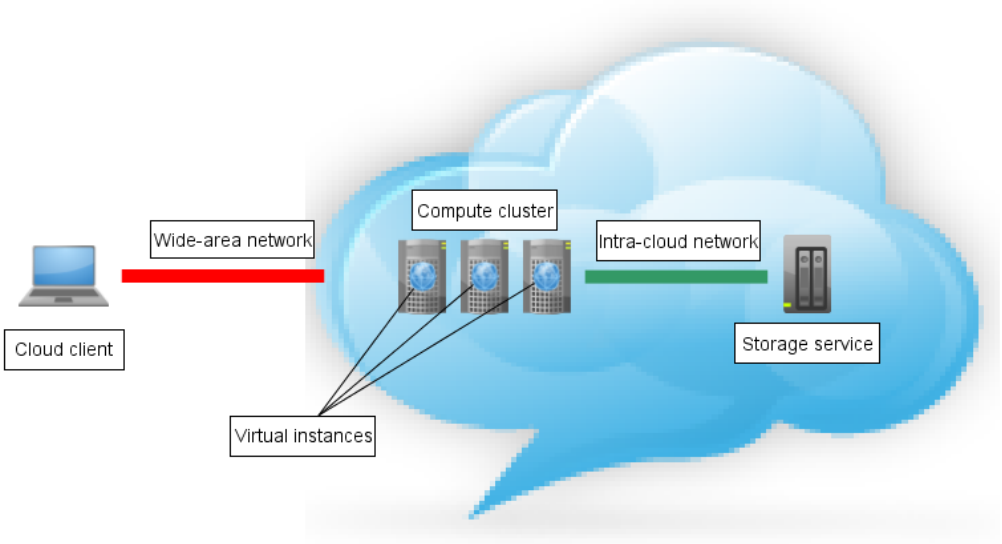

Figure 2.5: The relationship between clients and virtual machines to the services in a cloud infrastructure.

2.1.2.1 Elastic compute cluster

An elastic compute clusters consists of vendor provided VMs that are deployed to execute the customer's applications. This functionality provides the customer with the ability to have multiple instances of software working in parallel. This creates an environment in which exploiting the cloud's cost effectiveness and lower operating costs reduces the customer's overall costs for deploying a web based application. This is in contrast to traditional web hosting services where there may even be a many-to-one mapping of web sites to web server [14].

Different cloud providers use different techniques to provide this functionality. These techniques differ in three key areas: the underlying hardware (the hardware located in the data-centers), the virtulization layer (the chosen virtulization techniques used by the provider - such as Xen, VMWare, or other solutions), and the hosting environment (how the provider has configured and delivers the functionality to their customer) [12].

2.1.2.2 Persistent storage

The storage functionality offered by cloud providers adheres to the classical behavior of storing data in the VM [12]. This is done in the same way as traditional server and database solutions, with some added functions for the purpose of scaling and used in a cloud like environment [15]. Persistent storage solutions include database solutions such as Apache's Cassandra and NoSQL [16].

2.1.2.3 Intra-cloud network

Providing an internal network with high bandwidth is crucial for running of cloud based applications which require more than a single processor. When dealing with a single server, the services required for execution (e.g. a web application) are all available on the same system. For applications deployed in the cloud we (may) need to be able to communicate with services running in parallel on multiple systems [12, 14].

2.1.2.4 Wide-area delivery network

Most providers offer geo-distributed data centers in order to rapidly process requests all around the globe [12]. These so called edge-locations ensure service availability, but raise the problem of how to ensure data validity. If new data does not propagate to all servers, then we have to deal with data being provided to users that is no longer consistent or up to date. The great benefit of these edge location is the ease of dealing with large amounts of data and to distribute

this data to users all over the globe (as might be needed for a video streaming service). Using a single location to provide this functionality results in not only higher delay, but high traffic loads on communication links [15].

2.1.3 Local infrastructure

Not using a cloud provider's platform is also possible, while still exploiting the cloud computing paradigm. This means that the company may installing and utilize a cloud platform solution on-premises to deploy their own cloud platform. Some of the more well known solutions for doing this are OpenNebula and Eucalyptus [17]. Comparing these two cloud managers gives us some insight into the current state of cloud platforms. OpenNebula set out to create an industry standard. In an open source manner Eucalyptus implements a platform very similar to that provided by Amazon. These two distinct managers enable companies to make different decisions with regard to how to implement their own cloud. Choosing Eucalyptus results in an implementation compatible with a well established cloud provider (facilitating migrating from the company's own cloud to Amazon's cloud), while OpenNebula makes all of the deployment customizable [18].

2.1.4 Cloud orchestration

Cloud orchestration is the process of selecting and deploying different services in a cloud infrastructure at a provider and integrating their functionality [2]. If horizontal scaling of a service running on a cloud platform is desired, then the implemented service or chosen software should be designed and configured to be able to work in parallel with duplicate instances running concurrently. An example of horizontal scaling is adding an additional instance of a web server to a web server system in order to reduce the load on the existing elements of that system - so as to be able to handle higher load and/or provide better performance. In order to make this transparent for the user or software using the service, load balancers are used to distribute the work over the servers. Traffic is routed through the load balancer to the different instances. The load balancer must take into account what happens when the load on the service decreases and some of the instances can be shut down, hence it must route traffic only to the instances that will continue to be used. The usage of load balancers can be extended for use internally within the cloud platform, such as between web servers and application servers [14].

Software that handles the task of orchestrating a service to be deployed in the cloud is available in different varieties. Functionality is either provided by the cloud service provider itself or by a third party. This functionality is also

available thorough software such as Ubuntu's juju. Diaz, et al. have said that the deployment of services ranges from using ready-made packages to using custom configured ones or deploying custom operating systems [19].

2.1.5 Characteristics

The hardware offered by providers on which the cloud services and solutions are running is not the real difference offered by cloud computing, but rather the difference in the way that this hardware is used. The same statement can be made to some extent about the software, although software is now being explicitly designed to be deployed in a cloud based infrastructure [20]. As a result some of the problems encountered when considering a cloud based solution are not new, but are quite familiar problems. For example, when deploying a web application on a traditional server, a developer would take into account the same types of threats as when deploying on a cloud platform.

However, deploying service in the cloud results in some new security aspects that do need to be taken into consideration. One of the most obvious aspects is that the applications deployed in the cloud run on shared resources at the provider, hence we introduce the risk of a covert-channel which a malicious user can use to attack applications running on the shared resources [11]. Another problem occurring in the shared infrastructure originates from the fact that the applications are *involuntary* linked. For example, if one client's application causes disturbance to the shared resources resulting in downtime or access issues, then these problems could spill over to the other users of the shared resources [15]. These instance of outright failures should be dealt with by the cloud management system.

Addressing these aspects of deploying and running a web application in the cloud must be done both at an implementation level and by trusting the provider. Trusting the provider might sound abstract, but is a reality that must be accepted when deploying in the cloud – since only limited access to the underlying infrastructure of the cloud is given to the customers, if the customers get any access at all. For example, consider data availability: if database access is hindered for some reason this could render a deployed web application useless, thus the customer must be confident that the provider can fix the problem with the database access so that the web application can operate correctly. Additionally, the customer must consider what happens if they wish to move their data to another provider (see for example Vytautas Zapolskas' recent Mater's thesis [21]). These are some of the kinds of issues that need to be addressed when considering moving an application to the cloud

2.2 Embedded systems

Embedded systems differ a lot between platforms, thus it is hard to describe a general solution. Different embedded systems also offer different advantages and disadvantages with respect to Internet enabled usage [8]. For example, cryptographic processors can be used to shift the burden of doing the calculations needed for encryption from the main processor to the cryptographic processor. Similarly read only memory can be used to prevent tampering with stored keys and addresses used for communication. Hardware support of these sorts can make some systems more suitable than other systems with respect to communications with over the Internet.

This thesis will focus on the Syntronic Midrange platform as a general platform for various tasks. This section gives an introduction to this platform and the operating system that it runs.

2.2.1 The Midrange platform

The Midrange platform was developed by Syntronic to be a general purpose and adaptable platform to suit a variety of different customers needs. The platform is based on the ARM Cortex-M3 micro-controller[10]. The platform is clocked at 72 MHz and is equipped with 256 KB of flash memory and 48 KB of static random access memory (SRAM). The platform has a variety of connectors including GPIO and RS-232 for communication, but this thesis will focus on the usage of the Ethernet interface to provide Internet access. A photograph of the Midrange platform is presented in Figure 2.6.

2.2.2 FreeRTOS

FreeRTOS [22], a real-time operating system (RTOS), is used as the operating system on the Midrange platform. FreeRTOS has been developed to be a small but feature-full OS for embedded systems.

2.3 Cloud connectivity

This section introduces the concept of cloud connectivity for embedded systems by highlighting the limitations of both the cloud platform and the embedded system. When developing a middle-ware solution to enable a software developer to interface a constrained computing platform (such as an embedded system) to the Internet novel ways of handling communication, authentication, and security are often needed. Combining these constraints with cloud computing creates an even more intricate scenario. Topics concerning connectivity and

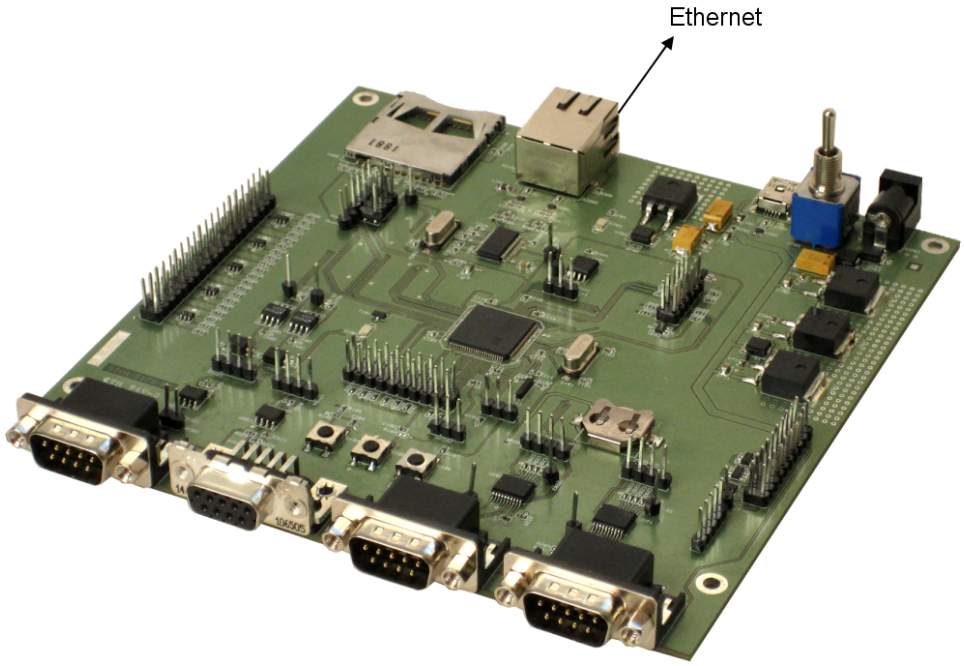

Figure 2.6: Photograph of the Midrange platform.

communication of this middle-ware with the embedded platform need to be addressed. Standard solutions such as the use of DNSSec and encrypted communication need to be evaluated. Solutions suitable for desktop computing, such as SSL/TLS and DNSSec might not be suitable for usage on the embedded systems, although they might be suitable for the end-user's communication with the middle-ware. The parties involved in such a solution are presented in Figure 2.7.

New technologies used in the cloud, e.g. load balancing, need to be evaluated with respect to their suitability, while older techniques such as storage solutions might now be accessible in a more flexible manner and also may need to be looked at and considered. Finally, the cloud providers themselves should be evaluated to determine their implementation technologies, specifically their application programming interfaces (APIs) and their test software (including emulators). This final step is crucial since deploying a solution too reliant on a specific vendor's APIs could lead to a less flexible solution than desired and also lead to provider lock-in.

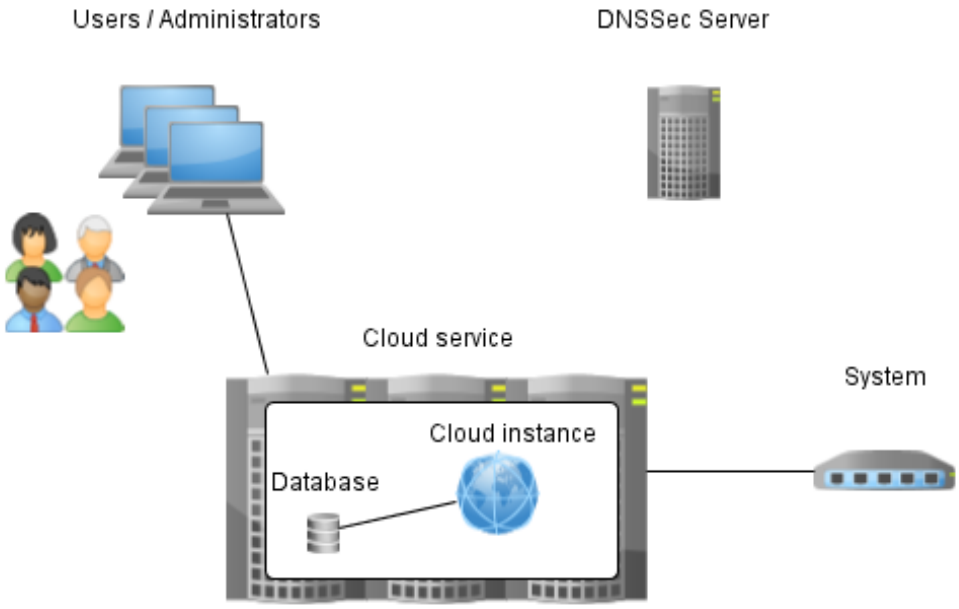

Figure 2.7: The different parties involved in an example solution.

2.3.1 Functionality

To illustrate the relationship between all the technologies and areas the following two scenarios will be considered:

- Updating the firmware of the embedded platform either by (a) a user (an administrator) explicitly performing an update by using an interface to upload the firmware to the platform, which replaces its current firmware after receiving the update correctly or (b) for the embedded platform itself to notice a new version of the firmware is available, retrieve it, and to perform the update without interaction from a user.
- A user controls the embedded system in various ways, for example via a web-interface to check properties of tasks operating in the embedded platform by issuing commands to modify runtime parameters.

The steps involved in the updating and control functions are:

1. An administrator connects to a web application via a browser and authenticates him/herself.
2. The administrator uploads new firmware to be used in the embedded system(s).

3. The administrator selects which specific units should receive the new firmware and issues a command to update them.

4. The management software establishes communication with the unit(s) to be updated and sends the new firmware.

5. Each unit verifies the integrity of the firmware and invokes the update mechanism. The administrator is notified when the unit has received and verified the firmware.

6. After the update is completed the unit reports back to the management software that the update was successful. The management systems terminates its communication session with this unit.

The steps involved in controlling a system are as follows:

1. An user connects to a web application via a browser and authenticates him/herself.

2. The user sends a command that issues a command to the embedded platform to report the value of a parameter.

3. The management software relays this command to the specific embedded platform that is to execute this command or perform some other action(s)

4. The target unit executes the command and sends back a response to the management software which in turn relays this response to the user.

The steps taken above illustrates the usage of a web-service as a middleware deployed in a cloud environment which interfaces to a back-end, which in this case are realized by the deployed embedded systems.

Chapter 3

Related work

This chapter will explore some of the work already done concerning the cloud and embedded systems.

3.1 Embedded cloud computing

Work relating to the subject area of this thesis has been conducted on so-called wireless sensor networks (WSNs) which consist of very limited embedded systems. Some of these systems have utilized web-servers and are frequently designed to be accessed over RESTful[23] APIs and be controlled from the cloud. This work is very relevant to the back-end's communication with the embedded systems and how these end systems (the embedded systems in the scenario in section 2.2.1) should be able to communicate with the middleware in order to receive commands [7]. However, the Midrange platform has considerably greater computational, storage, and networking resources than most WSN nodes.

3.1.1 Internet for embedded systems

A web-server for the Midrange platform has been developed by Joakim Axe Löfgren to serve users with web-pages stored on the Midrange platform [24]. The web-server was a basic implementation that communicated over HTTP without providing any security. Firmware updates to the platform were made by transferring a new executable's image over HTTP. A rudimentary check of the firmware's integrity was done by validating the CRC[1].

[1]Cyclic redundancy check (CRC) is a technique to verify if transmitted data has been changed after being transmitted.

The result of this work is interesting since it shows the capabilities of the platform while communicating over the Internet. Since the web-pages are being served by the Midrange platform itself the work load on the Midrange platform increases with an increase in the number of users. Having users or administrators directly interfacing with the constrained platform directly over the Internet places even more load on the platform and increases possibility that the embedded system's tasks will be interrupted. For example, if a lot of users want to access parameters in a single embedded system at the same time the system might not be able to handle all of the TCP connections or all of the actual commands. Whereas if an instance of a management system running in the cloud acts as an intermediary, then the embedded system only needs to be ask once and its response can be cached and distributed by the intermediary for some (probably short) amount of time.

Evaluation of the cryptographic capabilities of the target platform is currently being conducted by implementing various cryptographic algorithms the result are to be reported in [25]. The results from this work should be used as a guideline when choosing cryptographic algorithms for communication.

3.1.2 Cloud connectivity for mobile devices

With the advent of more capable mobile phones cloud connectivity has already made its way into constrained environments resulting in services having been developed and made available to users. Access methods such as using RESTful web services are interesting in that they rely on existing standards, such as the HTTP protocol. Having bi-directional communication functionality available, in our case the embedded system accessing the middle-ware and vice versa, provides interesting new ways of using the delivered solutions. For example, offloading of computational tasks from a mobile phone to the cloud service while providing functionality to access the mobile phones from a cloud service [26].

3.1.3 Cloud connectivity for embedded devices

The exploitation of cloud connectivity by embedded devices has been used in academic settings to enable remote access to embedded platforms [27]. The platform *mbed* has been designed to be used with a *cloud compiler*, that is, a compiler running as a SaaS on a server. This enables students to swiftly get an programming environment up and running, without having to setup their own environment. The benefits of this approach was reduced setup time and ease of emulating the targate hardware.

3.1.4 Simple Network Management Protocol

The Simple Network Management Protocol (SNMP) [28] is an established protocol for managing network equipment and Internet connected devices. The protocol is currently in its third iteration called SNMPv3. SNMP is based on around three different components: the managed device, an agent, and the network management system (NMS). The managed device is a device connected to a network and this device implements the SNMP interface enabling access to parameters in the device. The agent and the NMS, or managers, communicate using the SNMP protocol. The agent is deployed on a device, while the manager is a running on a computer and is used to manage one or more devices.

Information communicated via SNMP depends upon which variables should be available. This information is defined in so-called management information bases (MIBs).

IwIP has support for SNMP in the version that has been ported to FreeRTOS and has been deployed on the Midrange platform in the earlier project by Joakim Axe Löfgren .

3.1.5 Enterprise solutions

Currently there are companies and software solutions specializing in management of network connected devices, such as TIBCO's Rendezvous messaging product [29] and HP's Network Management Center (NMC) [30]. Comparing these two offerings shows an implementation heavy Rendezvous approach and HP using SNMP protocol resulting in a management tool that can be quickly adapted to any products using SNMP. Both offerings can be used in a cloud environment to manage and monitor the cloud resources themselves. NMC is a proprietary product that can be used in conjunction with SNMP. It offers less flexibility when customizing the end user's experience than by using Rendezvous and its many different APIs. [2]

[2] There are also open source SNMP network management systems, such as OpenNMS, Observium, Ganglia, Spiceworks, Nagios, and Zabbix.

Chapter 4

Method

Developing and deploying a web service in a cloud based infrastructure can be as easy as uploading a web-application to a web-server. For example, this could be done by uploading an application developed in Java Enterprise to a Apache Tomcat server running at a provider. However, in this approach the customer does not take advantage of the new functionality provided by cloud services and might even experience more problems. Take for example the situation of setting up the environment at a cloud provider. Choosing a less than suitable configuration, i.e. a slower VM which needs to be changed to a more powerful one or having the application less responsive because the edge-locations provided are further away can lead to a system that has poor performance.

If we instead make use of the elastic nature of the cloud we could have had the application request more resources in the event of an increased load. Evaluating the different offerings provided by cloud vendors while identifying suitable technologies for use with the middle-ware need to be conducted. By reviewing a set of cloud providers and their provided functionality in the areas presented in section 2.1.2 we get a picture on how cloud deployment of a web-service.

The elastic nature of the cloud becomes an issue when communicating with constrained end points which do not support advanced protocols and encryption schemes commonly used to establish communication with cloud deployed middle-ware. For example, creating a new cloud instance to handle an increased load would result in a new communication path needing to be authenticated and deemed reliable by both users and the embedded devices. Exploring novel approaches or adapting existing solutions must be done if deployment of a solution is to be considered reliable. Basing and extending on the proven functionality presented in the related works (chapter 3) conducted on the Midrange platform discussed in should be taken into account.

4.1 Goal of this thesis project

The goal of this thesis project is to evaluate and explore the idea of deploying middle-ware in a cloud infrastructure to interface to embedded systems and to provide functionality such as remotely updating firmware in these embedded systems, such as a web-browser.

Some of the key areas and problems that needed to be considered are:

- The communication between an end user and an end system via the middle-ware, i.e. should each of them use DNS servers and encryption protocols such as SSL/TLS.
- Addressing the difficulties caused by the elastic nature of the cloud (such as load balancers), and how to address the impact that this has on communication with both the end users and the embedded systems.
- How storing data (such as firmware) and communication critical information (such as system information and user authentication data) should be handled.
- What should be taken into account when adapting a web-service to a cloud based infrastructure. For example, what access method is most suitable and how (and if) these access methods need to be adapted. Also important is the distribution method used to deploy the service itself in order to create software images or other packages to rapidly deploy new functionality via cloud services.
- While flexibility of cloud services are always highlighted, how can we avoid problems such as being locked to a specific vendor's API or specific development tools. These solutions need to be evaluated and compared against other vendors' solutions to ensure flexibility when choosing a vendor.

When an assessment of the areas has been conducted the following points should be assessed concerning the solution as a whole:

- Propose a solution which takes into account the limitations and proposed functionality of the system.
- Work out if the solution and the platform self (cloud computing) are a viable tool for facilitating management of embedded systems. Identify technologies in the cloud computing paradigm suitable for usage. What benefits does the cloud bring to the management of embedded systems?

Chapter 5

Communication

This chapter will propose and discuss how communication with the back end systems via a web-service can be constructed. First the initial phase of discovery between the web-service, clients, and the back-end systems will be addressed in section 5.1. Later a proposed way of handling the exchange of information and commands will be addressed in section 5.2.

5.1 Connectivity

The first phase in establishing a connection between cloud deployed middleware and end systems (both the end user's system and the embedded system) involves the parties exchanging authentication messages to initialize thier communication. Establishing such a connection would involve the two parties using DNS queries to DNS servers to translate a fully qualified domain name (perhaps has part of a URL) to an IP address. DNS functionality is standard when dealing with the end user's PCs, but such functionality is not always standard in embedded systems. In the case of the FreeRTOS and wIP there is a DNS resolver. The connectivity issue also occurs in the cloud when new instances of a VM are needed to be able to know of and communicate with each other. An example of a DNS lookup is presented in Figure 5.1 where queries are sent by a computer and resolved by DNS servers.

Securing the DNS records returned by the resolving servers can be done by utilizing DNSEC in order to validate the responses [31]. This validation is done by using cryptographic algorithms to authenticate the origin of the information. DNSSEC security is based upon a chain of trust, where each of the resolving web servers validates the previously received record before further acting in the resolution process. The algorithms used are from the public-private key family and includes algorithms such as RSA and DSA. Securing the

DNS lookup phase would involve validating the DNS record on the end systems themselves using the chosen signing scheme. On the end user's system this functionality is already deployed and top level domains such as the Swedish TLD offer this functionality. On the embedded system, the validation would involve a computationaly heavy operation when the record is validated and additional functionality is needed in the DNS resolver.

The usage of DNSSEC when establishing a connection between the end user's client and the cloud based middle-ware requires that the clients validate the DNS look up responses, thus increasing the security of the communication between the middle-ware and the end user's clients. A light weight solution such as authenticating the middle-ware after a DNS lookup should also be considered. Using such a technique together with pre-configuring the embedded systems to try and contact a pre-defined IP address corresponding to a middle-ware management service circumvents the need for a DNS client on the embedded systems themselves. A proposal for such a solution will be presented in the following section. Note that a middle approach is to have a DNS resolver in the embedded system, but use a pre-configured secrete to authenticate the middle-ware after contacting it.

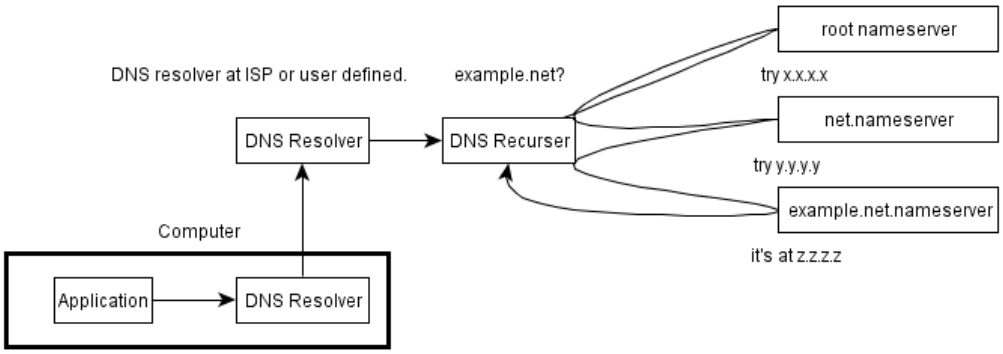

Figure 5.1: Illustration of how a DNS lookup request is completed.

5.2 Communication

Basing the communication on the REST protocol and the underlying HTTP protocol creates a flexible (in terms of clients) and abstracted view of the communication. Exposing parameters and executing commands in the embedded system can be abstracted to GET and POST commands respectively. By doing this we can create a common language for clients connecting to the middle-ware and the middle-ware communicating to the end-systems. As a

result the middle-ware acts as an authentication and communication gateway between the users and the embedded systems.

Commands would be sent from the middle-ware and vice versa by addressing functionality on both ends by URLs formated as: http://www.syntest.se/sys/1/param1. Where the number 1 corresponds to an embedded system and param1 to a requested parameter. In the middle-ware the request would be translated to an IP-address of the embedded system (which corresponds to the local embedded system number "1") and a request of the same parameter while adding authentication headers in order for the embedded system to validate the request. In this approach the URI path is used to single out a system to interact with and to pass it the appropriate parameters (and potentially a corresponding authentication header for this specific embedded system).

5.3 Authentication

The simplest way to authenticate when using a REST based protocol implemented on the end-systems is to use HTTP's basic access authentication, also called simply Basic. The Basic authentication method works by using the authorization header available in HTTP to pass a user name and password in order to authenticate the request. Such a header would look like this Authorization: Basic ZXJpazp0ZXN0. The header value passed is not encrypted but simply encoded in Base64 and thus offer no security if intercepted. If requests missing this header arrive or if the values passed are invalid, then the web-server can chose to ignore the request or simply respond with a HTTP 401 Not Authorized response code.

Relying on this method to secure a service is not considered wise. However we can continue by using the same idea of passing additional authentication parameters in the HTTP requests made to the middle-ware. We can use the fact that the embedded systems are configured in-house initially to our advantage by giving each system an AES key to be used in a HMAC authentication scheme.

In this way the middle-ware and the embedded system can mutually authenticate themselves. When a successful DNS lookup has been made a message will be transmitted from the embedded host to the middle-ware. This message is essentially a ping message for which the embedded systems expects a pong reply. Included in the reply message would be the embedded system's IP address (which of course could be deduced from the IP packet received by the middle-ware), a time stamp, and a unique identifier (UID) corresponding to the specific embedded system. Along with these parameters a signature validating these parameters will be included. This signature is computed by the

middle-ware using hash-based message authentication codes (HMACs), that is, using a shared secret to compute a signature of the hash of the messages. The hash value can be computed using a hashing algorithms such as MD5 or SHA2 (depending on the implementation complexity and resource consumption on the embedded system). The HMAC would be computed with all the transmitted parameters, the IP address, UID, and a shared secret. The HMAC is calculated with a shared secret known to the middle-ware and the embedded systems in order to avoid a third party being able to calculate the same HMAC. Examples of these messages are presented in Figure 5.2. The anatomy of the proposed authentication messages are presented in Figure 5.3.

```
POST http://syntest.com:8080/rest/device/1/ping HTTP/1.1
Location: 127.0.0.1
Time-stamp: 13:42 24:10:2012
UID: 1
Signature:  HMAC(location, time-stamp, uid, shared secret)

HTTP/1.1 200 OK
Message: OK
Time-stamp: 13:43 24:10:2012
UID: 0
Signature:  HMAC(message, time-stamp, uid, shared secret)
```

Figure 5.2: Information contained in the proposed ping and pong messages.

By using this solution together with HTTP's Basic authentication mechanism we get an authentication that is resistant to snooping and modifications by third parties. Further we ensure the integrity of the packets by requiring that access to each parameter involve successfully validating the HMAC in the header. We also ensure that the messages are only valid once, thus old requests cannot be retransmitted and deemed valid due to the inclusion of the time stamp.

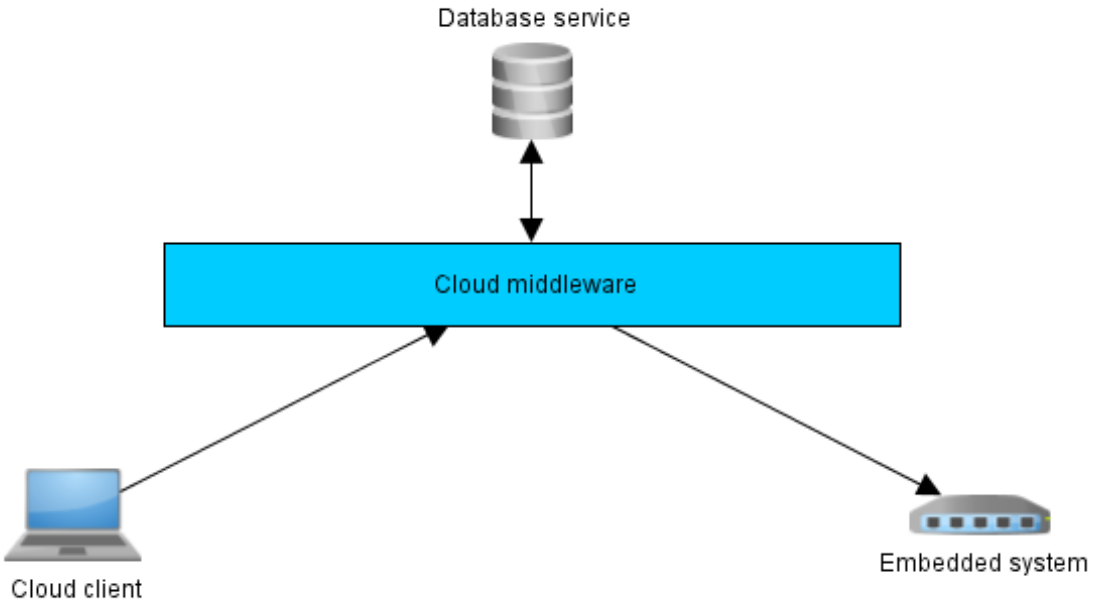

Figure 5.3: Topology of the proposed communication scenario.

5.4 Security

Further securing the communication would utilize encryption in order to ensure the data transmitted is not available to any third party. Here again standard solutions, such as SSL/TLS, [32] which establish a encrypted channel between parties can and should be used in order to secure the communication with the middle-ware. Additional functionality needed to establish a connection over SSL involves the use of (pseudo) random number generators in order to initialize the encryption schemes. Today many of the processors that are used in embedded platforms (such as the Midrange platform) include hardware to act as a random number generator

The TLS protocol is used to provide secure communication between computers by exchanging information in order to encrypt subsequent communication. In order to initiate the secure communication the parties initiate the TLS handshake protocol through which the parties negotiate the of algorithms and keys that will subsequently be used for encryption. The encryption is provided by a combination of symmetric and public-key encryption. The handshake phase is described in figure 5.4. TLS starts by the parties authenticating themselves using public keys and ends with them reaching and agreement for the use of symmetric encryption with a specific

session key. TLS is used in practice by encapsulating application protocols, such as HTTP or FTP, in order to secure them (HTTPS, FTPS).

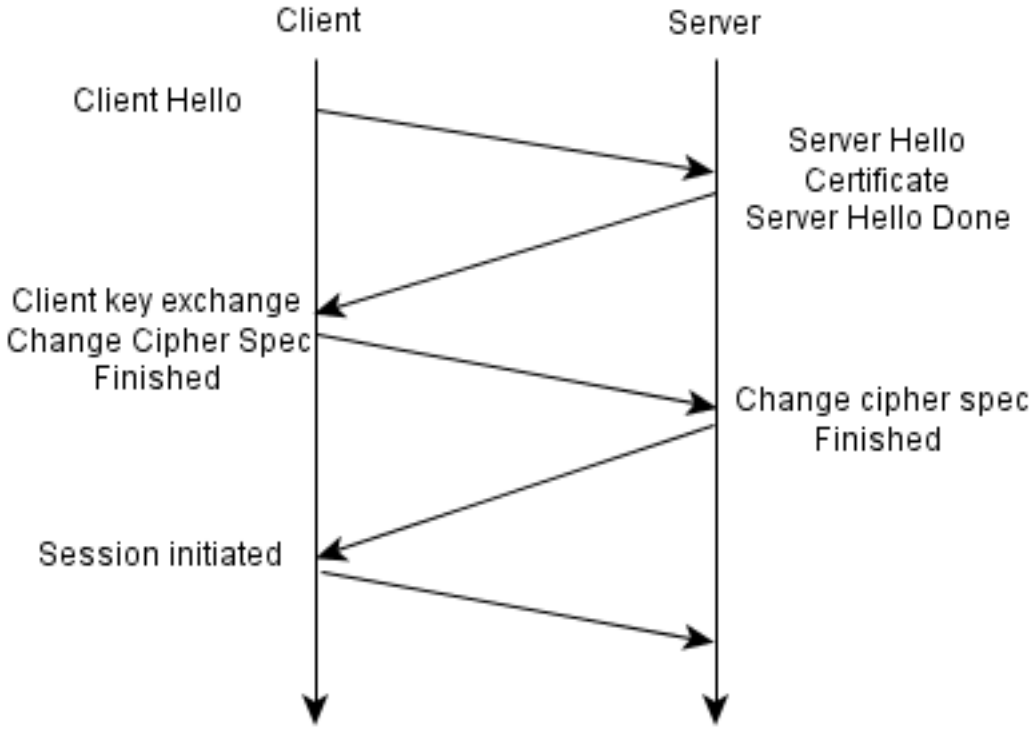

Figure 5.4: The TLS handshake phase illustrated

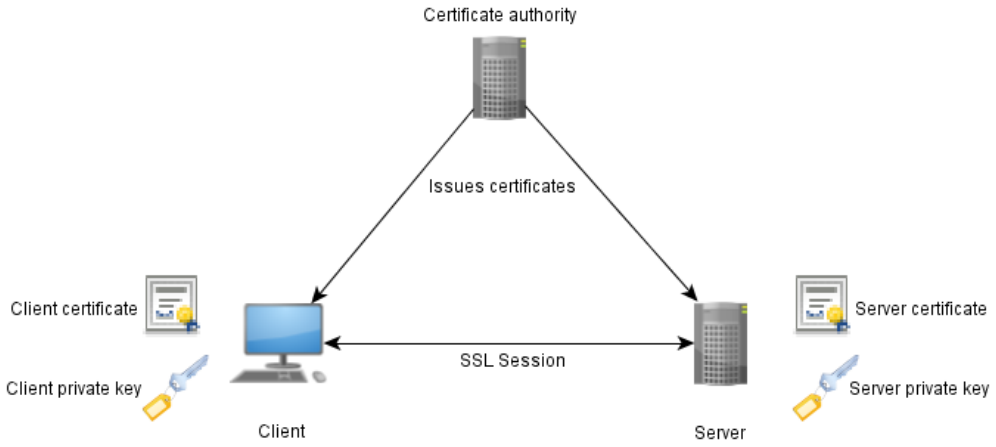

Figure 5.5: Figure illustrating how two parties use the CA in order to validate public keys.

Further securing the session is done by the usage of a Certificate Authority (CA) to act as a trusted third party to authenticate the public keys based upon the public key certificate being signed by a trusted CA.

Usage of SSL/TLS to communicate over HTTPS is an established way to secure communication over the Internet. Securing a web-application so that a user can securely transfer credentials and information is vital. Encryption used to establish this secure channel solution is relatively light weight (in terms of computational complexity). The functionality to utilize TLS is built into all modern web browsers. This functionality handles the initialization of a TLS session and the browser has a list of pre-configured CAs (along with their certificates).

By using any of the proposed protocols for authentication over SSL we now have an encrypted and authenticated way of communicating.

5.5 Message API

As an initial message protocol we will authenticate the messages by using a HMAC. The messages will be encoded using XML based definitions similar to that used for MIBs in SNMP. This allows us to define which parameters and functionality are exposed. An additional feature, is that these messages are both human readable and easily parsed. An example of such a XML is shown in Figure 5.6, this is used with a XML schema. The schema which can be used to validate the parameters that are being passed is presented in figure 5.7.

```
<parameters>
<parameter type="get" return="time">uptime</parameter>
<parameter type="get" sign="true" return="String">lock</parameter>
<parameter type="get" sign="true" return="String">unlock</parameter>
<parameter type="get" sign="optional" return="boolean">locked</parameter>
<parameter type="set" sign="optional" value="integer" return="boolean">
rate</parameter>
</parameters>
```

Figure 5.6: Defining the expected values for exposed parameters.

An example of using this message API is illustrated in Figure 5.8 for the case of a client accessing the middle-ware. Figure 5.9 shows an example when the middle-ware is communicating with the back end. As shown in Figure 5.3 the topology of the communicating systems are presented. In this figure we see that the middle-ware acts as a proxy to authenticate and store parameters while forwarding commands to the embedded system.

5.6 Summary

This chapter introduced the usage of REST on the embedded system to expose functionality via the use of middle-ware. In order to secure the communication between the different parties different solutions were presented resulting in different levels of security. In order to facilitate security the embedded systems should be pre-configured containing pre-shared keys and/or certificates depending on the level of security wanted. The most secure approach is by using SSL/TLS to communicate or additional security identifying the embedded systems and the middle-ware by the usage of HMACs with our without SSL/TLS depending on the level of security we want to achieve.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="parameters">
  <xs:complexType>
    <xs:sequence>
<xs:element name="uptime" type="xs:string">
<xs:attribute name="request" type="xs:string" fixed="GET"/>
<xs:attribute name="sign" type="xs:string" default="False"/>
<xs:attribute name="return" type="xs:string" use="Required"/>
</xs:element>
<xs:element name="lock" type="xs:string">
<xs:attribute name="request" type="xs:string" fixed="GET"/>
<xs:attribute name="sign" type="xs:boolean" fixed="True"/>
</xs:element>
<xs:element name="unlock" type="xs:string">
<xs:attribute name="request" type="xs:string" fixed="GET"/>
<xs:attribute name="sign" type="xs:boolean" fixed="True"/>
</xs:element>
<xs:element name="locked" type="xs:string">
<xs:attribute name="request" type="xs:string" fixed="GET"/>
<xs:attribute name="sign" type="xs:boolean" fixed="True"/>
<xs:attribute name="return" type="xs:boolean" use="Required"/>
</xs:element>
<xs:element name="rate">
<xs:attribute name="request" type="xs:string" fixed="SET"/>
<xs:simpleType>
<xs:restriction base="xs:integer">
<xs:enumeration value="0"/>
<xs:enumeration value="1"/>
<xs:enumeration value="2"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Figure 5.7: Example of an XML schema in that can be used validate the exposing of parameters.

```
GET http://syntest.com:8080/rest/device/status  HTTP/1.1

HTTP/1.1 200 OK
1:1,
2:1,
3:1,
4:1,
5:1,
6:-1,
7:1,
8:1,
9:1,
10:0,
11:1,
12:1,
13:1,
14:1,
15:1,
16:1

GET http://syntest.com:8080/rest/device/6/status  HTTP/1.1

HTTP/1.1 200 OK
6:-1
```

Figure 5.8: Status HTTP GET operation (from web client to middle-ware).

```
GET http://device_1.syntest.com/status  HTTP/1.1

HTTP/1.1 200 OK
1

POST http://device_1.syntest.com/open  HTTP/1.1
level:1
signature: HMAC(level, time-stamp, uid, shared secret)

HTTP/1.1 200 OK
OK
```

Figure 5.9: GET and POST operation (from middle-ware to the embedded system).

Chapter 6

Cloud providers

This chapter will give an overview of the technologies offered by cloud providers to be used to develop and run an application. Some of the services and technologies that are relevant are presented in Table 6.1. Each of these will be described in detail in subsequent sections in this chapter. AWS was chosen since it is on of the biggest cloud providers in the world while Azure for the sake of familiarity for developers already invested in Microsoft technology. App Engine was chosen as a contrast to the fully fledged IaaS providers by the fact that only PaaS functionality is provided.

6.1 Cloud services

As shown in Table 6.1 the functionality advertised by three main providers of cloud services is very similar. In some cases different terminology is used to describe the same functionality. The following sections of this chapter will try to make sense of these offerings, while discussing their similarities and individual functionality. This review will mostly focus on deployment functionality such as storage, load balancing, and frameworks as these are the functionalities needed when developing and deploying an application.

Table 6.1: Table presenting a subset of technology provided by three cloud vendors.

Service	Amazon Web Services	Microsoft Azure	Google App Engine
Architecture	EC2 virtual machines based on machine images managed via an API.	OS and services individually available or used in conjunction.	Distributed architecture
Service	IaaS/Paas	IaaS/PaaS	PaaS
Virtulization management	XEN hypervisor	Hyper-V derived hypervisor.	Java VM or the Python runtime running applications
Load balancing	Elastic load balancing	Built-in load balancing	Automatic scaling and load balancing
Fault handling	Availability zones	Fault domains	Fault-tolerant servers
Storage	Amazon Elastic Block Storage (EBS), Simple Storage Service (S3), SimpleDB, Relational Database Service	BLOB storage, tables and queues as well as SQL	App Engine datastore
Security	API over SSL, X.509 Certificates, access lists, SAS 70 Type II Certification	SAS 70 Type II Certification	Propitiatory Secure Data Connector, SAS 70 Type II Certification
Frameworks	Amazon Machine Image (AMI), MapReduce	.Net	Scheduled tasks and queues via Java and Python. Memcache

6.2 Cloud storage

Deploying an application as a cloud service involves trusting the cloud provider to keep your data secure. This section will look at different ways of storing data in the cloud, how a cloud provider secures its user's data, and how additional security can be applied to secure the application's use of the stored data. A clear distinction can be made between storage of data and the usage of cloud based databases. The distributed nature of storing files in the cloud is closely linked to the CAP theorem [33]. This theorem states that its impossible to guarantee consistency, availability, and partition tolerance at the same time in a distributed system [34]. Consistency is usually the first trait to be compromised, hence many storage solutions are said to offer *eventually consistency*. This

in comparison with the consistency that is expected from database systems guaranteeing the atomicity, consistency, isolation, and durability (ACID) set of properties. Unfortunately, an ACID system will not be very scalable [35].

6.2.1 Cloud storage

Cloud storage is basically defined as storing data in a cloud service where a third party operator provides the underlying infrastructure for hosting files. Storing data in the cloud differs from using a dedicated server to store files in the same way that hosting a web application in a VM in the cloud differs from providing a web server on a single machine do. In a cloud based storage solution data can be distributed across the infrastructure to edge locations to provide faster access, while still being indistinguishable from using a classic storage solution, thus an end user application will not know the difference when accessing the files from different servers. The nature of the cloud results in a storage solution that is less susceptible to failure in that the data is stored in a distributed manner and potentially the data can be stored with some defined level of redundancy. Local storage (e.g. storage connected directly to the running VM where programs are executing) is seen as little different from what is expected from a local running machine with the main difference being that if the VM instance is removed then the data is also removed. Persistent storage is needed and this persistence is offered by a variety of different techniques.

Persistent storage is offered by provider solutions such as Amazon's Simple Storage System (S3) [14] and Microsoft's Windows Azure Storage (WAS) [36]. Storage is realized by reserving virtual containers or buckets accessible over one or more different protocols. The functionality is limited to put and get commands utilized via a key and value API interface.

Storing data securely is a concern when outsourcing data warehousing. Trusting a cloud storage vendor and their security practices are crucial. Since the cloud computing paradigm offers very little configuration of the underlying infrastructure and the security practices deployed these areas need to be studied before selecting a vendor.

6.2.2 Cloud databases

Using the cloud to deploy databases on virtual machines or renting solutions offered by the cloud providers (Data as a Service) are common means of providing a database in the cloud. Deploying a database onto a virtual machine differs very little from traditional database usage with the main differences due to the initial configuration and deployment phase where virtual machines images including a database need to be created or bought from third parties

. Getting a VM image from a third party reduces the effort needed for configuration and interaction, but requires that you trust this third party. Deployment can be as easy as uploading the virtual image to the cloud and starting it. Shortly after the VM starts the database is ready to be used.

Deploying a commonly used database application such as MySQL, requires reviewing this application's functionality in order to asses its suitability. An example of using MySQL will be used in order to identify problems and how these problems can be solved in order to give an illustration of how one specific database could be used.

Since scaling is a very desirable feature of the cloud having MySQL scale such that it can be distributed over multiple instances at edge locations is desired in order to handle high load [37]. MySQL offers this functionality by configuring it to use master-slave replication. In this way multiple MySQL slave instances replicate the data from a master instance to offer redundancy and higher performance. This enables the deployed applications to intelligently utilize the different copies of the database. For example, we can direct all INSERT queries to the master database, while distributing all of the SELECT queries over the read-only slaves. Scaling can also be achieved by using a cloud storage provided by vendors. It is also possible to stop an instance of the database and reconfiguring it and redeploy the database on a more powerful VM. This enables us to use the same storage for the database while utilizing a more powerful VM instance. Note that in this approach we are still running one database rather than running it on the same server as before, but we are now running it on a bigger and faster host. One benefit of using proven database solutions is the availability of middle-ware API's such as JDBC and ODBC[1] to interface to the database.[38] Cloud providers, such as Amazon, also offer their own variant of scaling MySQL servers (for example, the Amazon Relational Database Service) based on replication. These virtual services offer the desired database functionality and they are pre-configured and ready to deploy [14]. A comparison of the standard database topology and a more flexible variant are presented in Figures 6.1 and 6.2.

[1]Database Connectivity (DBC) drivers for C and Java

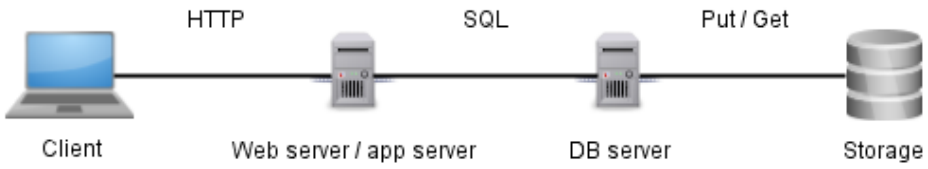

Figure 6.1: Traditional database topology

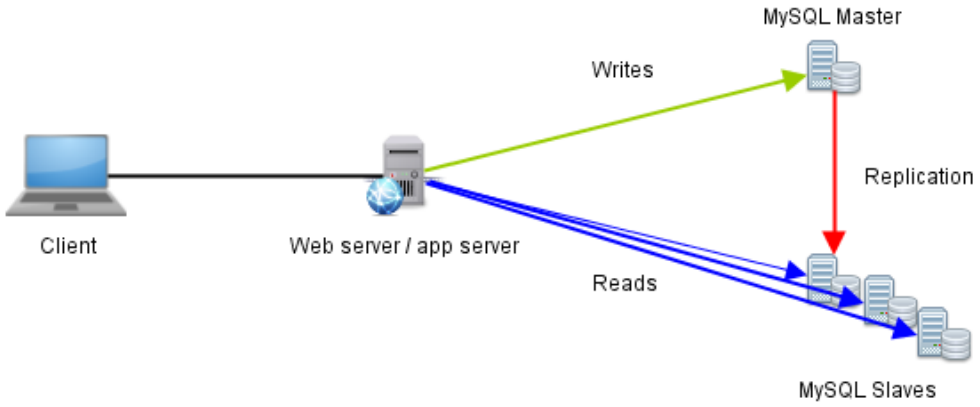

Figure 6.2: Databases configured to be deployed in a distributed scenario.

The security aspects of running a database in the cloud are important, thus we must configure the VM running the database to only be accessible internally to the cloud on the standard MySQL ports.. Additionally, we might keep only one other port open for remote access via SSH. This simple approach limits the VM's exposure to external threats. Another important security aspect concerns the data itself. Using cloud provided encryption of the local storage of the VMs offers additional security for the data, by preventing it from being accessed by other applications. However, it does not protect it from being accessed from application by a compromised instance of our own. Encrypting and hashing data stored in the database itself is a commonly used solution, even on databases deployed in a traditional server infrastructure.

6.2.3 NoSQL

The nature of the cloud with its elasticity and flexibility gives rise to an environment where traditional databases (such as MySQL) might not be the

best choice. Instead, non-relational databases (such as MongoDB [39]) have been designed in order to better suit the cloud paradigm. These non-relational databases are commonly deployed to handle large distributed databases [40]. One of the results of utilizing such a non-relational database is a database service which scales more easily. This functionality is achieved by a relaxing the ACID properties of relational databases. Thus rather than having a guarantee that a database transaction is reliable the NoSQL paradigm views transactions as being eventually consistent. The drawback of deploying an application from a traditional infrastructure onto the cloud is if the database service desired is a non-relational database, then the application could require a complicated rewrite. These non-relational databases are mostly used when dealing with large amounts of data being served to users via a web deployed application. Since the intended use of the service is currently not at a state where large amounts of data will be collected and distributed the benefits of using a non-relational database are not as relevant as the database services offered by cloud providers or simply deploying of an database server on an instance of a VM at the cloud provider.

6.2.4 Data as a Service

The main feature of Data as a Service (DaaS) is that the database is not managed by the user, but rather the database is managed by the provider. DaaS providers usually offer a set of flexible variants which differ in the maximum amount of storage space they provide. Products based on two techniques, deployment of a traditional relational database and a NoSQL variant, are offered by providers [41]. In comparison using the cloud provider's infrastructure to deploy databases (in a IaaS) we can use a vendor provided database to provide the desired database functionality. Amazon's SimpleDB offers a vendor provided variant of scaling and redirecting queries to a traditional database (such as the solution proposed earlier). The drawback of using such a product is the limitations in interaction with the database. Commonly used operations such as inserting and retrieving data are supported, but more advanced functionality offered in SQL such as join operations are not supported.

6.2.4.1 Amazon Web Services

Solutions based on file storage, such as AWS S3, support binary objects ranging from 1B to 5GB. These objects are accessible over SOAP and REST based interfaces using put/get commands [42]. Meta-data identify the files and can be updated individually for each object. The drawback of selecting files based on meta data is that selection is provided by key-value interaction. In this

approach each object is uniquely defined by a key which can be used to retrieve the object. If a selection is instead to be made based upon meta-data, then all of the meta-data needs to be retrieved for all objects and the object will be identified by identifying the key(s) for the object(s). An exception to this rule occurs when objects can be retrieved based on their timestamps. Security groups can be defined and configured to allow individual user permissions (such as read and write) to control access to a defined subset of objects.

6.2.4.2 Google App Engine

Solutions based on the use of Google App Engine are limited in the same way as the interaction capabilities of a database due to App Engine using its own query language GQL[43] (a abstracted and simplified variant of SQL). App Engine is based on the same idea of partitioning and replication as discussed earlier and is available in Amazon's RDS. Benefits of using App Engine's database functionality is the possibility to use distributed memory caching pre-configured to speed up database interaction by caching data in RAM in order to reduce the load on the underlying database layer. The App Engine cloud platform provides two distinct data storage solutions: Cloud SQL and Cloud Storage [44]. The former is a relational database based on MySQL and the latter is a storage service for blobs (binary large objects). Basing its relational database on MySQL gives the developer a familiar interface and the possibility to move an existing database effortless onto Googles platform. Google promotes the usage of a High Replication Datastore (HDR) over a master and slave configuration to achieve consistency. HDR is based around the Paxos distributed algorithm. Google replicates data across its data centers using its content distribution network (CDN).

6.2.4.3 Windows Azure

Windows Azure Storage provides three distinct kinds of storage solution to users. These solutions offer the possibility to store data in blobs, tables, and queues [36]. The blob storage is used to provided access to data both for use internally in the cloud and also to users over a streaming service. The service is intended to provide infrastructure to store unstructured data that has no indexing. Functionality is provided to select data using queries and to handle large amounts of data (up to 100TB from a single account). Files stored in the blob store are further divided into containers in order to group data and to create a subset of data in a logical manner. For example, Figure 6.3 shows a container for firmware that has been created to hold firmware images. Each container is owned by a storage account. This account can be used to

administer the data store. Referencing a specific blob is done by the following syntax: http://syn.blob.core.windows.net/firmwares/001.bin for the example in the figure. Functionality to access the blob store in a C# application is provided by a client library provided by Azure.

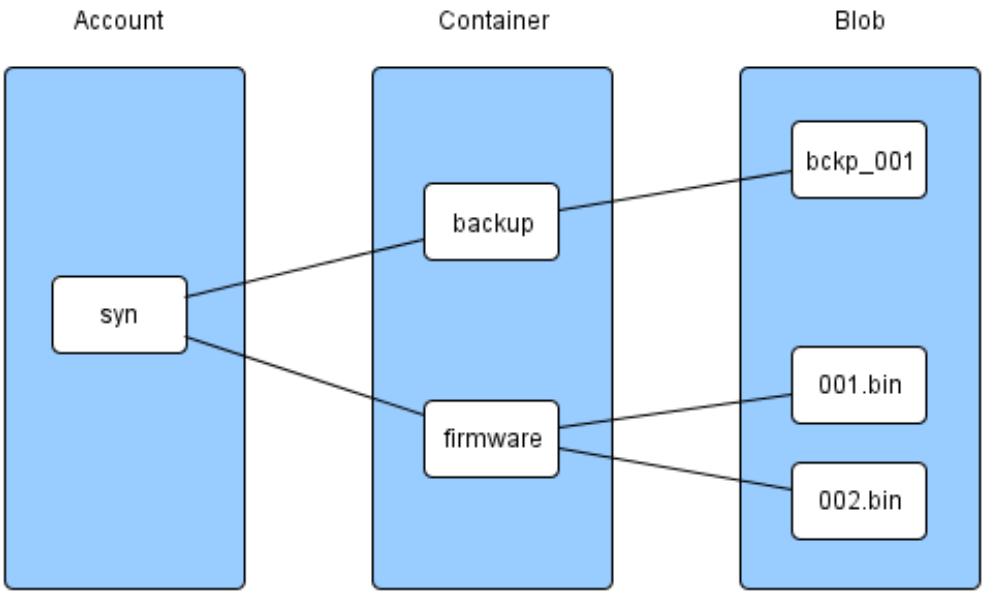

Figure 6.3: Topology of a Azure BLOB store.

The table service provided by Windows Azure also provides functions to store structured data using the NoSQL approach. The service is secured by only allowing authenticated access to the stored data when a query is executed. The information is stored in entities indexed in tables contained in name-value pairs, i.e. email, uid etc. This is shown in Figure 6.4. The tables are exposed using the Open Data (OData) protocol in a similar fashion to how JDBC or ODBC can be used to access an SQL server. The OData protocol is XML based and returns the entities in XML format for further processing. The access URL is constructed in the same manner as Windows Azure blob storage. For example the URL : http://syn.table.core.windows.net/users is used in the example shown in figure 6.4.

The queue storage is a storage solution for messages up to 64KB in size. This queue storage acts as distribution service from which client applications can access messages that have been placed in a FIFO queue using standard queue operations (such as GET, PUT, and PEEK). This provided functionality is meant to facilitate message passing between applications running in the cloud

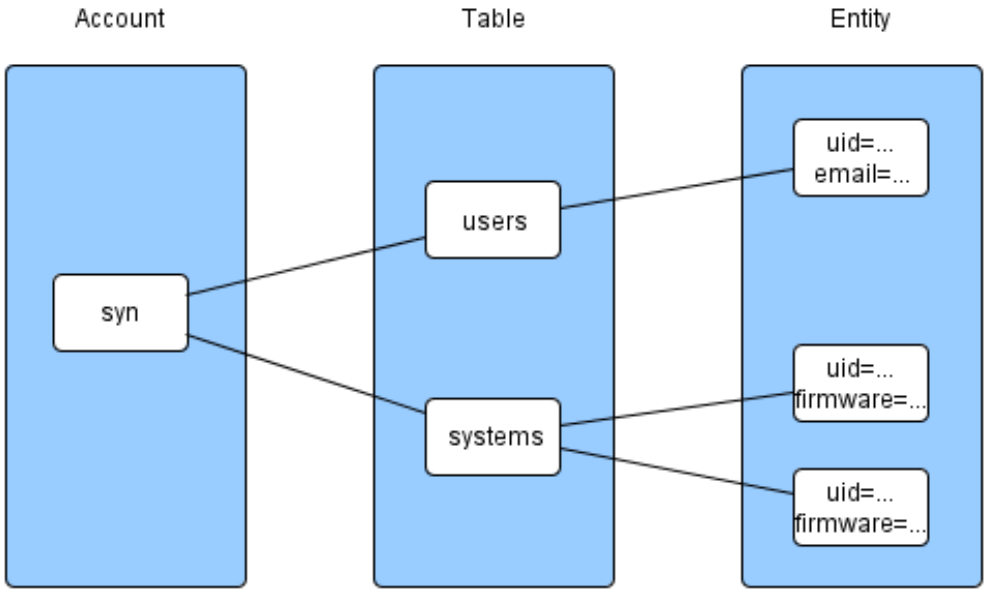

Figure 6.4: A classic relational database offered by Microsoft Azure.

over generic interfaces, such as REST. These functions are designed to be portable and they can be used in programs written in different languages. The messages themselves can also be constructed to enable cross platform compatibility by using XML. The result is highly a portable message format. Although the queue service supports an unlimited number of messages , the maximum time a message can remain in the queue is limited to a week. Consistency in the queue is also guaranteed, the order of the messages stored in the queue is not guaranteed, and duplicate messages may be returned by sequential GET operations. In order to ensure that messages are processed correctly after they have been retrieved from the queue the messages are not explicatively deleted when fetched from the queue, instead the messages are made invisible (e.g. they are still in the queue but will not be returned by GET operations). When a message has been processed the client who processed the message can actually delete the message from the queue. However, if this operation never takes place (for example the client handling the message crashes) the message will reappear in the queue after a timer connected to the visibility of the message times out. This will enable another process to retrieve this message and process it. Under high loads there is a risk that a message may be processed, but not deleted - hence when the timeout occurs it will be processed again!

6.2.5 Implementation

If the previously mentioned ways of securing the messages transmitted between the end systems and the middle-ware were to be used, the pre-distributed keys residing in the embedded systems also need to be available to the middle-ware in order for the middle-ware to authenticate these messages. Storing these in a cloud based database in plain-text would be a bad idea. If the database is compromised then spoofing messages intended for the embedded systems becomes trivial. The same scenario would arise as using public key encryption when the private key associated with the middle-ware needs to be accessible to the middle-ware. Using common techniques such as hashing login credentials with a salt in order to avoid having them available in plain text is not feasible in this scenario since the encryption keys actually need to be available in order to encrypt messages and payloads. Using a weaker form of encryption based on the user's password (PBE) to derive an encryption key is feasible.

Keeping to the intended functionality the content of the database should be encrypted, but different access levels could be utilized. For example, the update could be encrypted with a public private key where the private key resides in the client software. This approach creates an environment where the contents of the database cannot be decrypted in the web-application. This avoids the need for the database in the cloud to store private keys. However, it means that the clients need to be able to support the asymmetric cryptographic algorithms needed to be used with a private public key pair.

6.3 Load-balancing and fault tolerance

As shown in Table 6.1 the load balancing features of the three providers were described with different descriptions. Their specific functionality will be reviewed in this section in order to differentiate these three offerings.

6.3.1 AWS

Amazon offers services using well established techniques in order to scale an application, such as employing the N + 1 approach [14]. Using this approach in the AWS cloud means that there are N instances providing a service running at a given time and a fall back instance (+1) providing redundancy. Rules to detect load and adding more instances can be configured, along with rules of when to terminate instances. In contrast to this scaling approach of load-balancing AWS also offers a service called Elastic Load Balancing. Elastics Load Balancing distributes traffic to identical instances using different algorithms and heuristics. The heuristics can be configured to measure the load based

on CPU and memory availability or connectivity (such as network load to a specific instance). Load balancing is provided transparently to the end user by having the load balancer itself (the logical load balancer) addressed when an application is accessed by connecting requests to the load balancer via an externally available DNS name. The elastic load balancing features are also available internally in the cloud and can be used between application layers, such as accesses from application to a database.

Problems arising from using load balancing, such as a user being connected to a specific instance (or another type of back-end) after login are solved by using sticky sessions or session affinity based upon cookies. Authentication of the users and the back-end are still available but with a slight modification. If communication is supposed to be secured by using SSL, then rather than using a different certificate for each instance we can use one certificate for the elastic load balancer. This reduces the effort need to reducing some of the configure of the individual instances. SSL is also available internally by using private and public key cryptographic algorithms. The load balancer validates each back-end internally and encrypts the communication within the cloud.

In order to increase an application's fault tolerance different availability zones (AZs) can be defined. These zones consist of a subset of instances that have been compartmentalized and are independent from other instances in different AZs. Deployment scenarios illustrating this include creating a AZ containing a web-server and a database, then doing load balancing over the AZs rather than specific instances of the web-server and database. Instances and zones can be deployed around the world in different data centers which serve different regions further distributing the load.

6.3.2 AppEngine

Unlike AWS, App Engine offers much less for the developers to configure. This enables developers to focus on their application's functionality. The scaling and load balancing functionality is offered under the banner *automatic* [45].

6.3.3 Azure

Windows Azure offers load balancing functionality via a solution branded as "Windows Azure Traffic Manager" [46]. This traffic manager (TM) is similar to the offering from AWS in that it can be configured to distribute traffic to services in the same data center and to different edge locations. Traffic is distributed to services via the TM by having the TM respond to DNS lookups. The TM returns the IP address of different hosted services. As a result no data is passed via the TM (only IP addresses), unlike the case of the AWS Elastic

Load Balancer. This technique of load balancing is facilitated by using DNS records and setting how long the DNS records are valid. When a record's time-to-live (TTL) value expires the user's system performs another DNS lookup. This enables the client to utilize a new instance, or perhaps even the same instance selected in the earlier lookup. All of this mapping is determined based upon by the policy configured in the TM. The default and recommend TTL value for the DNS records is configured to be 300 seconds. This approach can encounter problems if DNS records are cached or software does not honor the TTL values. This will lead to unexpected behavior of the load balancing process , as clients will continue to utilize the same instance for longer than they should.

The TM offers different policies much like the AWS solution, but the policies are limited to three choices and does not include a general health policy as AWS does (basing the routing on infomration from the instances such as CPU load, memory usage and bandwidth). The three policies are: performance, fail over, and round robin. The performance policy selects a host to direct traffic to based upon network performance. The network performance is assessed by examining the round trip times (RTT) based on the user's location and the data centers hosting the requested service. This examination of the RTT is not done by examining the hosted service's performance, but rather simply based on looking at the originating user's IP address and the data center's IP addresses. The results are not a indication of real world performance, but simply an indication of what should be the best route to take.

The fail over policy is just what the name implies. If a service is deemed to be unreachable, then the traffic will be routed to the next pre-configured service in the policy. If the fail over instance is also unreachable, then the policy will select yet another service and so on. The round robin technique directs requests to the service next in line to receive traffic. When another request comes the subsequent service will receive the traffic and when the last service in the pre-configured order has been directed traffic, then the first service will yet again receive the request; thus creating a loop over the different instances of service.

Monitoring of the service's availability is configured manually by specifying an end point for a HTTP GET operation. The parameters for this monitoring include the time between checks. Additionally, the number of retries can be manually configured to suit the deployed application's characteristics.

6.4 Web service centric features

This section will address implementation centric features, such as the supported programming languages and the availability of APIs to interact with the provider's cloud infrastructure. Deployment facilitation functionality, such as the creation of packages and deployment via integrated development environments (IDEs), will be presented. The run time environment (such as which OSs are supported) will be presented for each of the IaaS providers. The the hosting technology, such as specific web-servers, will be addressed.

6.4.1 Programing languages

The main distinction to be made between the language support provided by the providers are by the different provided APIs. Since both AWS and Azure provide functionality to deploy entire virtual OSs their support for individual programming languages is trivial. It is simply up to the user to install and configure their chosen environment. Both AWS and Azure instances can be deployed with Microsoft Windows and Linux based OSs. Therefore .Net applications can be deployed in conjunction with applications running on Linux based OSs. In contrast, App Engine is a PaaS provider hence they do not offer this functionality and the offered functionality is limited to a defined set of languages. The languages by the providers when interacting with the cloud service itself are presented in table 6.2.[2]

Table 6.2: Table of supported languages.

Platform	**Languages**
AWS	Java, PHP, Ruby and .NET. [49]
App Engine	Java and Python (2.5). [45]
Azure	.Net, node.js, PHP, Java, and Python (2.7 recommended). [50]

The languages listed in Table 6.2 are available for use with the provider APIs. From this list we can conclude that the supported languages are all high-level languages and that there are similarities between the three providers. The only languages supported on all platforms are Java and Python. The inclusion of the version numbers for Python is necessary since Python has two currently branches: versions 3.0 and the 2.0. None of these providers currently support Python 3.

[2]This table does not contain languages available in a user customized VM or running on a another languages VM (such as Groovy [47] or Clojure [48]).

Both AWS and Azure provide a language independent API based on REST in order to use vendor provided functionality via language *independent* interactions. The interaction with the cloud infrastructure can be abstracted and made more available by using suitable APIs. The next section will address the functionality provided by the APIs when an application is implemented using one of the supported languages.

6.4.2 APIs and toolkits

In this section APIs and toolkits provided by the three providers will be presented and presented with examples of their interactions.

6.4.2.1 AWS

Amazon provides developers with language specific software development kits (SDKs) to enable developers to get started more easily. The provided library's provide interaction capabilities with the earlier mentioned AWS services, such as EC2 and S3. For example, the Amazon provided boto framework for Python to create a bucket in the S3 service in order to store a file would be coded as shown in Figure 6.5.

Notice the lack of authentication is handled by the developer when storing a file on S3. When using the available frameworks of complexity is hidden by the framework.

An even more comprehensive offering is provided for Java developers in which not only is a SDK provided to abstract the services, but an Eclipse plug-in exists to manage deployment and to facilitate debugging AWS interactions. The same functionality is available for .NET developers in the form of a plug-in for Visual Studio or via a stand alone deployment tool for .NET applications. Tools to facilitate deployment of applications developed in the other two languages (PHP and Ruby) are available in the form of command line applications that can be used in conjunction with the Git version control software to configure and deploy applications to the AWS infrastructure.

```
import boto
s3 = boto.connect_s3()
bucket = s3.create_bucket('firmwares.syntest.com')
key = bucket.new_key('fw/001.bin')
key.set_contents_from_filename('/home/ee/firwares/001.bin')
key.set_acl('public-read')
```

Figure 6.5: Storing a file in an S3 bucket.

6.4.2.2 App Engine

App Engine offers SDKs for Java and Python and both take advantage of language specific offerings which are compared here. Deploying a Python based web application on App Engine is based on using a web server gateway interface (WSGI) to interact with the provided features. WSGI is a protocol which acts as a universal interface connecting web-servers, application, and Python frameworks by a common interface. App Engine provides a framework called webapp2, but frameworks such as Django and Python Pylons are also compatible. When dealing with the provided services App Engine provides APIs for interacting with Datastore, Memcache, URL Fetch, mail, and image services.

As mentioned earlier when presenting the App Engines Data Store solution this solution relies on its own language derived from SQL. This limited variant of SQL and the capabilities of the API to provides the modeling of the entries provide the programmer with logical representations of objects. An example of the abstracted interaction available when using this functionality is presented in Figure 6.6. In this example program in the persons are now available in the database and queryable with query's such as "SELECT name FROM Person".

```
from google.appengine.ext import db

class Person(db.Model):
  name = db.StringProperty()
  uid = db.IntegerProperty()

erik = Person(key_name='Erik', uid=1)
erik.put()
Person(key_name='Jean', uid=42).put()
Person(key_name='Oskar', uid=2).put()
Person(key_name='Eric', uid=3).put()
```

Figure 6.6: App Engine database abstraction.

Using the memcache service we can speed up database querying by storing the database in the cache along with an expiration time (to mark entries as old). An example of how memcache functionality is accessed is presented in the example in Figure 6.7 in which a firmware version is stored for one hour. Having the application check this cached version of the firmware instead of querying a database provides a faster response, although the application could serve an old version of the firmware until its expiration timer has expired.

The same API functionality is offered for applications using the Java API.

```
from google.appengine.api import memcache
memcache.add(key="firmware_version", value="2", time=3600)
```

Figure 6.7: GET and POST operation (middle-ware to back-end system).

Another interesting functionality offered by App Engine is the availability to schedule tasks. This is done by Cron Jobs which executes HTTP GET operations on URLs and also a script on the result of the GET. This is similar to the use of Cron on a local linux machine used with curl. Both programming languages support a Development Server which mimics the functionality of App Engine. As with AWS, a plug-in for Eclipse is also available to interact with the development server and the cloud service itself.

6.4.2.3 Azure

As for the other providers, Azure offers libraries providing abstracted access to its services. Access to Azure's storage services (Blob, table, and queue) are provided for developers to utilize when coding an application for deployment on the Azure cloud. One area where Azure goes one further than the other providers is to make development easier on its service through an Azure environment emulator that enables the user to test and debug a distributed service. Azure is the only provider that I have examined who offers such a solution for developers (in terms of emulators provided by the company itself). Another feature provided by is the default use of Microsoft's Windows OS to run its instances via remote desktop. These Windows instances can be enabled to be controlled via a remote desktop. This functionality gives the developer access to the instances in a familiar manner, much like accessing AWS instances over SSH.

6.4.3 Servers

This section explores the web-server provided by the providers. Earlier we saw how the different providers supported applications developed in .NET and Java. The web-servers supporting .NET applications are easily deduced. Internet Information Server (IIS) is available on Amazon EC2 instances and of course on Windows Azure. With respect to developing and deploying a Java Enterprise application on the examined providers, deploying such an application would need an application server such as GlassFish [51] or Apache Tomcat [52].

When deploying a Java Enterprise application on AWS the following steps are needed:

1. The first step is to start an E2 instance of your choice.
2. This instance can be configured by installing a Java Enterprise server such as Apache Tomcat.
3. When the installed web-server is ready to use, then the application to be deployed is loaded onto the server. For example, we could upload a web application archive (WAR) into the web-server.

The Java Enterprise application is now ready to be used and manageable via AWSs, for example it could be configured with a load balancer. A similar process is done in order to deploy a Java Enterprise application on Windows Azure. Deploying a Java EE application on App Engine has some restrictions. The App Engines server is based on the Jetty web-server [53] – which is a HTTP server, client, and javax.servlet container. This web-server comes with only a subset of the Java EE technologies. Some of the Java EE APIs which are not supported are:

- Enterprise Java Beans (EJBs),
- Java Database Connectivity (JDBC),
- Java Message Service (JMS),
- Java Naming and Directory Interface (JNDI),
- Remote method Invocation (RMI).

However, the main functionality, such as Servlets and JSP, are available. In the above list we see that the functionality offered by JDBC is **not** available, but Datastore interaction can be done by using Java Data Objects (JDO) and Java persistence APIs. Other commonly used libraries used by web-applications, such as Apache Commons FileUpload, are not supported because the App Engine does not expose a writable file system – thus crippling the usage of the library.

6.5 Provider summary

A clear distinction can be made between the different technologies used by cloud providers to deliver DaaS. Solutions more suitable for storage of BLOBs are available in Amazon's S3 solution. Use of such functionality is exploited by web applications delivering media to end users. Information that should be provided in a queryable fashion, such as information about users, would benefit

from traditional approaches such as relational databases, in order to be able to select users based upon complex criterias, or provider solutions with a limited query language. The benefit of using the latter solution is reduced maintenance since providers as a rule do not give customers access to the underlying software solution and data management, hence these are the provider's responsibility rather than the application developer's responsibility. These divisions are shown in Table 6.3

From the three examined providers we can divide the offerings into three product categories with respect to database functionality:

Table 6.3: Division of vendor provided databases.

Database offered	**Provider offerings**
Relational Databases	Amazon RDS, Microsoft SQL Azure, and Google Cloud SQL.
Object storage	Amazon S3, Windows Azure Blob Storage, and Google Cloud Storage.
NoSQL	Amazon DynamoDB, MongoDB on Azure, App Engines Datastore on top of BigTable.

When considering the use of a relational databases in the cloud a point can be made about how comfortable a developer is with the syntax of the chosen product. Azure's advantage is that it uses Transact-SQL, the same language used in Microsoft's SQL Server. This in comparison to the syntax used by Amazon and Google's relational database offerings which are only used in their respective environments and not thus available in local solutions. The object store offerings on the other hand all utilize REST to access the storage. This is a standard HTTP syntax which offers a familiar way of access storage in the cloud. The NoSQL approach of accessing and storing data in the cloud is a novel way to store data in the cloud. Having data available in a key-value manner can be very useful for applications which require a lot of data. However, one could argue that this is not the case for this project. For example, the need to store a lot of different firmware versions is unlikely to be necessary, hence this functionality would not benefit the deployment scenario which we envision.

The distributed nature of deploying a web service behind a load balancer raises the question about how session affinity is supported. As discussed earlier this feature is provided in very different manners. When using AWS ELB the feature can be enabled via the configuration center, while no support is provided by Azure. In Azure a client must instead be linked to a server by the DNS record returned from the TM and the connection is routed to this same server until the DNS response cache time has expired. In contrast, Google limits the use of

sticky sessions currently this functionality is not supported.

Concerning the runtime environments themselves we saw that the offerings range from AWS (where you can deploy applications however you want on VMs) to Google's limited environment where support was limited to a a few languages. Java and Python were supported by all providers, but their functionality was limited in the case of App Engine where Java Enterprise support was limited. All of the providers provide APIs to be used when developing applications, but these APIs are not interchangeable which creates an environment where applications need to be ported to another provider if you are to change from one provider to another.

Chapter 7

Analysis

This chapter will present the proposed deployment scenario and also the specifics concerning the implementation of a solution to such an environment.

7.1 Deployment

The solution of using a middle-ware deployed in a cloud environment creates a flexible environment for clients and the embedded systems. Having the middle-ware to interact with the cloud services such as databases and load-balancers creates a scenario where stand alone clients interfacing with the middle-ware become independent and act only as presenters of data. For example, different clients based on different techniques such as web-based clients running in browsers, smart-phone applications, and standalone applications can be developed to use an API provided by with the middle-ware interfacing the embedded systems. This creates a view of the cloud deployed middle-ware as a proxy which acts as a relay between the parties.

We also get other desirable features such as a central point for defining and allowing different levels of access to the embedded systems and a central point for logging interaction. By standardization of the interaction via techniques such as web server description language (WSDL) we have a way for clients to rapidly be configured to the exposed API. More programmatically we can abstract the embedded systems as objects returned from API calls (for example JavaScript Object Notation (JSON) objects) in order to have the current known state of the embedded systems exchanged as objects.

By having the middle-ware act as a stateless interface for the embedded systems we arrive at a scenario suitable for deployment on cloud platforms. Considering the scenario of having to add an instance when the middle-ware is under high load, the two middle-ware instances now handle API calls

independently. This is achieved by distribution by a load balancer. The two instances now interact with the data back-ends such as databases and honoring configuration options such as cache timers resulting in a flexible environment where the two instances support all kinds of clients. Alternatively we could use web application directly interfacing between the back-ends and embedded systems. Adding another instance to this scenario would mean that we have two complete services providing data to the users and we have less flexibility when consuming and presenting the data. We would need to store the sessions with which the users are interacting and have logs in order to provide synchronization. When removing and adding instances to the scenario we would have to ensure users are not distributed over the different hosts during a session in order to have consistent interaction.

The stateless nature of the middle-ware also gives us the benefit of an fixed set of dependencies and configuration parameters to be compiled and used when a new instance is added in order to limit the amount of time necessary to setup an instance.

We also achieve isolation of the different services where we can expose different functionality in different clients. For example isolating administrator instances versus client instances. We can provide a more fine grained configuration of access control via clients. We also limit the exposure of configuration files and interaction parameters for the embedded systems leaking from clients by having all such data in the middle-ware and not in a web-client instance directly interacting with the embedded systems and other back-ends. There is no database interaction (such as SQL queries) or calls to the embedded systems directly available for the clients to make.

7.2 Environment

The subject of leveraging the clouds capabilities to bring new functionality to the embedded systems are closely linked to how secure communication can be setup between the different parties. Modern solutions such as SSL/TLS and DNSEC was introduced as a solution for solving these kinds of problems. However when dealing with a constrained system such as the Midrange platform different levels of security might be considered in order to not put too much strain on the platform. These kind of parameters are closely linked to the evaluation of the cryptographic capabilities of the platform already conducted. The levels of security was presented as a basic authentication scheme providing no additional security over a Base64 encoded password to and authentication scheme relying on pre-distributed secrets to be used to authenticate messages using the HMAC technique.

This approach provides a less cryptographic heavy communication protocol but provides no additional security such as encryption. This technique should thus be used to exchange non critical information. Further the securing the communication would be the logical use of SSL/TLS in order to encrypt the data exchanged. This would be a more computational heavy phase then just using standard hash algorithms to verify communication but would provide better security and should be considered a trade-off.

The discovery phase would involve the embedded system utilizing DNS in order to look up the middle-wares IP-address. This step would more or less be needed since the services on the cloud can rapidly change. By using the proposed way of authentication we can have the two party's exchange authentication messages after the embedded system has conducted a successful lookup to further validate the middle-ware without the usage of DNSSEC. This approach is closely linked to the fact that the lwIP stack has already been ported to the Midrange platform and includes DNS capabilities.

The issue of the instantly changeable nature of a cloud service has already been brought up when dealing with the discovery process but more can be said. This elastic nature has already been counteracted by the cloud providers themselves by providing such products as load-balancers to distribute the load between instances running applications. These load-balancers can be heavily leveraged when dealing with user clients connecting to the middle-ware to distribute load. To a lesser extent the embedded systems themselves would use this functionality in the discovery phase and communication initiated by the embedded systems themselves.

When storing data and parameters at a cloud provider additional steps such as hashing and encryption should be done. This is very central for any cloud service in order to be sure that information leaks can not result in any damage. Having the middle-ware handling communication with different kinds of databases to store data and to be able to define cache parameters in order to limit the amount of requests heading to the embedded systems. The different kinds of databases provide a way to differentiate the data such as storing user data in relational databases distributed to regions where the users are or utilize non-relational databases as a store for log events.

In order to circumvent the flexibility of the cloud a stateless approach and a solution of using a detached client for access to the service creates itself flexibility. The solution of relying on a REST protocol for communication is a simple approach of facilitating communication between a client and the stateless middle-ware. We would then not have to worry about which middle-ware the client got its response from if the situation called for more to be deployed. By having a defined set of applications needed for the middle-ware instances to run (such as web-servers and library's) we have a fixed set of packages needed

for deployment. This fact is useful when new instances should be deployed and configured creating a scenario which can be automatized with the use of shell scripts or the use of images of instances being deployed.

The multitude of different services, APIs, and tools provided by cloud vendors should draw some concern. Any common grounds of interaction was not visible creating an environment where vendor lock-in is a fact. This creates a scenario where if a switch of provider might cause a heavy re-write phase of applications. Standard solutions such as using cloud deployed instances of for example databases instead of using the one provided is a solution counteracting this fact but then maintenance and concerns such as storage space must be considered. When this kind of functionality is offered by the providers maintenance free and with unlimited storage one should take the migration fact into count.

Chapter 8

Conclusions

8.1 Conclusion

This chapter will draw conclusion on the work presented and suggest future work relating to the thesis project.

8.1.1 Goals

The goal of this thesis was to explore how the offerings on different cloud providers could be used to exploit the possibilities of embedded systems to carry out different tasks for users over the Internet. In order to conceptualize this scenario the communication between the different parties involved (the users, software running in the cloud, and the embedded systems) was explored based on the capabilities of the embedded system. An approach of using a REST based communication protocol both on both ends was suggested in order to expose functionality. Later the approach of using middle-ware for interaction between the parties, rather than using an isolated web-client was proposed in order to exploit and counteract the elastic nature of the cloud. The development of such a middle-ware and the technologies it could use was explored by examination of the services provided by three cloud providers in order to get a picture of the current state of the offers provided by cloud vendors. The proposed solution takes into account the limitations of the embedded platform while also addressing the new opportunities to use a cloud deployed solution to enable new functionality in a cost effective manner. By having rapidly deployable services to handle customers and scenarios we gain much when dealing with setup time and infrastructure investment.

8.1.2 Insights and suggestions for further work

The approach of using cloud services to create an abstraction of the embedded systems by using cloud deployed middle-ware to handle authentication and interactions provide additional features for the developers to use. Instead of letting the embedded systems being directly used by the end users authenticated interaction take place and logging of such interaction can take place in order to create a more secure access scenario for users accessing the embedded systems.

Utilizing the stateless approach to communication we achieve an expendable solution in the terms of different kinds of clients and introduce a central point where an API can be created in order to define communication for a variety of different types of clients. Defining such an API with the desired functionality should be done by using available techniques such as WSDL in order to create an environment where rapid prototyping is available.

A positive side-effect of standardization of a REST based protocol used for both back-end and front-end communication is the possibility to decouple the embedded systems in the scenario and be used without the middle-ware. This can be achieved by creating limited clients for example a simple HTML page with Javascript or shells scripts executing a series of HTTP GET operations to be used on for example internal networks. We also expose the functionality of other programs so that they can fetch information from the middle-ware by for example implementing rich site summary (RSS) functionality and enabling third-party programs to be used for checking for example on an embedded system's status and other operations.

8.2 Future work

This section will present areas where future work could and should be conducted.

8.2.1 What has been left undone?

The usage of the API in the embedded system itself needs to be realized and implemented in order to be able to fully utilize the scenario of using a cloud deployed middle-ware to interface with the embedded systems. The suggested APIs complexity in terms of the encryption algorithms should be evaluated to be able to see what the effect would be on real world usage. In order to realize the communication a real implementation scenario would be needed where constraints such as the availability of computing power on the embedded systems on given times and the amount of storage space available on the system when deployed with an application already executing on the system.

8.2.1.1 Cost analysis

The approach of paying for only what you use in a cloud based solution creates very interesting opportunities in terms of cost of a solution running and scaling not only in terms of the cost itself by having the deployed services closely customized for the intended deployment. This pay-per-use model and the fact that no hardware is required creates a flexible approach toward utilizing the new services.

8.2.1.2 Security

The security of using in-house server compared to the off-site solutions provided by cloud vendors is a reality. We can use the widely used concept of dividing the security aspects into the areas of confidentiality, integrity and availability (CIA) identify and asses theses security risks. Problems arising in these three areas are not limited to cloud environments but to co-hosting in general. For example sensitive user data should be encrypted when stored in database to avoid it being readable if exposed. If not we are exposed to having data leaked inside and outside the cloud itself. By compartmentalization and layerization we can limit the exposing of data by using different access levels. The integrity of the data is thus also relevant having the possibility to restrict and view access logs. We also have limited information available about such leakage and we need the cloud provider, if the situation occurs, is up front about the fact that servers have been compromised and data could have been leaked. The last area, availability, is one benefit of the using the cloud to store data. Being able to distributed the data to different availability zones around the world provides additional safety if one data-center is unreachable.

The novel addition to the standard (CIA) areas is trust. Since a customer has no control of the underlying infrastructure the provider needs to be trusted to deliver a reliable service and to quickly respond to enquirer about availability. The customer must thus try to find a provider whom they believe to be trusted. This goes beyond just finding a provider which fits its needs and is harder since no standard way of comparing providers by the trust criteria is available.

8.2.2 Next obvious things to be done

Choosing an cloud provider is based not only on the technologies examined in this thesis but general things such as the location of the provider's data center. One advantage of AWS is the fact that an edge location is situated in Stockholm Sweden. Such factors might be considered when choosing a cloud provider to use. The next obvious thing is to decide which provider to choose and develop on.

8.3 Required reflections

This project presented and evaluated the deployment of a solution to bridge use of embedded systems by users via a cloud deployed service. This creates an environment in which the end users will be able to interact with embedded systems via a fault tolerant solution creating new capabilities for interaction. These new features enable previously unavailable scenarios in both the developers and the users use of the embedded system by facilitating interaction. Making data from the embedded systems more available can have great impact on their usage. For example, being able to remotely ask an embedded system in a card to provided a detail error code for maintenance without having to drive the car to the repair shop.

The economical benefits of the solution are both present in the deployed service and the embedded systems. Not having to support a local infrastructure of servers or having underutilized servers idling and waiting to be used reduces both capital and operating expenses. Avoiding the need to physically go to each embedded system to collect data is a economical benefit. The economical benefits are closely linked to the environmental benefits where no new hardware needs to be acquired to house the service and avoiding idling servers reduces the amount of electrical power waiting to be utilized.

Bibliography

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[2] Sun Microsystems. Introduction to cloud computing, white paper. 2009.

[3] McKinsey & Company. Clearing the air on cloud computing. Technical report, 2008.

[4] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf, 2009, National Institute of Standards and Technology, Information Technology Laboratory.

[5] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. Security in embedded systems: Design challenges. *ACM Trans. Embed. Comput. Syst.*, 3(3):461–491, August 2004.

[6] Mussie Tesfaye. Secure reprogramming of a network connected device : Securing programmable logic controllers. Master's thesis, KTH, Communication Systems, CoS, 2012.

[7] Guinard Dominique, Trifa Vlad, Pham Thomas, and Liechti Olivier. Towards physical mashups in the web of things. In *Proceedings of the 6th international conference on Networked sensing systems*, INSS'09, pages 196–199, Piscataway, NJ, USA, 2009. IEEE Press.

[8] Tao Lin, Hai Zhao, Jiyong Wang, Guangjie Han, and Jindong Wang. An embedded web server for equipments. In *ISPAN*, pages 345–350, 2004.

[9] Kevin Lee, David Murray, Danny Hughes, and Wouter Joosen. Extending sensor networks into the cloud using Amazon web services. In *NESEA'10*, pages 1–7, 2010.

[10] Syntronic AB. Midrange platform. Technical report, 2009.

[11] Victor Delgado. Exploring the limits of cloud computing, 2010. Masters thesis, KTH Royal Institute of Technology, School of Information and Communication Technology, TRITA-ICT-EX-2010:277, November 2010, http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-27002.

[12] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: shopping for a cloud made easy. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[13] T. Chan. Full interview: AT&T's Joe Weinman. Interview on GreenTelecomLive. http://www.greentelecomlive.com/2009/03/16/full-interview-att%E2%80%99s-joe-weinman/, 2009.

[14] Jinesh Varia. Architecting for the Cloud: Best Practices Whitepaper - Amazon. http://jineshvaria.s3.amazonaws.com/public/cloudbestpractices-jvaria.pdf, January 2010.

[15] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 2010. 10.1007/s13174-010-0007-6.

[16] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[17] G. von Laszewski, J. Diaz, Fugang Wang, and G.C. Fox. Comparison of multiple cloud frameworks. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 734 –741, june 2012.

[18] Qingwen Chen. Towards energy-aware vm scheduling in IaaS clouds through empirical studies. 2011.

[19] Javier Diaz, Gregor von Laszewski, Fugang Wang, and Geoffrey Fox. Abstract image management and universal image registration for cloud and hpc infrastructures. In Rong Chang, editor, *IEEE CLOUD*, pages 463–470. IEEE, 2012.

[20] Yanpei Chen, Vern Paxson, and Randy H. Katz. Whats new about cloud computing security? Technical Report UCB/EECS-2010-5, EECS Department, University of California, Berkeley, Jan 2010.

[21] Vytautas Zapolskas. Securing cloud storage service, June 2012. KTH Royal Institute of Technology, School of Information and Communication Technology (ICT), http://kth.diva-portal.org/smash/record.jsf?pid=diva2:538638.

[22] Andreas M. Nilsson. Realtidsoperativsystem på mätinstrument, 2007.

[23] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[24] Joakim Axe Löfgren. Remote software update on an arm cortex m3 platform, 2011. Masters thesis, KTH Royal Institute of Technology, School of Industrial Engineering and Management (ITM), Machine Design, MMK 2011:66 MDA 403, http://kth.diva-portal.org/smash/record.jsf?pid=diva2:541867.

[25] Jonas Lund. Utvärdering av lämpliga krypteringsmetoder för ett inbyggt system, 2011 (Ongoing).

[26] Jason H. Christensen. Using restful web-services and cloud computing to create next generation mobile applications. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 627–634, New York, NY, USA, 2009. ACM.

[27] J.O. Hamblen and G. M. E. Van Bekkum. Using a web 2.0 approach for embedded microcontroller systems. In *Frontiers in Education: Computer Science & Computer Engineering*, FECS 2011, pages pp. 277–281, Las Vegas, NV., July 2011.

[28] J.D. Case, M. Fedor, M.L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157 (Historic), May 1990.

[29] TIBCO. Tibco rendezvous. www.tibco.com/multimedia/ds-rendezvous_tcm8-826.pdf, 2008.

[30] Hewlett-Packard Development Company. HP intelligent management center. h17007.www1.hp.com/docs/mark/4AA3-4496ENW.pdf, 2011.

[31] D. Eastlake 3rd. Domain Name System Security Extensions. RFC 2535 (Proposed Standard), March 1999. Obsoleted by RFCs 4033, 4034, 4035, updated by RFCs 2931, 3007, 3008, 3090, 3226, 3445, 3597, 3655, 3658, 3755, 3757, 3845.

[32] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[33] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[34] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the weather tomorrow?: towards a benchmark for the cloud. In *Proceedings of the Second International Workshop on Testing Database Systems*, DBTest '09, pages 9:1–9:6, New York, NY, USA, 2009. ACM.

[35] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, October 2003.

[36] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[37] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 579–590, New York, NY, USA, 2010. ACM.

[38] David M. Kroenke. *Database Processing: Fundamentals, Design and Implementation.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 8th edition, 2001.

[39] Mongo. Mongodb howpublished = `http://www.mongodb.org/`, year = 2012.

[40] Jaroslav Pokorny. Nosql databases: a step to database scalability in web environment. In *Proceedings of the 13th International Conference on*

Information Integration and Web-based Applications and Services, iiWAS '11, pages 278–283, New York, NY, USA, 2011. ACM.

[41] V. Mateljan, D. Cisic, and D. Ogrizovic. Cloud database-as-a-service (DaaS) - ROI. In *MIPRO, 2010 Proceedings of the 33rd International Convention*, pages 1185 –1188, may 2010.

[42] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 251–264, New York, NY, USA, 2008. ACM.

[43] S. Kotecha, M. Bhise, and S. Chaudhary. Query translation for cloud databases. In *Engineering (NUiCONE), 2011 Nirma University International Conference on*, pages 1 –4, dec. 2011.

[44] Alexander Zahariev. Google app engine. In *TKK T-110.5190 Seminar on Internetworking*, pages 1 –5, http://www.cse.tkk.fi/en/publications/B/5/papers/1Zahariev_final.pdf, 2009.

[45] Google. What is google app engine? https://developers.google.com/appengine/docs/whatisgoogleappengine, 2012.

[46] Microsoft. Overview of windows azure traffic manager. http://msdn.microsoft.com/en-us/library/windowsazure/hh744833.aspx, 2012.

[47] Chanwit Kaewkasi and John R. Gurd. Groovy aop: a dynamic aop system for a jvm-based language. In *Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies*, SPLAT '08, pages 3:1–3:6, New York, NY, USA, 2008. ACM.

[48] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM.

[49] Amazon Web Services. Sample code & libraries. http://aws.amazon.com/code, 2012.

[50] Microsoft. Develop center. http://www.windowsazure.com/en-us/develop/overview/, 2012.

[51] Oracle Corporation. Glassfish howpublished = http://glassfish.java.net/, year = 2013.

[52] The Apache Software Foundation. Tomcat howpublished = http://tomcat.apache.org/, year = 2013.

[53] Eclipse Foundation.

TRITA-ICT-EX-2013:036

www.kth.se