Integrating an online booking service with access control systems

Rickard Lindahl

July 1, 2014
Master’s Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Jan-Erik Moström
Examiner: Fredrik Georgsson

UMEA UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN
Abstract

Fault tolerance is a property that describes systems that can execute properly even in the presence of errors. This report describes different methods and techniques to create fault-tolerant systems, as well as a study on the dependability of the cloud computing services offered by Amazon, Microsoft and Google. The research acted as a foundation when developing a system that integrates an online booking service for hotels and hostels with access control systems. The project resulted in two prototypes that together automates the creation and insertion of codes into an access control system in a fault-tolerant manner. The integration plays an important part in expanding the booking service’s market as facilities with access control systems do not have to manually insert codes into the access control system for every new booking.
Acknowledgments

I would like to thank Henri Toivonen for giving me the opportunity to do this project and for giving me advice and support during my work. I would also like to thank all nice people at c/o Huvudkontoret in Luleå for making me feel welcome and for helping me with the project in one way or another.

Special thanks go to Jan-Erik Moström for taking time to read my report and for giving valuable feedback.
# Contents

1 Introduction .................................................. 1  
1.1 Background .................................................. 2  
1.2 Thesis Outline ............................................... 3  

2 Problem Description ........................................... 5  
2.1 Problem Statement ........................................... 5  
2.2 Goals ....................................................... 6  

3 Fault Tolerance Methods and Techniques ....................... 7  
3.1 Introduction ................................................ 7  
3.2 Dependability ............................................... 8  
  3.2.1 Attributes ............................................... 8  
  3.2.2 Weaknesses .............................................. 9  
  3.2.3 Approaches .............................................. 10  
3.3 Redundancy ............................................... 11  
  3.3.1 Hardware redundancy ................................... 12  
  3.3.2 Software redundancy ................................... 15  
  3.3.3 Information redundancy ................................ 16  
  3.3.4 Time redundancy ....................................... 16  
3.4 Software Techniques ....................................... 16  
3.5 Design and Evaluation ..................................... 17  
3.6 Conclusions ............................................... 18  

4 Dependability of the Cloud .................................. 19  
4.1 Introduction to Cloud Computing ........................... 19  
4.2 Security and Dependability ................................ 20  
4.3 Comparison of Cloud Service Providers .................. 21  
  4.3.1 Conclusions ............................................ 23  

5 Working method ............................................... 25  
5.1 Preliminaries ............................................... 25  
5.2 How the work was done .................................... 25
# Contents

6 Results ................................................................. 27  
6.1 System Overview .............................................. 27  
6.2 BAPSI client .................................................. 28  
6.3 Middleware .................................................... 30  
  6.3.1 Storage .................................................. 30  
  6.3.2 API ...................................................... 31  

7 Conclusions ....................................................... 35  
7.1 Goals .......................................................... 35  
7.2 Restrictions .................................................. 35  
7.3 Limitations ................................................... 35  
7.4 Future Work .................................................. 37  

References ............................................................. 39  

A Bewator Entro Administration Details ...................... 45
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An explanatory view of the Dashboard in HappyBooking.</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>The system stack.</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>The failure rate of components as a function of time.</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>Triple Modular Redundancy with a single voter. Each module gets the same input and the output is sent to the voter which potentially filters out faults.</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>Hot standby sparing with $n$ modules and $n - 1$ modules acting as spares.</td>
<td>14</td>
</tr>
<tr>
<td>3.5</td>
<td>Self-purging redundancy with $n$ modules.</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>The separation of responsibility in each cloud service model.</td>
<td>20</td>
</tr>
<tr>
<td>6.1</td>
<td>The locations of the applications in the system.</td>
<td>27</td>
</tr>
<tr>
<td>6.2</td>
<td>An overview of the system and its parts. HappyBooking does an HTTP POST with new bookings to the middleware, which generates codes for each booking and maps them to the correct room in the Bewator Entro system. This booking is stored in an Azure Table Storage, a NoSQL database. The BAPSI client does an HTTP GET every X minutes and asks for new bookings. If there are new bookings they are converted into a BAPSI command and sent to the Bewator Entro Server.</td>
<td>28</td>
</tr>
<tr>
<td>6.3</td>
<td>A basic flow chart of the BAPSI client. A timestamp representing the last time the HTTP client checked for new bookings is read from persistent storage. This timestamp along with a facility id is sent as an HTTP GET request to the middleware. The middleware responds with bookings that were stored or updated later than the given timestamp or an empty array. New bookings are made into BAPSI commands and sent to the Bewator Entro server. Lastly the timestamp representing the time of the HTTP request is stored in the database. The program sleeps X minutes before repeating the process.</td>
<td>29</td>
</tr>
<tr>
<td>6.4</td>
<td>The client request containing the hash and the encrypted data in the request message body.</td>
<td>33</td>
</tr>
</tbody>
</table>
List of Tables

4.1 SDKs available in Amazon Web Services, Microsoft Azure and Google Cloud Platform. .................................................. 23
4.2 Guaranteed Monthly Uptime Percentages. ........................................ 24
4.3 Availability percentages and the corresponding downtimes. ................. 24

7.1 The BAPSI client goals together with a motivation as to why it was reached or not. ......................................................... 36
7.2 The goals that were set for the middleware together with a motivation as to why it was reached or not. .................................. 36
7.3 The general goals that were set together with a motivation as to why it was reached or not. ................................................. 37

A.1 Bewator Entro access methods to doors in the system ......................... 45
A.2 Example of BAPSI Data Packet classes and types .............................. 46
A.3 BAPSI Data Packet Structure .................................................. 46
A.4 The RC4 Encryption Key Structure ........................................... 47
Chapter 1

Introduction

The goal of this thesis is to integrate an online booking service for hotels and hostels with access control system in a fault-tolerant manner. This is done by researching about different methods and techniques to design fault-tolerant applications. Additionally the dependability of cloud computing services offered by Amazon, Microsoft and Google is looked at and compared to determine if the cloud is suitable to host the system.

The expectations on software applications have become so big today that if an innovative application suddenly is unavailable people might lose faith in it and find comfort in a competitor’s solutions. Every application is expected to work and design flaws in systems could ruin a business. Critical applications like those used in health-care or in unmanned space shuttles have even bigger demands on design and fault tolerance because any errors could be extremely expensive or even result in death. These systems must be designed so that any potential fault that could prevent the system from executing as intended is taken into account. The problems could be caused by natural catastrophes, human errors from using the system or design flaws in both hardware or software. How can one go about to design and implement systems that takes these factors into consideration and thus make it fault-tolerant?

With the emergence of cloud storage services such as Dropbox, Microsoft OneDrive and Google Drive people have become used to being able to access all their data from every corner of the world. Startup companies with limited budgets and companies with higher turnover are moving their business to the cloud using services offered by providers such as Amazon Web Services, Microsoft Azure and Google Cloud Platform to mention a few. These cloud computing services have become so affordable and easy to develop for that they are an interesting topic for every company with an IT unit. Although there are many advantages of choosing a cloud computing service, rather than hosting servers privately or perhaps even building a private data center, there are some risks in doing so. What protection and methods do the cloud computing service providers offer to make sure that data is not lost or that applications are available in case of failure?

The study was conducted in Luleå at meramedia, an agency creating digital and physical interaction through technology, events and design [1].
1.1 Background

This section covers background information concerning this thesis project.

**meramedia** is an agency creating digital and physical interaction through technology, events and design. The company was started in Luleå in 1998 by two people who were really passionate about web development. Since then it has grown to eight people working with development and events. The company has developed an in-house application called HappyBooking which is used to manage bookings at hotels and hostels [2].

**HappyBooking** is an internet based application, meaning that it is accessible from anywhere via a browser. It is an administrative tool to manage bookings at hotels, hostels or any other organisation or company that needs to keep track of objects that can be booked. The goal of HappyBooking is to be easy to use and affordable. These two attributes makes it perfect for a wide market ranging from small hostels to large hotels [3]. A screenshot from the HappyBooking website showing its online tour can be seen in Figure 1.1.

![Figure 1.1: An explanatory view of the Dashboard in HappyBooking.](image-url)
Citysleep is a hostel that is located in the center of Luleå, next door to meramedia. It is the first facility to use HappyBooking. At Citysleep they are using an access control system called Bewator Entro to manage access to doors in the building. The hostel is owned by the CEO of meramedia, a major reason why this project became a reality.

Bewator Entro is a secure and flexible access control system with focus on usability. The system is used to administer and manage access to doors in buildings where the access should be limited to some extent. Administration is done with a program in a Windows environment where it is possible to manage who gets access to specific doors in the facility. It is also easy to block user access to specific doors if the situation requires it, like if a user loses the access card or forgets the code. The launch of Bewator Entro in 1997 was an instant success and today it is one of the largest provider of secure access control systems. The system can consist of up to 512 doors and 40000 cards and works with many different methods of door access like card readers or keypads [4].

Bewator All Purpose Socket Interface (BAPSI) is an Application Programming Interface (API) that enables users to integrate external applications with Bewator Entro. Examples of external applications could be booking applications, time attendance applications or joint card administration applications.

1.2 Thesis Outline

Chapter 1: Introduction - Contains an introduction to the reason behind the thesis and some background information that is relevant to the problem.

Chapter 2: Problem Description - Gives a detailed description about the problem to be solved through research and prototyping throughout the project.

Chapter 3: Fault Tolerance Methods and Techniques - In this chapter we look at methods, techniques and what to think about when designing and implementing a fault-tolerant system.

Chapter 4: Dependability of the Cloud - How sure can one be that data is not lost in the cloud? A little background of cloud computing, cloud dependability and the disaster recovery plans offered by the providers.

Chapter 5: Working Method - Here you can read about how the project was planned and how the work was carried out.

Chapter 6: Results - This chapter contains more technical details of the prototypes and images showing how the system could work.

Chapter 7: Conclusions - In this chapter the limitations and problems are discussed and whether the initial goals were met is examined.
Chapter 2

Problem Description

This chapter contains a detailed explanation of the problem. This includes the reason behind the project, the parts included in the project and the goals that were set.

2.1 Problem Statement

As HappyBooking, the in-house application developed by meramedia, was starting to take off they saw a potential in expanding its market by integrating the booking service with access control systems such as Bewator Entro used at the hostel Citysleep, a facility in Luleå using HappyBooking. Today a receptionist at Citysleep has to manually update codes for every new booking that is made. The codes are updated using the Bewator Entro administration application. Integrating HappyBooking with access control systems, Bewator Entro to begin with, would make it possible for facilities using control access systems to run its business without needing a receptionist.

The implementation part of the project can be divided into two tasks. The first part is to write a BAPSI client that can communicate with the Bewator Entro server located at Citysleep. The BAPSI client must also somehow get the information about new bookings and map the room id from HappyBooking to the corresponding door in Bewator Entro before inserting or updating new codes into the access control system. A more detailed explanation of how this is done can be found in Appendix A.

The second part is to write a prototype of a middleware. The idea is to have HappyBooking forward all new or updated bookings made to facilities using access control systems to the middleware. The middleware should in turn generate codes associated to each booking. The code along with other relevant information about the booking should then be sent to the BAPSI client. An important factor is that the code should only be valid during the time of the booking.

The biggest problem to solve is that the information about bookings must always reach the access control systems. Otherwise, there is a possibility that the guests will not be able to enter their rooms during the booked time. This results in a negative experience for the guests and because dissatisfaction can spread quickly the reputation of the hotel or hostel might get tarnished. To minimize the risk of losing information the system must be designed and implemented in a fault-tolerant manner. The possibility, advantages and risks of hosting the system in the cloud should also be researched.
2.2 Goals

The goals that were set for the project can be summarized into the following items:

**BAPSI client**
- Receive bookings from the middleware via a secure protocol.
- Map bookings to BAPSI commands.
- Communicate with the Bewator Entro server according to BAPSI specification.
- Find suitable off-the-shelf hardware for private hosting.

**Middleware**
- Receive bookings from HappyBooking securely.
- Secure communication with the BAPSI client.
- Generate unique codes associated to bookings.
- Map booking details to access control system.
- Research the suitability of hosting the middleware on a public cloud service.

**General**
- Identify potential sources of faults and suitable methods and techniques to implement fault tolerance.

The information that is found during the research and the implemented prototypes should serve as a proof of concept for future work.
Chapter 3

Fault Tolerance Methods and Techniques

This chapter discusses the concept of fault tolerance and some methods and techniques used to achieve fault tolerance. Fault tolerance is a broad concept and the techniques and practices that have been developed to achieve fault tolerance in systems are many. To narrow it down the focus will be on dependability and four types of redundancy methods including hardware, software, information and time. The chapter starts out with an introduction to fault tolerance including its definition, why it is important and common types of faults before moving on to redundancy methods and techniques. The last section consists of conclusions on what types of applications the different methods and techniques could be suitable for.

3.1 Introduction

This section presents the definition of fault tolerance, different types of faults and examples of systems requiring fault tolerance.

Fault tolerance is a concept that was formulated by Algirdas Avizienis, Professor Emeritus of Computer Science UCLA. In the book Fault-Tolerant Design, written in 1967, Avizienis wrote:

"A system is fault-tolerant if its programs can be properly executed despite the occurrence of logic faults." [5]

Since 1967 both hardware and software have become increasingly complex, for instance, CPU architectures like ARM and x86 which have gotten support for more instructions over the years. Along with this evolving technology the number of fault sources has also increased. In the early days of computing faults mainly occurred from logic errors but in modern days the applications and operating systems are complex and it is nearly impossible to develop big systems without introducing bugs. About 50% of development and maintenance costs are spent on finding and fixing bugs and about 7% of bug repairs introduces new bugs [6]. Because of the complexity of some systems it is also a challenge for the developers to make applications that are easy to use and understand. Additionally the systems should be designed to be able to handle malicious actions.

Fault tolerance is important in systems that require high dependability. These systems could be safety-critical applications used in, for example, health-care where fatalities must
be avoided, mission-critical applications such as in airplanes and spacecrafts or business-critical applications like in bank and automated trading systems [7].

3.2 Dependability

This section sheds light on dependability of fault-tolerant systems, its attributes, weaknesses and common methods used to calculate dependability. Dependability is defined as the ability of a system to deliver its intended level of service to the users [7].

3.2.1 Attributes

There are three primary attributes of dependability that will be explained, reliability, availability and safety. These attributes are useful to a different degree to evaluate how systems behave [8]. In systems requiring non-stop delivery of its services like pacemakers or systems where a computer is controlling physical objects like engines, reliability is the most important attribute. Less critical applications, like e-commerce websites which do not require 100% uptime, availability is an important measure to find the proportion of time a system is able to deliver its services. Safety is the most important attribute in systems requiring to perform its services correctly or to shut down in a safe manner, like nuclear power plants [7].

Reliability can be written as a function of time, \( R(t) \). This function describes the probability that the system is working correctly in the time interval \([0, t]\), under the assumption that it was performing correctly at time 0 [7].

Reliability is closely related to Mean Time To Failure (MTTF) and Mean Time Between Failures (MTBF) which describe the mean time between two subsequent failures. The difference between MTTF and MTBF is the time that is required to repair the system after the first failure, Mean Time To Repair (MTTR), giving the following equation [8].

\[
MTBF = MTTF + MTTR
\]

Availability \( A(t) \), is an average fraction over time that the system is up on the interval \([0, t]\). The long-term availability, which can be interpreted as the probability that the system will be up at some point in time, can be denoted as the following [8].

\[
A = \lim_{t \to \infty} A(t)
\]

The long-term availability is only a useful measurement if the system is self-repairing. It can also be calculated from the MTTF, MTBF and MTTR [8].

\[
A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}
\]

The probability that the system is up at the particular time \( t \), called Point Availability can be denoted as \( A_p(t) \). A system with low reliability can still have high availability, for instance a system with a downtime of one second once every hour. This would give an availability as follows [8].

\[
A = \frac{MTTF}{MTBF} = \frac{3599}{3600} = 0.99972
\]
3.2. Dependability

Safety is an attribute that can be considered as an extension of reliability. When measuring reliability all faults are considered the same but in safety the faults are partitioned into fail-safe and fail-unsafe. An example could be an alarm system which may fail to function even though there is a breach in security, being fail-unsafe, or it may trigger when it should not, being fail-safe. Safety $S(t)$ can be defined as the probability that the system will function correctly or discontinue in a fail-safe matter. It is often used in safety-critical applications [7].

When calculating reliability or availability there are different definitions of time to consider. Time could be defined as wall-clock time which is the human perception of time from the start to completion of a task, or central processing unit (CPU) time which is the active time used by the CPU and does not include waiting for input/output (I/O) operations.

3.2.2 Weaknesses

There are a two terms that are often used when discussing dependability weaknesses. Faults (or failures) could occur because of hardware defects or software bugs while errors are manifestations of faults. Both faults and errors can spread through the system, causing nearby components to fail as well [8].

Faults can be classified according to its duration.

**Transient fault** is a fault that is temporary and will normally be recovered from. An example of a transient fault is noise that could occur when talking on the phone [8].

**Intermittent fault** is also a temporary fault that sometimes occur but never really disappears, for instance a loose monitor cable that sometimes makes the image on the display flicker [8].

**Permanent fault** is a fault that forever puts the component out of play, e.g., a light bulb that burns out [8].

![Figure 3.1: The system stack.](image-url)
A fault-tolerant system should be able to tolerate one or more faults in the entire system stack, see Figure 3.1. Some common errors and how they could manifest in each level of the system stack are explained in the following list.

**Devices/Logic** - Soft errors occurring on *chip-level* where radioactive atoms in the chip’s material decay and release alpha particles into the chip, causing memory cells to change state - a transient *bit-flip*, or *system-level* where data being processed contains noise that the computer tries to interpret as a data bit. Permanent faults at the logic level could be, for example, *stuck-at-fault* where some gate or line could get stuck at a certain value (one or zero) [9].

**Architecture** - Errors include design defects and defects related to component wear out, for example, an Arithmetic Logic Unit (ALU) that stops working. Some units might experience errors like the cache line having a single bit-flip due to cosmic ray strikes.

**OS/VM** - Faults in the Operating System (OS) or Virtual Machine (VM) are often caused by errors in kernel or device drivers. These faults occur when the operating system detects fatal errors it cannot recover from and they often manifest in the form of a *kernel panic* on Unix-like systems or the infamous *blue screen of death* on Windows machines [10].

**Application** - Faults could be caused by concurrency bugs and memory corruption errors. Configuration files needed by the application could be corrupted, binary files could be mutated or programming errors could create race conditions or atomicity violations.

**User/Operator** - Faults caused by carelessness from using the incorrect commands or GUI actions are common. Other faults could be created by errors in configuration files because of the parameters being misunderstood.

Other sources of fault resulting in system failure could be poor specification, design errors in software, hardware failure or interference on the communication subsystem [7].

### 3.2.3 Approaches

There are many approaches, methods and techniques to use when designing and developing a dependable system.

**Fault prevention** is done by eliminating any identified faults before the system becomes operational. It is done in two stages, *avoidance* and *removal* [11].

1. **Avoidance** - In the avoidance stage attempts are made to limit the introduction of faults by using strict specification of requirements and design methods [11].

2. **Removal** - The removal stage includes verification, diagnosis correction of faults. This can be done via preventive or corrective maintenance. In preventive maintenance parts in the system are replaced or adjusted before failure occurs, as apart from corrective maintenance where parts are replaced after the failure occurred [11].

Fault prevention is not enough in some systems, like in unmanned spacecrafts, because it is inevitable that at some point in time hardware components fail. That is where fault tolerance comes in.
3.3 Redundancy

Fault tolerance is achieved when creating systems that recover from faults and thus continues to work even in the presence of faults. Recovery from a fault is a process that takes the system to a previously valid state or creates a new state that is valid [12]. There are different levels of fault tolerance depending on the application.

1. Full fault tolerance is achieved when a system continues to operate in the presence of faults, without significant loss of functionality or performance. This level of fault tolerance is required in safety-critical applications [13].

2. Graceful degradation is achieved when systems continue to operate in the presence of faults, with a partial loss of performance or degradation of functionality [13].

3. Fail-safe systems maintains its integrity while accepting a temporary halt in its operations [13].

Fault forecasting is done by doing qualitative or quantitative evaluations of the system behavior. Qualitative evaluations are done to figure out the exact causes that lead to a system failure while quantitative evaluations are done by measuring the probabilities that the system will live up to its intended level of service [14]. It is also possible to estimate system coverage by measuring the number of redundant success paths in the system [7].

Fault tolerance is mainly accomplished by using redundancy of some kind. In Section 3.3 some common methods of how to design and implement fault-tolerant systems by using hardware, software, information and time redundancy are explained.

3.3 Redundancy

Redundancy plays a vital part in the creation of dependable, fault-tolerant systems. There are four common types of redundancy explained in this section. You can read about hardware redundancy, software redundancy, information redundancy and time redundancy in Section 3.3.1, 3.3.2, 3.3.3 and 3.3.4 respectively.

Redundancy methods can be divided into two parts, fault masking or fault detection.

Fault masking is done by hiding faults that occur in components. It is often achieved by correcting errors or compensating for errors, such as in Error-correcting code (ECC) memory where single-bit errors are corrected, thus making the error invisible to the user. It is also used in Triple Modular Redundancy (TMR) systems that uses majority voting [7].

Fault detection is done with, for instance, acceptance tests where code has to pass a certain test to be able to continue. Another variant is comparison tests used for systems with duplicated components. Any failed test indicates the presence of a fault. These tests can only tell that a fault has occurred, not where it is located [7].

These parts are often accompanied by fault detection and fault location methods.

Fault location is used to locate exactly where faults occur. Software bugs in the form of logical errors can be difficult to find but syntax errors are often detected by modern Integrated development environments (IDE) [7].
Fault containment is done by isolating modules in the system with the aim to limit the fault from spreading throughout the system. It can be done practically by frequent fault detection and consistency checks between modules [7].

3.3.1 Hardware redundancy

The definition of hardware redundancy is to provide two or more physical instances of a hardware component. There are basically two approaches to take when creating fault-tolerant hardware systems. A bottom-up approach includes designing an infrastructure of autonomously fault-tolerant subsystems consisting of microprocessors, memories or sensors and creating a reconfigurable and externally supported recovery. The top-down approach, which is the prevailing practice that will be focused on in this chapter, is one where the system is built with existing subsystems bought "off-the-shelf" that may have little or no fault tolerance at all. This is also common because it is more efficient in the form of development time and time to set up hardware [14].

Hardware redundancy can be used in different forms depending on the available resources and the demands on the system. A common usage is to have the system partitioned into modules that are backed up by spares, creating a protective redundancy [13]. If a component fails the system must include some mechanic to detect this and automatically replace the faulty component [7].

The rate of failure of a hardware component depends on the age of the component, the amount of voltage or physical shocks it receives, the temperature at which it operates and the technology it uses. A conceptual tool to visualize the failure rate of a component can be seen in Figure 3.2. It is commonly described as a bathtub curve and shows how poorly functioning components have a great risk of failing in its early days before moving on to a constant failure rate until the component wears out in the end [7].

![Figure 3.2: The failure rate of components as a function of time.](image)

Some hardware components contain built in support for fault tolerance, e.g., ECC memory. ECC memories can detect and correct the most common kinds of data corruption. It has an extra chip that checks if the data was correctly read or written by the memory module. The memory corrects single-bit errors meaning that the data being read from memory always corresponds to the exact data that was written. A disadvantage of using ECC memories is that the performance might decrease 2 – 3% compared to non-ECC memories. Some non-ECC memories with parity support can detect errors but are unable to correct them [15].
There are three forms of hardware redundancy that will be mentioned in this section and they are passive, active and hybrid methods.

**Passive redundancy** is done as a mean to mask faults in the system, rather than detecting them. The main advantage of passive redundancy systems is that computations can continue even in the presence of faults [13].

**Triple Modular Redundancy (TMR)** is one of the most common form of passive redundancy. It is a structural redundancy where the system consists of three modules that execute the same functions. The output from these modules are connected to a so called voter that compares the outputs and removes any error. A TMR system fails when two or more modules produce a faulty output [7].

![Figure 3.3: Triple Modular Redundancy with a single voter. Each module gets the same input and the output is sent to the voter which potentially filters out faults.](image)

**N-modular Redundancy (NMR)** is an extension of TMR. The difference is that $n$ modules are used instead of three, making the system tolerate more faults [7].

A few different voting techniques exists in passive redundancy. The examples above with TMR and NMR are written under the assumption that the voter always produce a perfect result. In fact, if the voter fails the whole system fails, i.e. the voter is a so called single point of failure. One possible fix is to decentralize voting by having for instance three voters instead of one. Another variant of voters is to have a master-slave relationship where a failed voter is replaced by a spare. Voting can be achieved by both hardware and software [7].

**Active redundancy** is a method that works by detecting faults before performing the actions needed to recover to a valid state. There are three common methods of active redundancy [7].

**Duplication with comparison** is a method where two identical modules perform the same function and their result is compared using a comparator. If the results differ an error signal is generated. This method can only be used to detect errors, not fix them [7].

**Standby sparing** consists of using $n$ modules where only one modules is active and $n-1$ modules are on standby. Each module is connected to a fault detection unit which acts as a switch in case any errors occur in the active module. The switch then tells the next module to take over. The spare modules can be in hot standby where all modules are running and ready to replace the faulty model instantly. The alternative is cold standby where the spare modules are powered down until a faulty module must be replaced. A standby sparing system can tolerate $n-1$. 
When the $n$th module fails the fault will be detected but there are no modules to replace the faulty module with [7], see Figure 3.4.

**Pair-and-a-spare** combines the methods described above but two modules are run in parallel and their output it compared with a comparator. If any fault is detected the location of the fault will be detected and the faulty module replaced [7].

Active redundancy is useful in applications where it is okay with infrequent, temporary erroneous results the system can recover from.

**Hybrid redundancy** is a combination of passive and active redundancy. Hybrid redundancy methods use fault masking to hide faults. Additionally they utilize fault detection, fault location and fault recovery [7].

**Self-purging redundancy** uses $n$ identical modules with each output being connected to a switch. The output from the switch is forwarded to a voter that acts as a gate. The output of the voter is compared with the output of each individual module and the switch connected to faulty modules are turned off, purging the faulty modules from the voting process. This method is capable of masking $n - 2$ faults [7], see Figure 3.5.
N-modular redundancy with \( k \) spares works like self-purging redundancy but with \( k + n \) modules. Only \( n \) modules are actively providing input to the voter that will replace any faulty model with one of the \( k \) spares in case of faults [7].

Triplex-duplex redundancy is a combination of TMR and duplication with comparison. Six identical modules are grouped into three pairs that compute in parallel. The result from each pair is compared and sent to a switch. If the results are equal the switch will forward it to a voter, otherwise the pair will be purged from the voting process [7].

Hybrid redundancy offers the most fault tolerance and is also often the most expensive method as it requires more resources than the other methods.

### 3.3.2 Software redundancy

As with hardware there is an array of different methods and techniques to use to achieve fault tolerance in software. Most software faults are caused by programming errors, so called bugs, created by the developers. Because it could be expensive to develop fault-tolerant systems it must first be decided how to allocate the available resources - should the focus be on fault tolerance or fault-avoidance, error detection or error prevention? No matter the choice one should strive to detect errors and recover from them as early as possible [14]. The following is a list of some common techniques used when implementing fault-tolerant software.

N-version programming is an approach used to reduce the risks of creating bugs in critical systems. It uses static redundancy, for instance different developers writing several versions of the same program which are given the same input. The output from each version of the program goes through a voter, like in TMR systems. The difference is that the output from each program may not be exactly the same but it must fulfil some assessments so the voter can identify and remove the faulty ones [13].

Module partitioning is a more dynamic approach than N-version programming. The software is partitioned into modules where the output from each module has to pass some acceptance test. If it does not pass the acceptance test some redundant code block can be executed instead [13].

Parallel execution is similar to the N-version approach. One or more versions of the same program could run in parallel with the difference that some versions of the program have less advanced algorithms and compute with less precision. The programs are then less probable to produce faults [13].

Sequential execution is an alternative to parallel execution. The different versions of the program are executed sequentially on a single computer, making is significantly slower than parallel execution [13].

Most fault-tolerant systems today are complex and are implemented with some sort of persistence, i.e. the state of the system outlives the process that created it. This is often achieved by writing the state to non-volatile memory, e.g., a database on a Hard Disk Drive (HDD) or flash memory. If the state is not written to non-volatile memory it would only exist in Random Access Memory (RAM) which is cleared every time it loses power [16].
3.3.3 Information redundancy

Because every component has a risk of failing, even persistent storage like HDDs, some measures must be taken to avoid losing data. Information redundancy can also be used as a mean to verify the integrity of data. Information redundancy is mainly achieved with hardware but there are solutions that include both hardware and software.

**Parity bit** is a bit that is added to the end of binary code. The parity bit indicates whether there are an odd or even amount of bits with the value one in the binary code. It is used as the simplest form of error detecting code. Parity can be calculated via an XOR sum of the bits, resulting in a zero for even and a one for odd parity. Parity can only be used for error detection and cannot correct any errors [17].

**RAID** is short for Redundant Array of Independent/Inexpensive Disks and it is a common form of information redundancy. There are a few different variants of RAID. One technique is called mirroring. The same data is written simultaneously to several different physical disks meaning that all disks have to stop working for data to be lost. Another variant is to use one or more disks for parity bits [18].

**Checksum or Hash sum** is a method that is commonly used to verify the integrity of particular data or to check if it has been altered during transmission or storage. There are multiple different algorithms available but perhaps most notably are MD5 and SHA-128 or SHA-256 [19].

Other types of information redundancy could be linear codes like *Hamming code*, cyclic codes like *Cyclic redundancy check (CRC)*, unordered codes like *Berger codes* or arithmetic codes like *AN codes* [7].

3.3.4 Time redundancy

Time redundancy is a technique that is often used when the extra time needed to guarantee correctness is of less importance than extra hardware. Sequential execution, mentioned in Section 3.3.2, is a form of time redundancy. Time redundancy is achieved by re-computing and comparing the results with previous results. If re-computing is done twice with one module creating a faulty result the fault will be detected. By repeating the computation three or more times the fault can also be corrected. Time redundancy can also be used to detect if faults are transient or permanent. If faults disappear after the re-calculation it must be a transient fault and the hardware does not need to be replaced. Common forms of time redundancy include re-computing with **alternating logic, shifted operands, swapped operands** and **duplication with comparison** [16].

3.4 Software Techniques

Some project methodologies and development processes are better suitable for writing software applications where bugs in the source code need to be kept to a minimum. The following methods and processes could be useful when developing a fault-tolerant system.

**Project methodologies** have moved from classical approaches such as the waterfall model to more agile frameworks like *Extreme Programming (XP)* and *Scrum*. These agile frameworks enables the developers to work iteratively and be flexible so they can respond to changes quickly [20].
Development processes used in agile frameworks varies but common techniques with the purpose of reducing bugs are Test-driven development (TDD), Behavior-driven development (BDD) and Cleanroom software engineering (CSE). TDD and BDD are similar in the sense that both processes involve writing acceptance tests while CSE is a team-oriented, theory-based process for developing high-reliability software systems under statistical quality control [21]. Pair programming is also a widely used method that involves two programmers taking turns in coding and reviewing the code. It is a method that has been found to reduce the number of bugs in code by approximately 15% at the expense of more time spend developing [22].

Error handling and recovery is something that is very important when achieving fault tolerance. There are at least two different error recovery techniques that are commonly used.

Forward Error Recovery (FER) is a method where the system continues executing with some form of compensation for corrupted or missed data. FER is often achieved by error handling in programming languages [13]. In traditional programming languages like C it is common to use error codes in the return value as a means to error handling. Some languages have integrated support for error handling, for example, Java where a function can define which exceptions it can raise [13]. In Node.js it is also possible to throw exceptions but as it runs on an event-driven platform it is also common to provide callback functions as parameters to non-blocking functions. These callback functions are encouraged to follow the convention of having the first parameter as an error object or null if no error occurred [23].

Backward Error Recovery (BER) involves taking the system back to a previously valid state before re-executing the code block. The concept is applicable to all systems that have access to persistent storage. There is however no guarantee that the error will not persist when re-executing the code block, perhaps by having outdated information in the stored recovery file [7].

BER can be implemented by using recovery blocks with the following principle [13].

- Create a recovery point in the entrance to a block.
- Create an acceptance test at the end of the block.
- If the acceptance test fails, restore the system to the recovery point and execute an alternative module.

The choice of recovery method is application-dependent. It can however be difficult to implement BER in real-time applications because the process of recovering with BER could be time consuming.

3.5 Design and Evaluation

The system designers should always strive to detect and recover from errors as early as possible. Apart from the hardware and software aspects discussed in Section 3.3.1 and 3.3.2 respectively there are a couple of factors one could think about before starting the implementation.

Identify phases - The environment and operating conditions of the system should be identified. Perhaps the system should operate in extreme heat or in cold environments? These factors must be taken into consideration [14].
**Fault classes** - Try to identify the possible fault sources - what could go wrong? Identify internal and external factors that could take the system into a faulty state [14].

**External support** - Should the system be able to repair itself automatically or should it be possible for a human to interact with it if the system fails? Determine the possibilities of external support [14].

**Service modes** - What are the acceptable modes of service, i.e. the expectations of the system when it is running in full/reduced/degraded/emergency mode [14].

**Dependability** - Determine the demands on the system availability, reliability, maintainability and security [14].

The system should be evaluated and tested continuously during the entire development process to make sure that the requirements on the system are fulfilled [14].

### 3.6 Conclusions

It is expensive to implement systems in a fault-tolerant manner. The common denominator in all systems is *redundancy* which can be utilized in different ways depending on the requirements on the system. There are a lot of fault sources throughout the entire system stack that need to be tolerated but the problem domain does not end there as natural catastrophes and blackouts must also be considered.

Fault-tolerant systems have different requirements on the three *dependability* attributes *reliability*, *availability* and *safety* and the approaches to meet these requirements differ. The choice of methods and techniques therefore depends heavily on the application. *Passive* or *hybrid redundancy* are best suited for applications where momentary erroneous results are not acceptable, as fault masking enables the computations to continue even in the event of faults. Systems that need to be restored quickly should use *active redundancy* methods and extremely critical applications that require the highest possible reliability should use *hybrid redundancy* methods [7].

The amount of fault tolerance achieved is also dependent on the amount of resources available. *Pair programming* might be more expensive and take more time than normal but when developing critical applications every penny spent trying to reduce bugs might be worth it in the long run.
Chapter 4

Dependability of the Cloud

In this chapter we will look at cloud computing services provided by Amazon, Microsoft and Google. What is cloud computing and what protective measures do the providers offer to make sure that business data does not get lost in their clouds? What availability do they offer in their Service-level agreement (SLA)? The responsibility areas of the different cloud service models will be explained and some security issues regarding the models will be discussed.

4.1 Introduction to Cloud Computing

Cloud computing is something that has been available for a long time but has just recently started to take off as it has become affordable and more secure. Cloud computing is a model to enable on demand network access to configurable computing resources. These resources could be networks, storage, applications or services that require minimal management effort or service provider interaction [24]. There are at least three cloud service models, each with different operating areas and responsibilities, as seen in Figure 4.1. The following list contains a short introduction to the three services models.

Infrastructure as a Service (IaaS) is the most basic of the cloud service models. The users are provided with processing, storage, networks and more, where they can deploy and run arbitrary software. The software could be hosted on a physical or a Virtual Machine (VM). The users have control over operating systems, storage and their deployed applications, making it the most difficult model for the providers to implement secure environments [24].

Platform as a Service (PaaS) is a less configurable service where the users can deploy their applications developed in specific programming languages, libraries, services or tools that the provider supports [24].

Software as a Service (SaaS) is used for applications that are accessible through thin clients via the web browser or other interfaces. The users have even less control over the hosting environment in this service [24].

It is not without reasons that cloud computing has become a big thing. Companies hosting applications privately must buy their own hardware and hire personnel to maintain the servers which can be time-consuming and expensive [25]. Some of the main advantages of
hosting infrastructures, platforms or services in the cloud include the ability to pay for using computing resources on a short-term basis, making it easy and cost-efficient to deploy new services or applications. Load balancing is done automatically to make sure applications will not crash when network traffic suddenly increases. Scaling can also be done arbitrarily up or down on demand [26]. Another advantage is data redundancy which many providers offer similar solutions to, including locally or geographically redundant storage [27].

### 4.2 Security and Dependability

History has shown that cloud services do not come without its flaws. On June 15, 2004, Akamai Technologies were targeted for a Distributed Denial-of-Service (DDoS) attack which compromised the company’s Domain Name Service (DNS) and affected the availability of Google and Yahoo! for a time [28]. A lightning storm on June 29, 2012, brought down some services hosted on AWS in the eastern region of the United States. Instagram were one of the services affected by the downtime [29]. Malicious attacks together with infrastructure failures such as power outages are probably the largest threat for cloud service providers today.

Because the cloud service provider is to great extent responsible for managing a company’s business operations the fate of the company is in the hands of the providers. In a study measuring hidden costs of the cloud it was discovered that 43% of companies taking the poll had lost data in the cloud and had to recover from backups [30]. If providers keep losing data and companies feel the need to keep private backups, that could mean that even more resources are allocated on both cloud services and private hosting rather than hosting everything privately. Cloud service providers are now facing a tough challenge in gaining (or regaining) the trust of a large user base.

The IaaS model is by far the most difficult to keep secure, followed by the PaaS and the SaaS model, as the users have more control over the services. VMs accessible via the IaaS model exposes the system to new sources of attacks as the trusted computing base of virtual environments include not only the hardware and the hypervisor but also the management OS. The entire state of a VM can be saved to a file for migration and recovery. An infected VM can be inactive when the systems are cleaned up and infect other systems as it later becomes active [31].
Resource management is also a challenge for the cloud providers. It is a difficult thing to implement as it involves controllers keeping track of the global state of the complex system. As the external load and the state of individual resources changes rapidly the controllers must function with incomplete or approximate knowledge of the system state [31].

The PaaS model is often isolated from interaction with other applications on the server. One method that cloud providers isolate applications are via Linux Containers (LXC) in which the applications can only see and interact with its own files and processes. LXC is an operating system-level virtualization method for running multiple isolated Linux systems on a single control host. LXC uses namespace isolation so a user is never able to get full root access [32].

Security in the cloud depends heavily on the framework the developers use and how the applications are designed but there is almost always some form of isolation to protect the users from getting into trouble [33].

One method providers can use to increase their security is to deploy honeypots, decoy systems configured intentionally to be vulnerable to attacks. Honeypots allow researchers to for instance identify sources of attacks and to collect malware for reverse engineering. The results gathered from the honeypot attacks can then be used to implement an even more secure environment [34].

### 4.3 Comparison of Cloud Service Providers

In this section we look at the details in the SLAs of Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform. Why should users choose one provider over the other?

**Service-level agreement** (SLA) is a definition of performance measures, warranties, problem management and more that is offered by the service provider. It often includes technical definitions in terms of MTBF, MTTR, uptime percentages or other measurable details. It is an agreement between the service provider and the customer. Often the providers define potential compensation if the provider does not meet the SLA. The areas covered by SLAs include details about the hosting facilities, platforms, OS, applications and storage services [35].

SLAs are one of the major considerations when buying cloud computing services as it describes what one should expect from the service provider [36]. Apart from the SLA some important factors when choosing a provider might be what frameworks and tools are available.

**Amazon Web Services** offers storage, a wide selection of compute instances, caching, high performance databases and tools to manage these resources. They have Software Development Kits (SDK) to quickly develop applications for AWS in Java, Ruby, .NET, PHP, Python and Node.js [37]. It is also possible to manage AWS services via their Command Line Interface (CLI) tool [38].

AWS Elastic Compute Cloud (EC2) describes unavailability as when there is no external connectivity on the running instances [39]. AWS guarantees that services running in their cloud should have an availability, a monthly uptime percentage, of at least 99.95% during any monthly billing cycle. If AWS do not comply with the SLA the users are guaranteed service credit percentages which can be used as a sort of coupon in future billings greater than one dollar. To get these service credit percentages users are required to email details about any incidents to AWS support. Other services offered by AWS like Simple Storage Service (S3) and Relational Database Service (RDS)
have their own SLAs and their monthly uptime is guaranteed to at least 99.90% [40] and 99.95% [41] respectively.

AWS offers services and features that are essential for disaster recovery. As AWS is available in several regions the users can choose where to host their production environment as well as choose a disaster recovery site. For their storage services every object is redundantly stored on multiple devices across multiple facilities within a region. They also have support for data retention and archiving via versioning of files. It is possible to create point-in-time snapshots of data volumes used as both protection and as starting points for new volumes. Users have full control of scaling up or down processing power with completely controllable VMs [42].

Amazon Web Services is available in several geographical regions around the world. An example is North America which consists of the four regions US East (Northern Virginia), US West (Oregon), US West (Northern California) and AWS GovCloud (US), the last being an isolated service designed for US government agencies [43]. Each region contains a number of availability zones, i.e. a data center in different cities within the region. All these data centers are spread out across the globe so the users can choose a data center close to its business, lowering latency and increasing the network throughput [44].

**Microsoft Azure** is the cloud service provided by Microsoft. It was previously known as Windows Azure but changed its name in April 3, 2014 [45]. Azure is similar to AWS and Google Cloud Platform as to the services offered including storage, databases, processing power and more. They provide SDKs for .NET, Java, Node.js, PHP, Python, Ruby and CLI support for Windows, Mac and Linux [46].

The monthly uptime percentage on Azure is calculated by:

\[
\frac{\text{MaximumAvailableMinutes} - \text{Downtime}}{\text{MaximumAvailableMinutes}}
\]

They guarantee at least 99.90% uptime per service for all their services except for VMs where they guarantee 99.95% uptime. For their Storage services they guarantee even higher uptime with 99.99% [47].

In Azure all data is also stored redundantly to minimize the risk of data loss in case of unexpected events like application errors, hardware failure or data center shut down due to natural disasters [48].

Each SQL Database instance has three replicas on three different physical machines within a data center, one primary replica and two secondary replicas. A transaction to the database is not considered as committed until the write is completed on the primary replica and one secondary replica, a so called *quorum-based* commit scheme. If the hardware hosting the primary replica fails the data is at least accessible from two other sources [48].

Microsoft Azure offers tools one can use to restore data in case of, for instance, user based errors like accidentally dropping a database table. Furthermore all databases are backed up every five minutes as a safe-guard against simultaneous or catastrophic hardware and system failure. To protect from widespread loss of data center facilities caused by, e.g., natural disasters. They recommend their customers to use Geo-replication of their data and to create a backup and restore strategy in case any errors occur [48]. By using Geo-replication all data is backed up on a secondary data center in a different region on the same continent as the primary location [49].
4.3. Comparison of Cloud Service Providers

Google Cloud Platform consists of Google App Engine (GAE), their PaaS, and Google Compute Engine (GCE), their IaaS. The services offered are similar to the ones offered by AWS and Azure with Cloud Storage, SQL and No-SQL databases, VMs, processing power and more [50]. They provide SDKs for development in Java, Python, PHP and Go [51].

For services covered in both the App Engine and Compute Engine Google guarantees at least a monthly uptime percentage of 99.95% per service [52] [53].

To protect the consumers’ data Google provides physical and environmental security with redundant systems. For every critical component in the data center a primary and alternative power source, each with equal capacity, is provided. In case the primary electrical power source fails the alternative power source will provide power until the backup diesel engine generators can take over. In case of service interruption due to hardware failure or other catastrophes Google implements a disaster recovery program for all its data centers. The data is replicated and backed up to multiple systems within a data center and in some cases also replicated to multiple data centers. Google conducts regular testing of its disaster recovery plan by simulating disasters in a geographic location or region. Systems in that location are taken offline and transferred to fail-over locations and it is then verified that the systems can operate at that location during the test [54].

One feature on Google’s Cloud Platform is Persistent Disk Snapshotting which makes it easy to create a backup of disks one can move to other Google data centers or use to start up a new VM [55].

In order to deploy fault-tolerant applications that have high availability Google recommends deploying applications across multiple zones in a region. This helps protect against unexpected failures of components in a single zone [56].

4.3.1 Conclusions

All in all it seems like the three cloud computing service providers offers very similar services, frameworks, redundancy functionality and implements similar disaster recovery plans. Table 4.1 shows a summary of the available SDKs offer by each provider.

<table>
<thead>
<tr>
<th></th>
<th>Amazon</th>
<th>Azure</th>
<th>Google</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ruby</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>.NET</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>PHP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Python</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Node.js</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Go</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>

The guaranteed monthly uptime percentages differed a little as can be seen in Table 4.2. But what exactly do these numbers tell us? In Table 4.3 a calculation of the downtime per year, month and week is done [57]. When writing critical applications with requirements on high availability, do these guarantees on availability suffice? The highest guaranteed monthly uptime percentage with 99.99%, translated to a downtime of 1.01 minutes per
Table 4.2: Guaranteed Monthly Uptime Percentages.

<table>
<thead>
<tr>
<th></th>
<th>Amazon</th>
<th>Azure</th>
<th>Google</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>99.90%</td>
<td>99.99%</td>
<td>99.95%</td>
</tr>
<tr>
<td>Database</td>
<td>99.95%</td>
<td>99.90%</td>
<td>99.95%</td>
</tr>
<tr>
<td>VMs</td>
<td>99.95%</td>
<td>99.95%</td>
<td>99.95%</td>
</tr>
<tr>
<td>Applications</td>
<td>99.95%</td>
<td>99.90%</td>
<td>99.95%</td>
</tr>
</tbody>
</table>

week, is offered by Azure for its Storage services. The average monthly uptime percentage offered by all three providers is 99.95% for which the users of the cloud services should tolerate a weekly downtime of 5.04 minutes. Take note that uptime and availability is not the same thing as a system can be up but not available due to, for instance, a network outage [57].

See Table 4.3 for examples of availability percentages and its corresponding downtimes per year, month and week [57].

Table 4.3: Availability percentages and the corresponding downtimes.

<table>
<thead>
<tr>
<th>Availability %</th>
<th>Downtime / year</th>
<th>Downtime / month</th>
<th>Downtime / week</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.90%</td>
<td>8.76 hours</td>
<td>43.8 minutes</td>
<td>10.1 minutes</td>
</tr>
<tr>
<td>99.95%</td>
<td>4.38 hours</td>
<td>21.56 minutes</td>
<td>5.04 minutes</td>
</tr>
<tr>
<td>99.99%</td>
<td>52.56 minutes</td>
<td>4.32 minutes</td>
<td>1.01 minutes</td>
</tr>
<tr>
<td>99.999%</td>
<td>5.26 minutes</td>
<td>25.9 seconds</td>
<td>6.05 seconds</td>
</tr>
<tr>
<td>99.9999%</td>
<td>31.5 seconds</td>
<td>2.59 seconds</td>
<td>0.605 seconds</td>
</tr>
<tr>
<td>99.99999%</td>
<td>3.15 seconds</td>
<td>0.259 seconds</td>
<td>0.0605 seconds</td>
</tr>
</tbody>
</table>

The conclusion based on these numbers is that perhaps the best solution is to build a private cloud if even higher reliability, efficiency and availability is required. Health-care applications and similar will probably never tolerate these downtimes as unavailability may have fatal consequences. On the other hand if the system is made to tolerate a few minutes of unavailability each week the cloud is the perfect tool. It offers a cost-effective and efficient solution to get started with newly developed applications. The question on which provider to choose is subjective and ultimately the choice lies in the comfort one finds in the SLA and perhaps what frameworks and tools are available.
Chapter 5

Working method

In this chapter you can read about how the project was planned and how the different parts of the project were carried out.

5.1 Preliminaries

During the first few weeks of the project there were many unknown factors that needed to be researched and clarified, especially regarding Bewator Entro and writing external applications with BAPSI. No one at meramedia had any experience with either but they had all documentation needed to get started. These documents were studied thoroughly in order to get a grip of how the Bewator Entro system works and what functionality an external application actually could have.

Two BAPSI libraries were found early on. The first was included with the installation disc for the Bewator Entro administration program and came in the form of a .dll-file with documentation specifically for external applications in C++. The other was an open source library written in Python [58]. Meramedia wanted full control over the source code and wanted it to be written in a programming language they were familiar with. It was therefore decided that instead of using any of the existing BAPSI libraries the source code should be written in Node.js which uses an event-driven, non-blocking I/O model that runs on Chrome’s JavaScript platform [59].

The BAPSI client was decided to be hosted privately on a computer located at Citysleep, instead of the alternative to host it on a cloud service. The reason for this was that it would be easier to send a pre-installed computer that does not need configuration to other facilities using Bewator Entro rather than forcing customers to set up port forwarding in their routers in order to expose the Bewator Entro server to the internet.

The middleware was uncertain as to if it should be hosted in the cloud or on meramedia’s private servers.

Before starting to research, design or implement anything a few days were spent getting acquainted with Node.js and JavaScript in general.

5.2 How the work was done

The first thing that was done when starting the project was to discuss with meramedia what they wanted to achieve and how we should do it. After that a project plan was created,
including rough estimates of the time needed to complete each task. The project plan was revised about every other week to see if anything had to be changed.

To avoid getting stuck reading documents for weeks some prototyping was started early on. Most of the focus was on developing the prototype for the BAPSI client and understanding the Bewator Entro system.

The correctness of the code was verified using the JavaScript test framework *mocha* [60] together with the BDD style assertions provided by *should.js* [61].

Testing of the BAPSI client was done continuously at Citysleep. Unfortunately they did not have any test-system so the up-and-running Bewator Entro system was used.

As the research progressed the prototypes were redesigned, if needed, and implemented. In case of uncertainties on how to solve a problem they were fixed by looking online at similar problems. A few hours was also spent on the phone with Bewator Entro support trying to figure out BAPSI.

We had meetings at meramedia roughly every other week where the work was presented, questions brought up and feedback was given.

The documentation was done in parallel to the development of the prototypes but it was not a major priority until a few weeks from the end.
Chapter 6

Results

This chapter presents the prototypes that were developed during the project. Section 6.1 provides a general description of the prototypes, where they are hosted and how they communicate. A more detailed description of the BAPSI client can be found in Section 6.2 and the middleware is further explained in Section 6.3. The result is strictly a back-end system with the purpose to automate the creation and insertion of codes into the access control systems.

6.1 System Overview

This section provides an overview of the system and its parts. The system consists of two prototypes that were created during the project, the BAPSI client and the middleware. HappyBooking and the Bewator Entro server already existed before starting. The location of each prototype in the system can be seen in Figure 6.1.

![Figure 6.1: The locations of the applications in the system.](image)

The system works the following way:

- HappyBooking sends HTTP POSTs with new bookings to the middleware.
- The middleware generates codes and maps the bookings to the correct rooms in the Bewator Entro system. The result is stored in a database.
- The BAPSI client fetches new bookings from the middleware every X minutes. For every new booking a BAPSI command is created and sent to the Bewator Entro Server.
Figure 6.2: An overview of the system and its parts. HappyBooking does an HTTP POST with new bookings to the middleware, which generates codes for each booking and maps them to the correct room in the Bewator Entro system. This booking is stored in an Azure Table Storage, a NoSQL database. The BAPSI client does an HTTP GET every X minutes and asks for new bookings. If there are new bookings they are converted into a BAPSI command and sent to the Bewator Entro Server.

See Figure 6.2 for a graphical overview of the system and how they generally work and communicate.

6.2 BAPSI client

This section contains a more technical description of the BAPSI client. Some implementation details are left out intentionally on request by meramedia. Encryption and security details are left out for readability but is described in detail in Section 6.3.2.

The BAPSI client is responsible for fetching bookings from the middleware and communication with the Bewator Entro server located at Citysleep. The program consists of two modules, an HTTP client and a TCP socket client, and all code is written in Node.js.

Figure 6.3 shows a flow chart of the BAPSI client. The program starts by saving the current time in a variable called now, before reading a UNIX timestamp from persistent storage using a minimalistic JSON storage module. The timestamp read from storage represents the last time bookings were fetched from the middleware. The timestamp together with the facility id from HappyBooking is sent to the HTTP client, which uses them when sending an HTTP GET request to the middleware. The middleware responds with an empty array or an array of bookings that were added to the middleware database after the timestamp that was sent in the request. If the response contains any bookings these are converted into a BAPSI command depending on the action-property of the booking. The action can be Add, Update or Remove and translates to the two BAPSI commands Add or Update Card/Person or Remove BAPSI Card. These commands are then sent to the Bewator Entro Server using TCP sockets. When the TCP communication has completed the variable now is written to persistent storage, overwriting the value of the previously read
6.2. BAPSI client

timestamp. The program sleeps for X minutes before repeating the process indefinitely.

Figure 6.3: A basic flow chart of the BAPSI client. A timestamp representing the last time the HTTP client checked for new bookings is read from persistent storage. This timestamp along with a facility id is sent as an HTTP GET request to the middleware. The middleware responds with bookings that were stored or updated later than the given timestamp or an empty array. New bookings are made into BAPSI commands and sent to the Bewator Entro server. Lastly the timestamp representing the time of the HTTP request is stored in the database. The program sleeps X minutes before repeating the process.

As mentioned in Section 5.1 the BAPSI client was chosen to be hosted privately on a computer located on site at the customer, i.e. Citysleep. The hardware chosen to run the prototype was a Raspberry Pi which is an energy efficient, credit card sized computer without any spinning parts [62]. As long as the software running on the Raspberry Pi does not produce memory leaks or does not heavily utilize the SD-card the computer has a theoretically low probability of crashing. Nothing is impossible though so some fault tolerance must be prepared.

First of all it is not enough with only one Raspberry Pi running the BAPSI client. Based on the research done in Chapter 3 a suitable method for hardware redundancy could be standby sparing where the system is composed of a primary unit and one or more spares.
The fault detection unit could be implemented with connections or APIs the modules use to communicate with the primary module. If the primary unit does not respond, it can be replaced using some arbitrary method.

A Pi UPS could be used as a backup power source to each Raspberry Pi. The Pi UPS provides power via two regular AA batteries in case there is a blackout [63]. A software program called Forever could be used in case there are any errors in the prototype code or any other fault occurs that prevents the application from running. Forever works by monitoring the process and automatically restarting it in case of failure [64]. A backup internet connection, like a 3G router, could also be used to take over if the main internet connection fails.

Some potential errors were detected in the software of the BAPSI client. Node.js provides callbacks with an error object as the first parameter and this can be used to handle errors. The creation of BAPSI data packets requires a four byte random crypt to be generated and it could fail if there is a lack of entropy in the system. This is fixed by having a default value of the crypt in case the generation fails. Loading timestamps from the SD-card could cause an I/O error and if that occurs, the last used timestamp is available via RAM.

If the SD-card would get out of free space, the system could start behaving strangely. What exactly happens is unsure and might need further investigation. After three weeks of stress testing the systems, sending requests every other minute and receiving more than the normal amount of bookings, the log file was 21 MB indicating that there is about 1 MB of data being written each day. The SD-card is 8 GB and that means it would take approximately 22 years to run out of storage. It is therefore highly unlikely that it will happen in the normal case.

### 6.3 Middleware

Based on the research on cloud dependability, it is safe to say that the middleware is suitable to be hosted in the cloud as the application does not require 100% uptime. The dependability offered by the providers are good enough and the middleware was therefore designed for the cloud. The prototype was up and running on Microsoft Azure during the last weeks of the project. The middleware is the program responsible for generating codes and mapping rooms from HappyBooking to the correct one in the access control system. It basically consists of a Representational State Transfer (REST) API, hereafter referred to as API, with the standard create, read, update and delete (CRUD) functions, except deleting is never done externally. See Section 6.3.1 for information about the storage used and Section 6.3.2 for details on the API used by the middleware.

#### 6.3.1 Storage

Some kind of persistent storage was needed to achieve the wanted functionality, as described in Section 2.2. Because Windows Azure was chosen to host the prototype, on request by meramedia, the alternatives for persistent storage were to use Azure SQL Database or Azure’s non-relational data storage including Blob, Table, Queue and Drive Storage [65]. The decision fell on Azure Table Storage because of its simplicity. Table Storage offers NoSQL capabilities that can be used to store large amounts of unstructured or semi-structured data [27]. This is simple because any entity can be stored, such as a JSON-object, without needing a scheme telling it how to be structured. This makes it easy to add support for more access control systems in the future. The only requirement is that each entity added must include the following two properties [66].
6.3. Middleware

**PartitionKey** - Used to support load balancing across storage nodes. It is a unique identifier for the partition within a given table and is required for every insert, update and delete operation [66].

**RowKey** - A unique identifier for an entity within a given partition. The RowKey together with the PartitionKey uniquely identifies every entity within a table [66].

A third property, **Timestamp**, representing the time an entity was last modified is automatically added on inserts and updates [66]. This property is used by the middleware to sort out new bookings on HTTP GET requests made by the BAPSI client, which is further explained in Section 6.3.2. The storage used by the middleware should not be a redundant storage to what already exists in the HappyBooking database but merely a complement to achieve what is wanted of the system.

The following four tables were created in Azure Table Storage:

**Booking** - Information about bookings consisting of the code, the time the booking is valid, the Bewator Entro access group name (which could be anything else if using another access control system), and an action which can hold the value Add, Update or Remove, as described in Section 6.2. The PartitionKey is the facility id and the RowKey could be the booking id from HappyBooking.

**Facility** - Basic information about a facility including the access control system used and, for example, how many numbers there should be in the code. This table also contains a private key used to encrypt the request body when using the API. Every facility could share the PartitionKey to this table because the table will contain one entity per facility. The RowKey is the facility id from HappyBooking.

**ReservedCode** - Contains the names and codes that are reserved by, for instance, the cleaning staff. The PartitionKey is the facility id and the RowKey could be a Universally unique identifier (UUID).

**Room** - The table used to map a room id from HappyBooking to the corresponding room or access group name in Bewator Entro. The PartitionKey is the facility id and the RowKey is the room id from HappyBooking.

6.3.2 API

The middleware uses an API to manage data about bookings. The API can be divided into two parts according to its use cases. The first part is used only by HappyBooking to add new bookings or update existing ones, i.e. via the HTTP POST and HTTP PUT methods. The second part is used by facilities with access control system to fetch data about bookings, i.e. via the HTTP GET method.

**Adding or updating bookings**

The information needed from HappyBooking for each booking is the booking id, the room id and the actual time the booking is active. These ids are stored in the HappyBooking database.

When HappyBooking does an HTTP POST with new bookings the middleware has to generate unique codes and map the room ids to their corresponding room in the access control system. The mapping of rooms is easy and only requires one lookup in the Azure Table Storage. To generate a unique code three calls to the Azure Table Storage are required.
The middleware fetches all reserved codes followed by the codes used by active bookings and lastly the amount of digits to be generated. A random number is generated and compared with the reserved codes and codes used in active bookings. If the generated code is not unique a new number is randomly generated. The codes on bookings with the property `action` with the value `Remove` are not considered when generating new codes. All calls to the Azure Table Storage are done asynchronously and every new booking gets the property `action` set to the value `Add`.

When updating existing bookings, usually extending the time of a booking, the `action` property is modified to the value `Update`. The property `Timestamp` of the existing entity is automatically updated by Azure to reflect the time it was last modified.

Bookings which have expired will have their `action` set to `Remove`. This will trigger it to be removed from the Bewator Entro server next time the BAPSI client fetches bookings.

**Fetching bookings**

Facilities using access control systems will utilize the HTTP GET method to fetch new bookings. A call to the API could look like the following example.

```
GET <url> /bookings/facilityId?timestamp=1399372904000
```

The `facilityId` is the actual facility id from HappyBooking. All bookings associated with the given facility id will be fetched from the the `Booking` table in the middleware. The `timestamp` parameter is optional and if provided it will filter out any bookings older than the given value.

**Security**

It is obvious that the API could be easily violated without securing it properly, otherwise the codes associated to bookings would be available to anyone. Some form of security must therefore be implemented. To avoid unnecessarily complex solutions like e.g. OAuth or SSL a method similar to Signature 2 used in Amazon Web Services was chosen [67]. The method involves sending user credentials with every request and it is secure enough to make sure the middleware can verify that the request is actually sent from a trusted facility [68]. The middleware stores a private key for every HappyBooking facility using access control systems. The private key is used for encrypting and decrypting data when accessing the API. It works the following way.

**Client**

1. Combine the facility id and the current time and create a hash of them.
2. Encrypt the facility id and the current time.
3. Put the hash and the encrypted data in the HTTP message body and send the request to the middleware.

**Middleware**

1. When a request is received the middleware uses the private key associated with the facility id provided in the request line to decrypt the encrypted data provided in the HTTP message body.
2. The decrypted data is hashed and compared to the hash provided in the HTTP message body. If the hashes are equal the client is trusted and the request can be accepted.

3. The decrypted data contains a timestamp that is compared with the current time. If the time difference is not within an acceptable limit, e.g. a few minutes, there is a large risk that the request is a *replay attack* and it will not be authorized [69].

4. The bookings matching the request are hashed and encrypted with the private key and then sent to the client.

The private key is never sent with a request, it is only used locally when encrypting and decrypting the data. The response message body sent from the middleware follows the same principle as the client request message body, see Figure 6.4, but the data being hashed and encrypted consist of bookings instead.
Chapter 7

Conclusions

In this chapter the result is compared to the goals that were set before the project was started. Some restrictions and limitations of the implemented prototypes are discussed before bringing up some future work that could improve the prototypes.

7.1 Goals

The goals that were set together with meramedia were broad and focused on the desirable behavior of the prototypes. This was done because they had opinions on how the system should work but not how it should be implemented. The goals together with motivations are summarized Table 7.1, 7.2 and 7.3 for the BAPSI client, the middleware and general goals respectively.

7.2 Restrictions

There were some restrictions regarding what could be accomplished with external applications with BAPSI in general. BAPSI was really designed for Bewator Entro operating in a state using card readers. It is then possible to take advantage of Bewator Entro’s internal booking system. Unfortunately this does not work when the system is operating without cards, like at Citysleep.

The documentation of BAPSI was not accurate as it stated that the timestamp when adding new entities into the access control system should be on the form \(YYYYMMDDHH-MMSS\) when it in fact did not care about the hours, minutes or seconds. This is a restriction that makes it a security risk to send the access code to guests as soon they have booked because the code would be valid too soon.

7.3 Limitations

The BAPSI client prototype was implemented in a fashion that only supports sending BAPSI commands that are meant for Bewator Entro systems operating with personal code. That means it will not work in any facility using card readers. This was however not regarded as a problem and actually encouraged by meramedia.
Table 7.1: The BAPSI client goals together with a motivation as to why it was reached or not.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Motivation</th>
<th>Reached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive bookings from the middleware via a secure protocol.</td>
<td>The BAPSI client communicates with the middleware via a secure API where each request contains encrypted credentials.</td>
<td>Yes</td>
</tr>
<tr>
<td>Map bookings to BAPSI commands.</td>
<td>Each booking received from the middleware contains all information needed to add or remove bookings, including the actions <em>Add</em>, <em>Update</em> or <em>Remove</em> which is translated to BAPSI commands.</td>
<td>Yes</td>
</tr>
<tr>
<td>Communicate with the Bewator Entro server according to BAPSI specifi-</td>
<td>The implemented prototype can successfully communicate with the Bewator Entro server.</td>
<td>Yes</td>
</tr>
<tr>
<td>cation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Find suitable off-the-shelf hardware for private hosting.</td>
<td>A proof of concept using multiple Raspberry Pi computers together with backup hardware was used.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 7.2: The goals that were set for the middleware together with a motivation as to why it was reached or not.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Motivation</th>
<th>Reached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive bookings from HappyBooking securely.</td>
<td>HappyBooking can send bookings via a secure API where each post contains encrypted data and credentials.</td>
<td>Yes</td>
</tr>
<tr>
<td>Secure communication with the BAPSI client.</td>
<td>The API provides secure communication with the BAPSI client.</td>
<td>Yes</td>
</tr>
<tr>
<td>Generate unique codes associated to bookings.</td>
<td>Every new booking is assigned a unique code as they are added to the Data Storage.</td>
<td>Yes</td>
</tr>
<tr>
<td>Map booking details to access control system.</td>
<td>Every room id from HappyBooking is translated into an access group name in Bewator Entro. This functionality is implemented but the correct data was never added to the Table Storage during development so it has not been tested.</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Research the suitability of hosting the middleware on a public cloud service.</td>
<td>Three cloud service providers were studied and the cloud definitely offers enough dependability for the problem.</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 7.3: The general goals that were set together with a motivation as to why it was reached or not.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Motivation</th>
<th>Reached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify potential sources of faults and suitable methods and techniques to implement fault tolerance.</td>
<td>The research on fault tolerance methods and techniques together with the research on dependability of the cloud provides a general understanding on how the problems could be tackled. Examples on how this could be realized in practice are included in the proof of concept prototypes.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The communication between the BAPSI client and the Bewator Entro server was implemented using a synchronous Node.js module that unfortunately did not compile on Microsoft Windows. Perhaps it could be fixed but until then the prototype does not execute on Windows machines, making any future move to Microsoft Azure more difficult.

7.4 Future Work

There are a couple of things that weren’t implemented during the project and should be fixed before the system could be safely used in real life.

**Hardware redundancy** - The actual implementation of the hardware redundancy was never done and needs to be implemented before setting up the system for real.

**Fault detection** - The software detecting if the primary computer has stopped working must be implemented.

**Real data** - The mapping of rooms to access groups in Bewator Entro is implemented but the Table Storage contains mock data and needs to be updated with the correct data from HappyBooking.

**Removing old bookings** - Removing inactive bookings from the Table Storage could be implemented to improve the performance of the middleware in the long run.

**Notify guests with codes** - The guests needs to be able to receive the code to their rooms and this should be implemented in the middleware.

**Notify support** - In case of security breaches, attempts to attack the system or if anything goes wrong support should be notified.

If performance turns out to be a big problem the system flow could be changed a bit. Instead of having the BAPSI client fetching new bookings every X minutes the middleware could send new bookings to the BAPSI client as they come in. This would reduce the network traffic and stress on the middleware. It could be implemented using persistent connections like WebSockets or the BAPSI client could have an API available to the middleware. It is however possible that the API method would require port forwarding in the routers and that would defeat the purpose of having private hosting of the BAPSI client.
References


REFERENCES


Appendix A

Bewator Entro Administration Details

Administration Details

The Bewator application for the Windows environment is used to administer doors, zones, access groups and users. The application is required in order to set up the functionality of the system after it has been physically installed.

Doors

Each door in the system can be assigned with different properties. Each door can be assigned different methods of access, see Table A.1. It is also possible to choose different methods during three different time schedules, e.g. a door might be completely open during the day but require a card or a code at night. Doors can also be programmed to be bookable objects, e.g. a laundry room which can be booked by users in the system. These can be booked using the system web service.

<table>
<thead>
<tr>
<th>Access method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card</td>
<td>The user swipes an application card in the card reader by the door.</td>
</tr>
<tr>
<td>Card + PIN</td>
<td>The user swipes an application card in the card reader and then inputs the PIN code of the card at the keypad by the door.</td>
</tr>
<tr>
<td>Group Code</td>
<td>The user inputs the group code, of the corresponding access group, at the keypad by the door to gain access.</td>
</tr>
<tr>
<td>Personal Code</td>
<td>Each user has a unique code that is used by the keypad at the door.</td>
</tr>
<tr>
<td>RFID tag</td>
<td>The RFID tag is scanned by the RFID tag reader at the door.</td>
</tr>
</tbody>
</table>

Access groups

An access group is a set of doors that, when assigned to a user, describes the doors the user has access to. When creating an access group one can choose to give the group a password.
This password can be used to open doors that are set to be opened by the Group Code access method. A user may be assigned several access groups.

**Restrictions**

The system can be set to operate in two states: 1. Card, Card + PIN, Group Code or RFID tag. 2. Personal Code. When using state 2, Group Codes cease to work because the system cannot distinguish between a Personal Code and a Group Code. Another feature that does not work in state 2 is bookable objects. This is because bookable objects can only be set to be opened with the methods in state 1.

**BAPSI**

BAPSI, Bewator All Purpose Socket Interface, is a way to integrate external applications with the access control system. It is not exactly an API but more a structured definition of how data should be sent securely to the system. Examples of external applications could be booking applications, time attendance applications or a join card administration application.

**Data Packet**

The BAPSI documentation provides a number of data classes and types, see Table A.2 for examples, which are used to send commands to the Bewator System.

<table>
<thead>
<tr>
<th>Class</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Login</td>
<td>Login to the system</td>
</tr>
<tr>
<td>Application</td>
<td>Add card/person</td>
<td>Adds or updates a card or a person.</td>
</tr>
<tr>
<td>Read Only</td>
<td>Read Cards</td>
<td>Export of the cards and persons in the system</td>
</tr>
</tbody>
</table>

The structure of the data packet, see Table A.3, consists of two parts - a header and the payload.

<table>
<thead>
<tr>
<th>Header</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not encrypted</td>
<td>RC4 Encrypted</td>
</tr>
<tr>
<td>MD5 Hashed</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
<th>Crypt</th>
<th>Length of Data</th>
<th>MD5 Hash</th>
<th>Class</th>
<th>Type</th>
<th>Data</th>
</tr>
</thead>
</table>

The Header contains the Length of Data and a random Crypt that is appended to the 16 byte application key, which is known to both the sender and receiver, to create the RC4 encryption key.

The data class together with the data type and data is MD5 Hashed. Hashes are widely used as a method to verify that data has not been modified on its way from the sender to the receiver. The MD5 Hash together with the data class, data type and data is encrypted with the RC4 algorithm. The receiver reads the crypt from the header and concatenates it...
to the 16 byte application key to create the RC4 key used to decrypt the encrypted packet, see Table A.4.

Table A.4: The RC4 Encryption Key Structure

<table>
<thead>
<tr>
<th>RC4 Encryption Key (20 bytes)</th>
<th>Application Encryption Key (16 bytes)</th>
<th>Random Crypt Data (4 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>