Bachelor Thesis

# Truly Random

## Verifiable Entropy Accumulation Based on a Comprehensible, Random Phenomena

BSc in Computer Science

University of Applied Sciences,
Bern, Switzerland

June 11, 2015

**Authors**

Morandi, Matteo
matteoalain.morandi@students.bfh.ch

Rothen, Tobias
tobias.rothen@students.bfh.ch

**Tutor**

Dr. Koenig, Reto
reto.koenig@bfh.ch

**Expert**

Dr. Flueckiger, Federico
federico.flueckiger@gs-efd.admin.ch

## Abstract

Random numbers are a crucial part of cryptography. If they can be determined by an attacker, even the best encryption algorithms become vulnerable. Today there are many established random number generators, but at the real base of it all lies the problem of providing reliable entropy. Solving this problem by means of a True Random Number Generator (TRNG) is usually avoided because of costs and complexity of most TRNGs. But since randomness seems to be all around us, we would like to prove that entropy accumulation must not always be complicated and expensive. By creating a prototype that is providing entropy in a easily comprehensible way, we also offer the possibility to verify the whole process of how the random data is gathered. A property that is missing in nearly every hardware random number generator so far. The following document describes the conception and implementation of such a verifiable random number generator in the scope of our bachelor thesis at the Bern University of Applied Science.

## Versions

| Version | Date | Status | Remarks |
|---|---|---|---|
| 1.0 | 11.06.2015 | Final | Final Version |

# Contents

**1. Introduction** **7**
1.1. Definition of the Project . . . . . 8
1.2. Fundamental Concepts of Random Number Generation . . . . . 8
1.3. The Entropy Source Issue . . . . . 9
1.4. Current State of the Art . . . . . 10
1.5. Comprehensibility and Verifiability of Random Number Generators . . . . . 11

**2. Conception** **13**
2.1. Our Approach . . . . . 13
2.1.1. Preliminary Work . . . . . 14
2.2. Generator Concept . . . . . 14
2.2.1. Use Cases . . . . . 15
2.3. Server Concept . . . . . 15
2.3.1. Use Cases . . . . . 15
2.3.2. The Fortuna PRNG . . . . . 16
2.4. Transmission Concept . . . . . 18
2.4.1. Initialization and Authorization . . . . . 18
2.4.2. Transport of Random Data . . . . . 19
2.4.3. Timeout and Offline Mode . . . . . 21
2.5. Risk Assessment . . . . . 21
2.5.1. Generator Security . . . . . 21
2.5.2. Transmission Security . . . . . 22
2.5.3. Server Security . . . . . 23
2.5.4. Trust Assumptions . . . . . 24
2.6. Testing randomness . . . . . 25
2.6.1. Statistical randomness tests . . . . . 25
2.6.2. Human perception . . . . . 26
2.7. Requirements . . . . . 29
2.8. Goals . . . . . 29
2.9. Project Schedule . . . . . 33
2.10. Milestones . . . . . 34
2.10.1. Phase Planning . . . . . 34
2.10.2. Phase Implementation . . . . . 35
2.10.3. Phase Testing . . . . . 36
2.10.4. Phase Conclusion . . . . . 37

**3. Implementation** **38**
3.1. Main Concept . . . . . 38
3.2. Development Environment . . . . . 39
3.3. Prototype structure . . . . . 39
3.4. Electronics . . . . . 45

3.5. Software overview . . . 49
3.6. Prototype Application . . . 51
3.6.1. Main . . . 51
3.6.2. Graphical User Interface . . . 52
3.6.3. Logging . . . 53
3.6.4. Configuration . . . 53
3.6.5. System / Hardware Access . . . 54
3.6.6. Sound output and alarming . . . 55
3.6.7. Camera . . . 55
3.6.8. Random Data Control . . . 57
3.6.9. Testing tools . . . 58
3.6.10. Image Processing . . . 64
3.7. Server Application . . . 67
3.7.1. User interface . . . 67
3.7.2. Seeding Strategies . . . 67
3.8. Data Transmission . . . 69
3.8.1. Hardware Implementation . . . 70
3.8.2. Software Implementation . . . 70
3.9. Test Procedure (Challenge Response) . . . 72

**4. Conclusion** **74**
4.1. Progress and Time Management . . . 74
4.2. Open Issues . . . 75
4.3. Prospect . . . 76

**A. Transmission Flow** **80**
A.1. Server Startup . . . 80
A.2. Server Data Transport . . . 81
A.3. Generator Startup . . . 82
A.4. Generator Data Transport . . . 83

**B. Installation Manuals** **84**
B.1. (Installation Generator(Raspberry Pi) . . . 84
B.1.1. Setup . . . 84
B.1.2. Prerequisites . . . 84
B.1.3. Starting the Application . . . 86
B.2. Installation (Server) . . . 87
B.2.1. Setup . . . 87
B.2.2. Prerequisites . . . 87
B.2.3. Starting the Application . . . 88
B.2.4. Application Recovery . . . 88

# List of Figures


1. Basic setup of a camera based generator . . . 13
2. Verifiability of a camera based generator . . . 13
3. Intrusion detection due to verifiability . . . 14
4. Generator of Fortuna . . . 16
5. Fortuna overall structure . . . 17
6. Authentication setup . . . 19
7. Data transmission . . . 20
8. Corrupt data handling . . . 20
9. Comparison of two two-dimensional random walks . . . 27
10. Comparison of bitmaps between random.org and PHP rand() [17] . . . 28
11. Project schedule . . . 33
12. Main Structure . . . 38
13. Airbox . . . 41
14. Tube . . . 41
15. Tube connector . . . 42
16. Tube hat . . . 43
17. Resulting air box split in two parts . . . 44
18. Schematic diagramm of LEDs . . . 48
19. Interaction between applications . . . 50
20. Main class . . . 51
21. MVC UML . . . 52
22. Hardware control UML . . . 54
23. System settings view . . . 55
24. Camera UML . . . 56
25. Camera settings view . . . 57
26. DataControl UML . . . 57
27. Random walk UML . . . 58
28. Random walk view . . . 59
29. Sound analysis UML . . . 60
30. Sound analyze view . . . 61
31. QR code UML . . . 62
32. Random QR view . . . 63
33. QR code in transmission settings view . . . 63
34. Image processing strategy setup . . . 64
35. Average grey-scale grid strategy . . . 65
36. Image processing strategy setup . . . 65
37. Verbose visualization on the user interface . . . 66
38. Setup Strategy Pattern for Seeding . . . 68
39. Transmission states . . . 69
40. Setup of the USB cable . . . 70
41. Setup Transmission . . . 71

42. Insertion of a test module . . . . . 72
43. Using the Verbose view for the testing procedure . . . . . 73
44. Actual project schedule . . . . . 74
45. Network Flow Server Startup . . . . . 80
46. Network Flow Server Data Transport . . . . . 81
47. Network Flow Generator Startup . . . . . 82
48. Network Flow Generator Data Transport . . . . . 83

## List of Tables

1. Prototype Hardware Goals . . . . . 30
2. Prototype Software Goals . . . . . 31
3. Server Related Goals . . . . . 31
4. Data Transmission Goals . . . . . 32
5. Conceptional Goals . . . . . 32

# 1. Introduction

Random numbers are essential for various purposes. They are used for simulations, games and most important of all, for the very basics of cryptography. As the fields of use vary, so do the requirements of the random numbers whereas the use in cryptography presents the biggest challenge. These high demands are not unfounded, considering how crucial random numbers are for the security of almost any system. Furthermore recent revelations (e.g. the NSA backdoor in the NIST standard [1]) have shown again that attacks on random number generators are not merely theoretical threats.

Although random number generators have improved in performance and reliability, at the very bottom of every system we are constrained by the provision of true random bits. Even the best pseudo random number generator (PRNG) is useless, if initialized with a weak seed. Providing true randomness however, has proven to be difficult and very expensive. In addition, well-established true random number generators (TRNGs) are usually a black box to the customer. He is promised true randomness, but can not reconstruct the generation of the random numbers and therefore his security is based on trust in the manufacturer rather than trust in the system.

But since randomness can be found almost everywhere, why do we have to rely on expensive TRNGs? With our project we intend to produce homemade, feasible randomness. Our motivation is to find a way to provide true randomness, based on a physical event which should be both affordable and easy to understand, even for a customer. To achieve this, we assume that a simple physical event such as swirling objects in a ventilated container could provide enough entropy to generate true random bits.

The comprehensibility is a key point of our project, since we want to enable any end user to verify the generation of the random numbers single-handed. Which enables the user to detect whether a component of the generator has been tampering the random data. A property that is completely missing in current TRNGs. There already have been similar projects (e.g lavarand [2]), yet they did not manage to establish and faded. With our prototype, we want to offer a way to produce true randomness and therefore high security with relatively low effort. Especially when considering that an intrusion in such a random number generator would require physical access and immense funds. In our time of omnipotent attackers, we consider this to be worth a try.

## 1.1. Definition of the Project

Before going into detail about technical or conceptional issues, we are going to provide an overview of the initial situation of our bachelor thesis. Submitted by our tutor Reto Koenig on March 16 2015, our bachelor thesis is defined as followed:

> True random generators which are in use today, are not verifiable by nature, as they claim to gather entropy from uncontrollable context (on quantum level). Hence they present us with data which results from a black box process. For cryptographic tasks such as randomized encryption, it is important for the sender to know that the randomization used during encryption is truly high entropy for any other party. This way it is important be able to trust the randomization gathering machinery. This is only possible if this machine can be challenged —verified. The goal of this bachelor thesis is to create such a verifiable randomization gathering machine and to prove its maximum entropy. A special challenge is the question on how to provide such a random stream to a computer in a possibly infected computer environment, without cheaply losing too much entropy.

To solve the problem given in the definition, we are going to create a concept on how to produce entropy in a way that is both understandable and verifiable for an end user. To address the advanced requirement of transmission of the random stream, we are going to provide a stable and safe solution for data transmission between the generator and a consuming server. To prove that our concept is feasible, we are going to implement a corresponding prototype.

## 1.2. Fundamental Concepts of Random Number Generation

In Random Number Generation we differ between two really basic constructs; Pseudo Random Number Generators (PRNGs) and True Random Number Generators (TRNG).

TRNGs are quite simple to understand, they extract entropy form an apparently random physical event to provide randomness (e.g. throwing a dice and writing down the resulted values). As we will see later on, current approaches of TRNGs measure atmospheric noise or the movement of electrons and are quite difficult to implement and therefore rather expensive. Yet they provide "true" randomness and hence high quality entropy, which is nearly impossible to guess or reproduce. TRNGs are relatively slow and usually fail to provide the amount of random numbers needed by most computers / servers. This is where PRNGs get useful.

PRNGs are software based random number generators. They basically take an initial value, the so called *seed*, and use it with a one-way function (e.g. hash function) that deranges the input and produces seemingly random outputs. The same process is then repeated with the

resulted output, which produces a big amount of pseudo random data from a small initial value. This leads to a big advantage in terms of performance, with little entropy a PRNG is able to provide a massive amount of random data. But there are also some weak points. For one thing, the output is just seemingly random. Any other computer could reproduce the full output, given the initial seed it is based on. Therefore it is essential for any PRNG that attackers are not able to retrieve the seed value. Furthermore, the PRNG will at some point reproduce its initial seed value and hence the output will start all over again at some point, which is called the *period* of a PRNG. This weak spot can be resolved by updating the seed regularly (*reseeding*).

## 1.3. The Entropy Source Issue

Mainly because of the performance and flexibility most systems use PRNG for random number generation. Yet all PRNG depend on initial seed values and on the fact that those are random. If they fail to provide a random seed value, an attacker could be able to reproduce it and hence reproduce the whole output. A quite popular attempt to solve this problem, is to use a variety of system states to derive a seed value. This could be for example a composition of usage information about CPU, RAM and other hardware components. However, these values are usually situated in a well-known range and can be guessed to a certain extent which means that they are not truly random and do not provide high entropy. Furthermore most systems usually have a problem providing entropy during start up and an attacker might even be able to influence those states from outside.

For example if a generator is using the servers incoming or outgoing network traffic as an input. An attacker could easily guess, maybe measure or even influence network traffic by manually sending requests to the server. In the past, poorly chosen system state based entropy sources have already led to serious security breaches (e.g. in Netscape [3]). Now if we provide entropy in such a manner on a server, there might be just about enough information to dodge even a well informed attacker. But what if we apply this concept to smaller systems such as a common router at home or embedded devices? Often these system have really big difficulties providing enough entropy for strong cryptography.

Although the usage of system states is the current best practice, it is not a really stable solution. Contemporary PRNGs have started to include this fact in their concept. They try to decrease the dependency of single entropy sources by splitting up the output of the sources in so called entropy pools. Those pools are stockpiling the random data and are used to seed the PRNG in irregular periods. The goal is, that even if one entropy source is influenced by an attacker, other sources can overcome the temporarily infection of the generator during the reseed procedure (see section 2.3.2 The Fortuna PRNG). Such approaches might increase security, but they do not solve the problem of the missing entropy. If a server is using a lot of cryptography we are going to reach a point where it is simply not possible to provide enough entropy. And this is where TRNGs come in to play.

## 1.4. Current State of the Art

Now that we have taken a closer look at the advantages and disadvantages of PRNGs, it has become obvious that a combination of TRNG and PRNG would be the best practise for cryptography. Especially for servers or network devices who need a lot of random input (e.g. for encryption). Of course it is neither common nor convenient to purchase an expensive, additional device solely for the provision of random bits. Nonetheless the interest in TRNGs grows and there are already lots of companies who are providing so called hardware based random number generators for that very purpose. These are some of the currently available solutions:

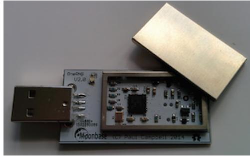
**Entropy Key**, Simtec Electronics (UK)
USB-stick based solution who measures noise on components who are under high voltage. The company describes the product as economically priced (36£) and simple to use. There is no explicit information about the performance.

**qStream**, QuintessenceLabs (Astralia / USA)
Server Module extracting random data from a built-in laserbeam on quantum level. Strong performance with 1GB "true" random bits per second. Information about the pricing is not publicly available.

RANDOM.ORG

**random.org**, Randomness and Integrity Services Ltd. (UK)
Measurement of atmospheric noise using common radios who are situated all over the world. Web-based service that can be accessed for free for a limited amount of random bits per day. There is also a pre-paid premium access, the costs depend on the amount of random bits required per day.

**Quantis TRNG**, ID Quantique (Switzerland)
Quantum level measurement of photons who originate from a collision of a light beam with a mirror. Available as USB-device or as PCI / PCI Express module. The performance varies between 4-16Mbit per second and with prices between 1100 and 3000€ this is a rather expensive solution.



**OneRNG**, Jim Cheetham and Paul Campbell (New Zealand)
A USB-Stick based solution which is organized as an open hardware / software project. The entropy is collected using a diode circuit and a radio-frequency circuit. It is the only generator we came across that is promising a full verifiability of hard- and software. The Firmware is signed and the cover can be removed to verify the built-in hardware components. With 350 kBit/s it might not be a really fast solution but the pricing for the hardware is supposed to be reasonable.

As we can see there are various approaches that differ in both performance and pricing. But apart from the oneRNG project none of them does provide a possibility to verify the soft- or hardware. Most importantly if we do not possess advanced knowledge in electronics or physics, we will not even be able to fully understand where those devices are getting their entropy from.

We must also mark that most companies are publishing impressive results of statistical tests to underline the reliability of their generator. It is fundamental to be aware that statistical test never are perfectly accurate. To really prove that a generator is providing "true" randomness, one would have to test its infinite sequence, which of course is just not possible. This means that a real TRNG could fail a statistical test because of a long, seemingly biased sequence of bits, but still be truly random when one would take a look at the bigger, infinite sequence. Since we are limited by time, testing a sequence with a statistical tests will always be just a fraction of the whole output and not fully representative. Furthermore, the output of a PRNG with regular seeding could be disguised as a TRNG. The output could pass all statistical tests, but anyone in knowledge of the seed-file could easily recompute the whole output on another computer. Nonetheless statistical tests are important to determine tendencies of an output, as we will see later on in our implementation. But it is important that we do not take statistical tests as a guarantee for reliable, solid randomness and certainly not for true randomness.

## 1.5. Comprehensibility and Verifiability of Random Number Generators

Now as we have seen previously most TRNGs produce random numbers in a context that can only be fully understood by experts of the corresponding field. This leads to the actual key point of our project, the comprehensibility and verifiability of TRNGs. The bigger part of the users of such hardware random number generators will not be able to comprehend how exactly those numbers are produced. More importantly, they will not be able to verify that those numbers have not been influenced. This might sound like a small peculiarity, but given the current circumstances this property represents a big security risk.

There are various ways on how an attacker could influence such a device before it gets to the end user. Firstly, since the latest NSA revelations one has to be aware that prominent attackers do not shy back from intercepting product shipments to incorporate their own soft- or hardware based back-doors [6]. Furthermore one has to trust the producers not to implement back-doors during the manufacturing process. A risk that is not unfounded since national surveillance programs often influence even private enterprise companies. Furthermore any company could get exploited by an attacker without their knowledge, which could also lead to unplanned changes to such a product.

Our conclusion is, that to enable trust in a specific hardware random number generator, we must be able to withstand such attacks. We think that this can be done by creating a concept that detects whether hard- and software is acting the way it is intended to. This implicates that the user must test the system on a really low physical layer, which also means that the user must understand the very basics of the context used for entropy gathering. Our attempt to create such a verifiability in a simple and understandable context is described in the following chapters.

# 2. Conception

After having clarified the current situation and the motivation of our assignment, we now define the conceptional side of our thesis. The following chapters are introducing the concept of our project. It is a refined summary of ideas, thoughts and solutions that we developed.

## 2.1. Our Approach

Our goal is to provide entropy with a hardware random number generator whereas the process must be verifiable. To do this, we need a random phenomena that can be detected by a simple camera. From the pictures taken, we must be able to then derive random values:

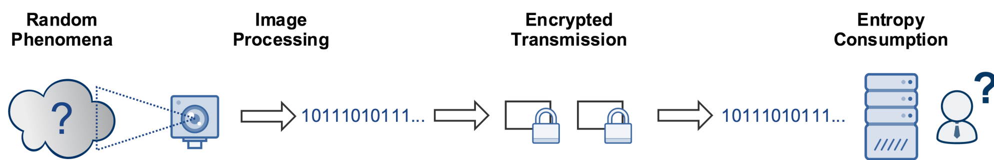



Figure 1: Basic setup of a camera based generator

To transport the values safely to a consuming server or host, the connection must be encrypted. Given that, we have the basic setup of a common TRNG. The key point is, that anyone is able to understand that a picture is taken every few seconds of this random phenomena. And that this picture can be displayed as numbers, or even bits. Whenever the picture changes, the representation of bits will differ and hence we have a resulting stream of always changing random bits, corresponding to the phenomena. Up to that point we do not have the possibility to check whether those bits really are created from the picture. To get rid of the black box process, one simply exchanges the motive of the camera with something static, something predictable:

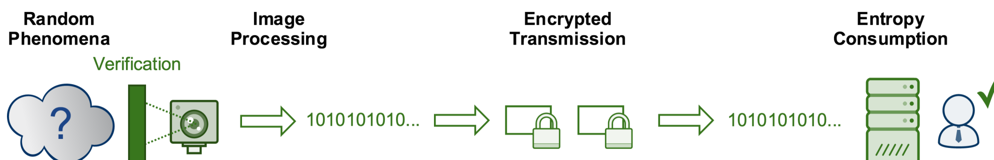



Figure 2: Verifiability of a camera based generator

This enables anyone to test whether the camera and the image processing are producing reliable output and therefore represents the verifiability of the concept. We could even go a step further and insert a changing motive instead of a static one, for example a short excerpt of a movie. This would result in seemingly random output, but could be reproduced

easily on a different machine and therefore tested. Any soft- or hardware based backdoor can therefore easily be detected, since the resulting output would differ from the expected verification results:

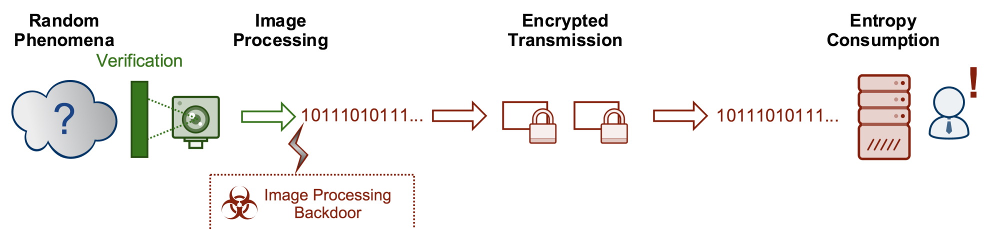



Figure 3: Intrusion detection due to verifiability

The real difficulty lies in finding a random phenomena that possesses such properties. Or rather constructing a device that can produce such a phenomena on demand. In our project we decided to use swirling objects in a ventilated tube. The concept can also be adapted to any random phenomena that fulfills the preciously described requirements.

#### 2.1.1. Preliminary Work

To enable a productive development project during our bachelor thesis, it was fundamental for us, to address the topic in previous modules during the past semester.

In the scope of the module *BTI7311 Computer Science Seminar*, we studied the theoretical background of random number generation and introduced it to our co- students.

Since the prototype signifies a big physical component, we also examined the feasibility of the random phenomena in the module *BTI7302 Project 2*. The main focus was to produce a first prototype, to show that entropy can be gathered from swirling objects in a tube with high air turbulence. This was merely a proof of concept to show that we can extract "something". How to increase the performance and quality of that "something" and whether the retrieved random data can be verified, transported and used on another system, was not addressed.

### 2.2. Generator Concept

Since we want to create random numbers with a TRNG, we need a corresponding random phenomena. The easier the phenomena the better, since the most important thing is that the phenomena is coherent and easily comprehensible for an end user. The best way to understand phenomena is by watching it happen. Hence we decided to head for a phenomena that is detectable by a camera and extract entropy from the images taken of the random

phenomena. That way it would be easier for an end user to reconstruct the process of random number generation.

### 2.2.1. Use Cases

We defined the following use cases based on the requirements we defined.

**UC1 - Provision of random numbers**
Our true random number generator provides entropy that can be seeded to an existing system.

*Test Case UC1*
The images taken by the camera are processed and transmitted to a target server. On the server we can verify whether the data is available.

**UC2 - Verifying the generation process**
The generated random data is verified and therefore indicates any intermediate manipulation.

## 2.3. Server Concept

The server receives entropy from the random number generator, checks its validity and seeds the data into the target PRNG of choice. The received entropy can be verified in coordination with the generator.

### 2.3.1. Use Cases

**UC1 - Use the received random data for seeding a PRNG**
A given application is in need of random numbers and would like to profit from our random number generator.

*Test Case UC1*
The server is seeding the received random numbers to a file or to a PRNG of choice. The PRNG therefore increases its entropy account and the application is not blocked anymore

**UC2 - Check the output during the verification**
A system administrator is testing the generator and wants to verify the random output used for seeding the server.

*Test Case UC2*
The system administrator checks the corresponding output files and compares the random bits with the expected values.

The seeding into a PRNG structure is the main goal of the server application. Since we decided to seed the /dev/random/ device, we took a closer look at this PRNG.

### 2.3.2. The Fortuna PRNG

Fortuna was introduced by a collective of cryptographic experts. It is widely known since it has been adopted in the /dev/random construct on Unix based operating systems. Fortuna is known to be a quite robust generator, not mainly because of it's algorithmic basics, but rather because it takes great care on how and where it takes its entropy from.

The Fortuna generator can be divided into two main parts. First of all, there is the generator core, which is running a block cipher in CTR mode. It is using a 128-bit counter as input and encrypts it with a given random key, which gets reseeded continuously. This main part of the generator does not differ from other common PRNGs, it is able to provide reliable random data, given that the attacker does not know about the random key.

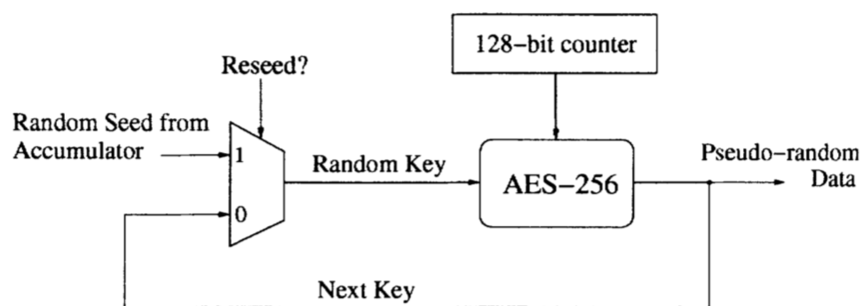



Figure 4: Generator of Fortuna

The other main part is the entropy accumulation. It provides entropy for the reseeding of the previously mentioned random key. But rather than using the entropy from a predefined source as most PRNGs do, Fortuna is using entropy from a elaborate system of different entropy sources.

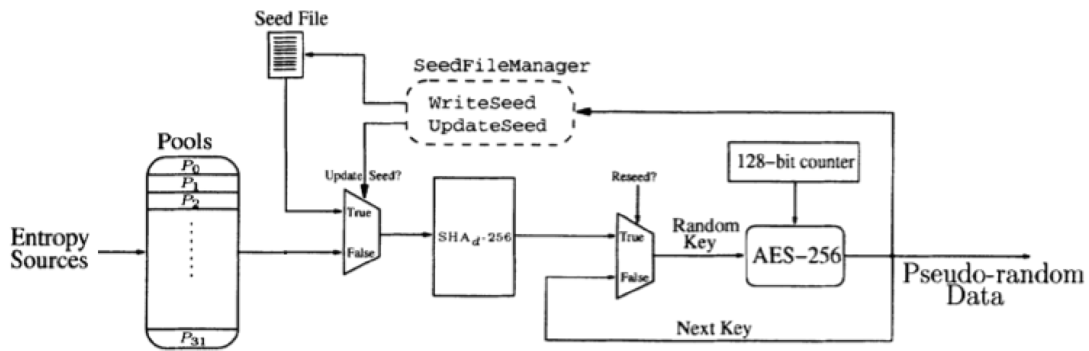



Figure 5: Fortuna overall structure

It uses so called entropy pools, where entropy from different sources are stacked. Fortuna is using 32 pools, but retrieves entropy only from those, who are dividable by the current 128-bit counter value. Hence, bigger pools are rarely used, and lower pools more often. The consequence is, that bigger pools such as pool 29, 30 or 31 are buffering a lot of entropy, thus they are usually able to seed the whole key on their own once they are used. This leads to the big advantage that Fortuna is able to recover from attacks. For example if an attacker controls an entropy source and retrieves the internal state. In that case, the attacker is able to predict any future outputs, yet after some rounds a higher entropy pool is going to reseed the key and reset the internal state. This is a very strong property, which also shows that one reliable entropy source (even if not that performant) can add a lot to the security of a generator.

Furthermore Fortuna is continuously seeding a so called Seed File during run-time. The purpose of this file is to provide entropy during start-up. If a server needs to restart, he is usually not able to provide a lot of entropy. In case of Fortuna, the server will just use the content of the Seed File to reseed its key and hence will be able to provide random output even if it was shut down for example by a Denial of Service attack.

## 2.4. Transmission Concept

The main question about the transmission was what medium should be used to transport the random data from the generator to a server.

Our first thought was to use Ethernet or rather, TCP/IP. Mainly because it is an established way of communication and because it would also offer the distribution of random bits to multiple servers in the same network. When we started setting up a concept for authentication and encryption, we quickly realized that it might offer too many attack opportunities for an attacker.

The main weakness of a random generator, is the fact that he is providing random numbers for a server who is in need of entropy. More closely, one has to assume that the server does not possess enough entropy when he is initially connected to the generator. Hence the only way to authenticate or encrypt such a connection, is to exchange a shared secret over a safe channel.

This was the main point why we decided not to use Ethernet. Because the encryption will solely base on the initial secret transmitted via safe channel and be vulnerable to brute forcing attacks on the long run. This also means that a refresh of the key using one of the common key exchange methods (e.g. Diffie Hellman) would be of no use, since any new key would be derived from the previously transmitted random data. Therefore a key exchange would not provide any additional security.

We therefore decided to use a simple peer to peer USB connection instead. This offers us more safety and privacy, since it way more difficult to listen in on a peer to peer connection than into network traffic. Furthermore there usually is an empty USB slot on any server, whereas network interfaces are usually scarce since and mostly already occupied.

### 2.4.1. Initialization and Authorization

At the very beginning of the transmission, the generator and the server must retrieve one another. The USB port can be figured out quite easily, if necessary the user can be instructed to plug in the USB cable during the start up of the application, to make the retrieval easier.

Even though we are using a USB cable to connect, we must assume that one way or another a attacker could influence this data-stream (e.g. via USB hub or other USB devices). To make sure that the server and generator are resistant against various attacks (see section 2.5.2), authentication is fundamental. As we have previously stated, we must assume that the server does not possess any entropy and hence we need to exchange a secret on a safe channel.

We chose the length of this secret accordingly to the current standards of encryption [7].

Since we decided to use a symmetric encryption (AES), we fixed the key-size to 256 bits. Once the secret is retrieved by both parties, they are starting with the authentication phase via USB.

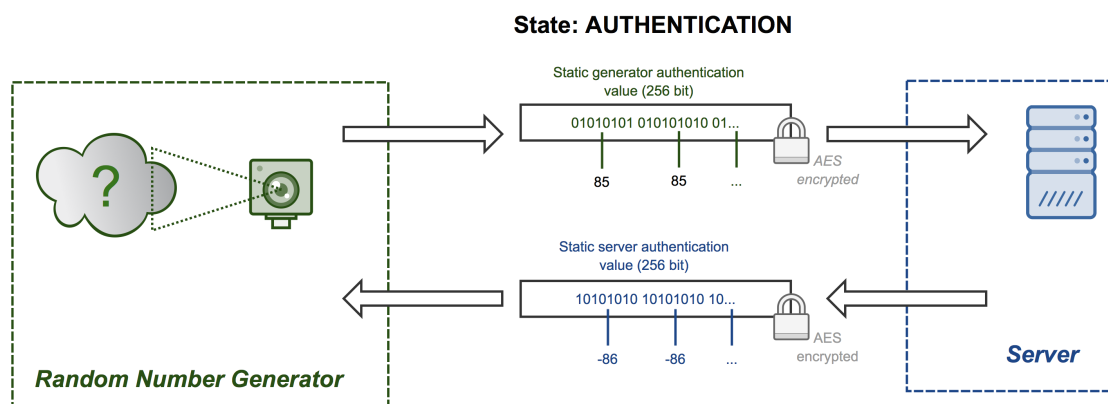



Figure 6: Authentication setup

Firstly, the generator is sending an encrypted authentication packet containing a predefined static value, using the previously exchanged secret as a key. The server can then verify the authenticity of the generator by decrypting the packet and checking the predefined value. This procedure is then repeated on the server side, using another predefined value.

### 2.4.2. Transport of Random Data

Now that both sides are authenticated, we can exchange the random data. To obtain freshness of the packets and avoid replay attacks, we add a sequence number to the random data. The sequence number is incremented on both sides (generator and server) and encrypted together with the random data. This enables the server to detect and drop invalid or forged packets sent by an attacker.

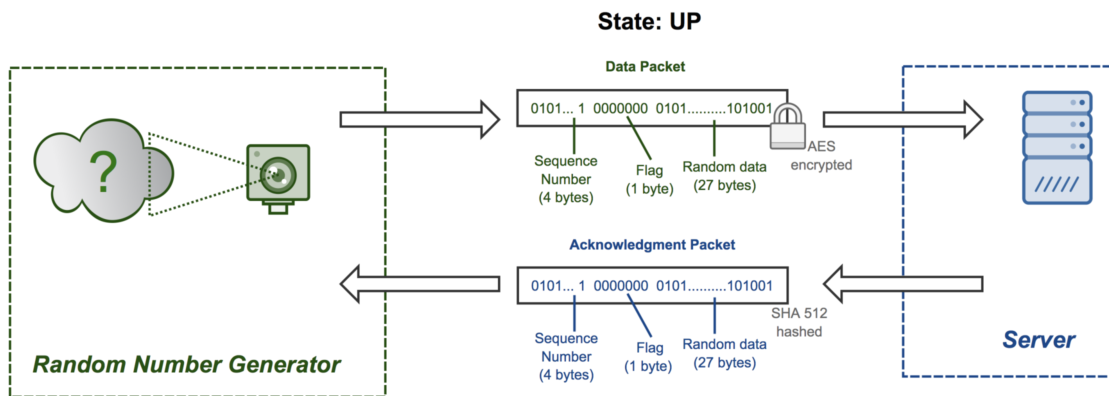



Figure 7: Data transmission

To make sure that the server does not blindly seed any data received, we send an acknowledge packet back to the generator for every "data" packet. This acknowledgment package contains the hash value of the previously received data packet. The generator is then testing whether the received data is valid and sends the next data packet.

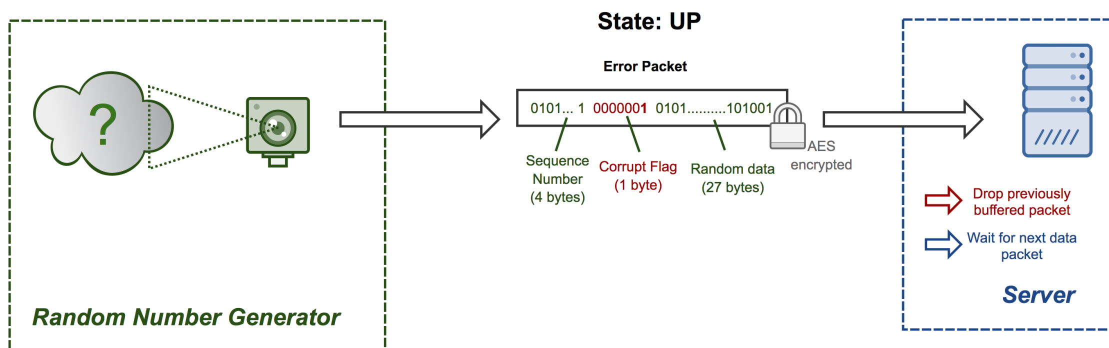



Figure 8: Corrupt data handling

If the data is invalid, the generator must be able to inform the server. To this and to enable further development of this data transmission concept, we added a additional flag-byte to the data packet. For now this only represents valid data (0) and corrupt data(1). Every ingoing data packet is therefore initially buffered by the server and not yet used for seeding. If the following packet contains a corrupt flag, the packet can be dropped. If the next packet follows, it can be safely used for seeding.

### 2.4.3. Timeout and Offline Mode

As we will see in the next chapter, it is important to handle interruptions of the transmission. To do so, we defined timeouts for both generator and server. If one component does not receive a valid answer from the other until the timeout expires, they are going to enter the offline mode.

If the transmission is in the state offline, it is properly authenticated, but has not been successful in transmitting data for a rather suspicious period. This can be caused by a temporarily interruption of the connection, by damaged hardware or even by an attacker. In the offline mode both server and generator are supposed to check for new data packets less frequently to avoid performance issues and since the other site is apparently not available at the moment. It is also important to inform the system administrator about this interruption. In the case of our prototype, errors are written to the log file, error messages are printed on the graphical user interface and even an audio warning is emitted.

## 2.5. Risk Assessment

Since this concept is going to influence a fundamental mechanism of the security of a system, we must analyze possible security threats arising with it. We analyzed possible threats on the three main components generator, server and transmission.

### 2.5.1. Generator Security

Firstly, and most importantly, we must face the security related risks of our generator.

**Hard- or Software Intrusion - Backdoor**
A probable attack on the generator would be the incorporation of a hard- or software based backdoor into the generator. Most certainly this would be a common PRNG structure who is producing outputs based on a seed file which is known by the attacker. Therefore the attacker could compute all outputs on another computer and reproduce every single random bit.

*Countermeasures*
This attack could easily be detected by the test procedure, since the output would not match the expected values from the verification of the generator.

*Advanced attempts*
To dodge the test procedure the introduced backdoor construct would have to recognize at what point the generator is tested and pause its malicious behaviour during the test procedure. One way to do this would be by using a sophisticated image processing strategy

which tries to fingerprint the insertion of the test image. It is highly questionable whether even a sophisticated image processing algorithm would be able to distinguish between the inserted and "real" picture. This also depends strongly on the implementation of the mechanics (e.g. whether the illumination changes during the verification procedure). Another attempt would be to add a second camera or a photo sensor, to detect solely the insertion of a verification module.

Especially the latter attempt would need a tremendous effort, since the camera or a sensor could be detected by the customer if he opens the generator during installation (the smaller the camera, the more expensive). Furthermore those components would need to be attached to the microcomputer which is easy to detect, even for an end user.

Our conclusion is, that advanced attacks on the generator would be too costly even for a strong attacker. And it would be too likely that they get detected by the end user. Hence we consider the generators as very resistant against attacks.

### 2.5.2. Transmission Security

Another attack vector would be the transmission from the generator to the server.

**Wiretapping**
An attacker might try to determine the transmitted random data, so that he can reproduce the seed values for the server PRNG and hence the output of the PRNG. To do that, he is going to try listening in on the conversation between the generator and the server.

*Countermeasures*
Firstly since the transmission is done via an USB peer to peer connection it is extremely difficult to establish a foothold on the transmission. Even if the attacker manages to listen in on the conversation (e.g. via infected USB hub), the data would still be encrypted. The attacker would have to break the initial 256bit secret. He would need to transmit this initial data and brute force the secret on an infected computer in or outside the network. The transport would have to be done either over a infection of the server or by overcoming the air-gap, which is a technique that has not been exploited on level of a USB device. Furthermore even if the secret is broken, an attacker would still have to transport the retrieved random data to his attacking computer. This must be done over the target server or by overcoming the air-gap. Once this is done, the random data stream would create a immense noise in the network, which is going to be easily detectable for network administrators.

**Man in the Middle**
An attacker might even try to go a step further and establish a man in the middle attack. This would enable him to replace the random data with its own pseudo random data. In this case, he would need to break the initial secret as well.

*Countermeasures*
This setup corresponds to the previously described attack. The attacker would also need physical access to the USB cable. Apart from just breaking the secret, the attacker would also need to maintain a PRNG somewhere, who is replacing the sent data with its own pseudo random data. This attack can easily be discovered by during the verification procedure, if done properly. It just takes a simple comparison of the random data on generator and server to find out that the data differs.

Given these circumstances, we strongly believe that our transmission concept is resistant, even to strong attacks. But only if the verification / test procedure is done properly and regularly.

**Denial of Service attack**
Another quite effective attack is to prevent the server from receiving entropy. To do this an attacker might try to block the transmission. This could for example to be done by using an infected USB hub who blocks the traffic between the server and generator after a few data packets.

*Countermeasures*
By doing that, both server and generator application would fall into the offline mode. Since we do not plan to include a web server into our server application, the real challenge is going to be to inform the user about such a fact.

### 2.5.3. Server Security

Lastly an attacker could also directly attack the server to disrupt the random number generation. It even seems to be the most likely attack vector, since it is the only component that is connected to the outside world.

**Denial of Service (DoS) attack**
The most probable attack would be if an attacker launches a DoS-attack towards the server. The server will shutdown unexpectedly and lose its initial 256 bit secret. Once restarted, he will not be able to reconnect to the generator and will lack of entropy.

*Countermeasures*
There are different approaches to deal with this problem. Since our system does not offer a way to directly contact the system administrator and inform him about the interruption, we had to take a more serious approach. We decided to save the secret key, as well as the current sequence number locally on the server during run time. This is fully based on the assumption, that the server is not infected and the hard disk is encrypted accordingly. That way, when the server is starting up the application, it can simply check whether such a 'hook'-file exists or not. An existing connection can then be retrieved, since the generator is still in offline mode and the server can reproduce the connection using the secret and

sequence number.

**Infected Server**
We did not investigate this scenario in depth, because providing randomness simply becomes useless once an attacker gained foothold on the server. On an infected server an attacker could weaken the security in various other ways. Mainly an attacker would take actions before the data is even encrypted. He might even use the provided random numbers to encrypt and transmit confidential data to a command and control server. The only possibility to verify such a intrusion, would be to compare the seeded data with the data on the raspberry PI. But this could be as well manipulated by the attacker and would include, that the attacker is still working against encryption instead of just moving around it which would the most uncomplicated way if he gained access to the server.

### 2.5.4. Trust Assumptions

As for most security related systems, it is vital to make as few assumptions as possible. Given the previously described scenarios, we nevertheless had to fix some points to be able to guarantee the safety of our concept.

*Assumption 1*: Safety of the system infrastructure
All components of our system are during run-time not physically accessible for potential attackers.

*Assumption 2*: Safety of target host
The target host / server, to which the random data is transmitted, is not controlled by an attacker. This mainly means that the resources of our application on the host / server are not accessible or modifiable by an attacker.

*Assumption 3*: Test procedure is done properly
The predefined test procedure of the components is done by a human continuously, in random sequences. During the test procedure we assume that we have a moment of privacy.

*Assumption 4*: Resistance of test procedure
The estimated effort to bypass our test procedure might be possible for strong attackers, but would require a tremendous effort and as a result be too expensive for the outcome.

Apart from the security related risks, as in every development project, there is a chance that our concept or ideas can not be implemented as planned or do not work as anticipated. To minimize such failures, we tested the functionality of our prototype beforehand as a proof of concept (see section 2.1.1 Preliminary Works).

Since we are aiming to produce a functional prototype rather than a final product, our thesis does not include any economical prospects. Hence it will not include any risks concerning

sales and distribution.

## 2.6. Testing randomness

To have a point of reference whether the random data produced by the prototype is random or not we read up on different ways to test for randomness.

A good random number generator will always produce small sequences that do not seem random, but are random nonetheless in the bigger picture. Therefore it is quite a difficult task to prove that a sequence, (or a produced output of a random number generator) actually is random.

### 2.6.1. Statistical randomness tests

The usual, theoretical approach for testing randomness is to take many sequences of a random number generator and feed them into statistical tests. Because a good random number generator will also produce sequences that will not look random at all, the idea is not to judge only on the criteria of failed or passed tests but to get a broad view over the whole generated random output. If the majority of data sequences pass the tests it will most likely be of good quality. Many failed tests on the other hand, should raise suspicion and induce further analysis.
Up to this day, there are many different statistical tests, the first of them being published by M.G. Kendall and Bernard Smith in 1938. In 1995 George Marsaglia published different and more advanced tests.

Kendall and Smith differentiated "local randomness" from "true randomness", implying that a true random source may fail these test for some sequences and thus not be locally random [13]. The tests from Kendall and Smith are based on the hypothesis, that all numbers and patterns that formed out of the sequences are distributed with the same probability. They are relatively easy to understand:

The **frequency test** is the most simplistic of all.

It tests whether each number occurs more or less the same amount of times.

E.g. well distributed random numbers: 0: 808, 1: 801, 2: 820, 3: 799, ...

E.g. badly distributed random numbers: 0: 408, 1: 801, 2: 1020, 3: 599, ...

The **serial test** and the **poker test** are working in a similar way. But rather than numbers, they test the occurrence of a sequence of numbers. Whereas the Serial test checks sequences of the length of two (e.g. 00, 01, 02, ..) and the Poker test checks sequences of five (e.g.

00001, 00002, 00003, ..) numbers.

Finally the **gap test** checks for the number of decimals between certain numbers. If the difference of those gap is continuously too small, there is a repetition in the random number generator. E.g. the gap between zeros in the sequence 026590 is 4, in the sequence 050 only 1.

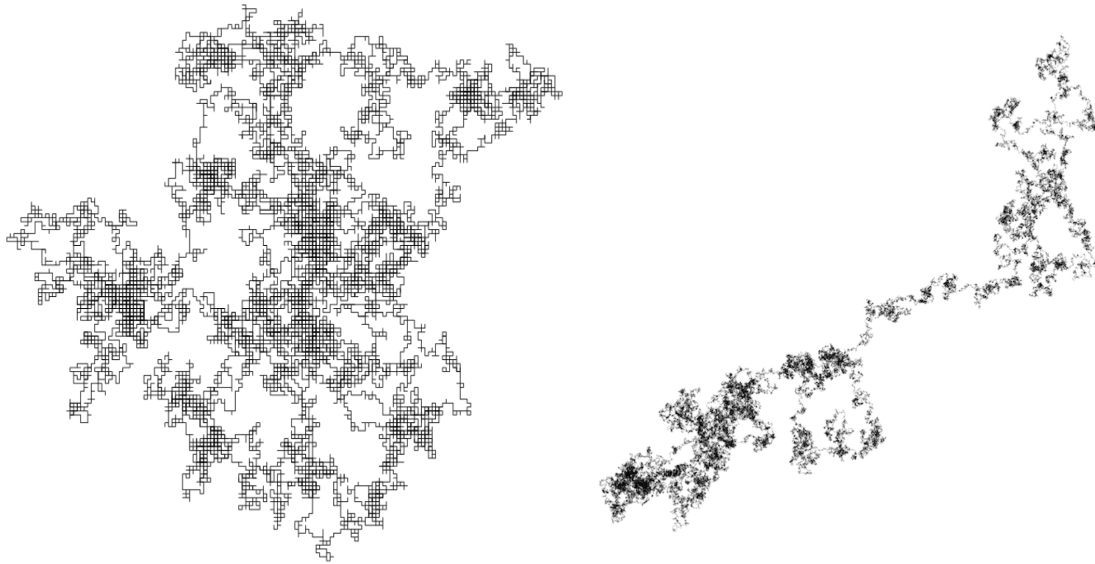
There are various statistical test that are more recent, for example comparing numbers and patterns in a three-dimensional way. Due to the in-depth statistical and mathematical structure of those tests we avoid a detailed explanation of such tests.

### 2.6.2. Human perception

Methods for getting an overview over a big sequence of apparently random numbers often involve the human perception. The human brain is an excellent pattern recognition machine and performs the complex task of seeking and recognizing patterns better than any computer or other deterministic machine. This finding is also used in other areas (e.g. Random Art [16]). In randomness testing, this advantage can be used in different ways to detect possible repetitions or patterns in a random source.

#### Random walks

Random walks can be performed in multiple dimensions, although the following chapter only covers two dimensional walks. One possible way to explain the random walk is as a person walking on a two dimensional grid having to decide whether to go left, right, back or forward after each step. If this decision is made by a random source, a map is generated allowing the observer to see either a path that is totally random or an apparently random walk that tends to go into one direction when looking at the entire generated path.



(a) More random input [14]

(b) Less random input [15]

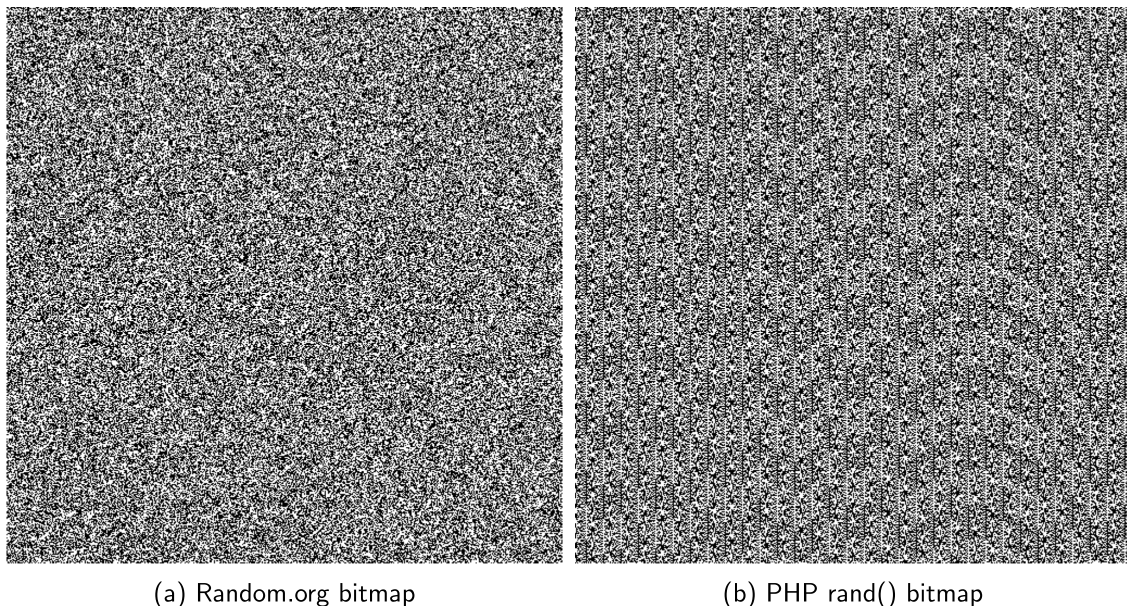
Figure 9: Comparison of two two-dimensional random walks

Figure 9a shows an evenly distributed random walk whereas figure 9b has a general direction in which the path moves. The latter is an excellent example for a seed that may look random when observing a local spot but fails to meet the requirements of random given the big picture.

### Graphical pattern recognition

A digital information, for example an array of bits can be represented using a bitmap, also referred to as raster graphics. A simple black and white bitmap for example, can represent an array of bits by simply filling out a field either in black if it is a 1, or in white if it is a 0. If a random sequence is fed into a bitmap, it is possible to spot visual patterns immediately, instead of having to use a complex statistical test.

A good example is the comparison between a random seed from random.org and PHP's rand() function, performed by Bo Allen [17].



(a) Random.org bitmap (b) PHP rand() bitmap

Figure 10: Comparison of bitmaps between random.org and PHP rand() [17]

Figure 10b clearly shows a pattern in a vertical and rippled fashion and illustrates a bad example. However, this method is not a fail proof indicator for the actual quality. Some random seeds will not show visible patterns even if they contain repetitive behaviours.

### Auditive pattern recognition

A digital information can also be fed into a digital to analog converter (DAC) which converts the meaningless binary information into an analog signal. This is one of the tasks that a regular sound card in a computer does. The amplified signal can be emitted as sound waves when connected to speakers and similarly to observing the bitmap, an eventual pattern in the random source could be heard as a regular tick, a repeating change of the noise or as any other pattern which would give a hint to poor quality of the random source.

## 2.7. Requirements

To be able to distinguish between necessary and optional components, we must define what basic requirements the system must fulfill.

The following requirements represent the fundamental properties of our generator prototype.

R.1 A concept for the verifiability exists

R.1 The prototype is able to produce entropy

R.2 The derived entropy can be transmitted to another host

R.3 Even in an infected network, the transmission of the random data can be granted

R.4 The target host is able to seed its local PRNG with the received random data

R.5 The whole process of entropy gathering and transport can be verified by a simple challenge response process

R.6 The development and the system itself is documented in a comprehensible way

R.7 The architecture of our prototype enables future development and improvements

## 2.8. Goals

Given the requirements we now can fix specific goals for our thesis. They show what steps have to be taken to satisfy the previously defined requirements. Since not all goals are similarly important, we added a corresponding criteria about their severity in relation to the project:

1 Extremely critical: A fault in the main functionality, the realization of the project is questionable.

2 Critical: A fundamental restriction of the functionality, the requirements will not get fulfilled entirely.

3 Basic: The Project can be realized but there is a high probability of inaccurate results.

4 Advanced: The Project can be realized, but some components will be missing

5 Optional: An irrelevant component will be missing, has no real impact on the project.

Since our project consists of different components, we can structure the different goals into the following main topics.

### Prototype Related Goals

The primary part of our thesis is to produce a stable prototype. During our thesis we are not only going to develop corresponding software, but we also must provide a stable prototype in terms of the physical properties. Hence we must define some goals that must be achieved in order to provide physical stability:

| *Goal* | *Severity* | *Description* |
| --- | --- | --- |
| G1.1 | 1 | The prototype extracts entropy |
| G1.2 | 2 | Structural issues are resolved (e.g. ionization, darkening) |
| G1.3 | 3 | A default configuration setting for all external factors has been defined. (e.g. air supply, exposure time, illumination..) |

Table 1: Prototype Hardware Goals

If the physical goals are reached, there still is a big part to achieve by means of Software. Our prototype must fulfill the following goals:

| *Goal* | *Severity* | *Description* |
|---|---|---|
| G2.1 | 1 | A functional image processing is implemented |
| G2.2 | 2 | The produced random data can be sent to a target host |
| G2.3 | 3 | A default strategy / configuration for image processing is set |
| G2.4 | 3 | It is possible to initialize a connection to a target host via user interface. (e.g. numbers, QR-codes) |
| G2.5 | 3 | A visualization of the verbose data is implemented on the user interface |
| G2.6 | 4 | Basic configuration of parameters can be done via user interface |
| G2.7 | 5 | A life view of the camera can be accessed via user interface |
| G2.8 | 5 | Additional features for analysis of the random data |

Table 2: Prototype Software Goals

#### Host / Server Related Goals

Apart from the generator prototype, we also have to consider what we want to achieve on a target host / server, who is consuming the produced random numbers.

| *Goal* | *Severity* | *Description* |
|---|---|---|
| G3.1 | 1 | Random data can be received by the target host / server |
| G3.2 | 1 | The received random data can be tracked on the host system (especially during the test procedure) |
| G3.3 | 3 | Random data can be seeded into the /dev/random construct |
| G3.4 | 3 | The received data can be extracted or directly fed into statistical tests |
| G3.5 | 5 | Enable usage of the random data for other applications (e.g. Save as File) |

Table 3: Server Related Goals

#### Data Transmission Goals

Additionally, the random data must be transported from generator to a target server.

| *Goal* | *Severity* | *Description* |
|---|---|---|
| G4.1 | 1 | The provided random bits can be transmitted to the target host |
| G4.2 | 2 | The transmission is safe, even in a infected network |

Table 4: Data Transmission Goals

#### Conceptional Goals

Given the previous goals, we have a stable prototype of a generator, transport and server application. Yet we still have not met our most important requirement, the verifiability.

| *Goal* | *Severity* | *Description* |
|---|---|---|
| G5.1 | 1 | The entropy accumulation can be verified with a corresponding test procedure |
| G5.2 | 3 | The results of the testing procedure are verifiable on the target host |
| G5.3 | 4 | The System documentation is coherent and enables further development |

Table 5: Conceptional Goals

## 2.9. Project Schedule

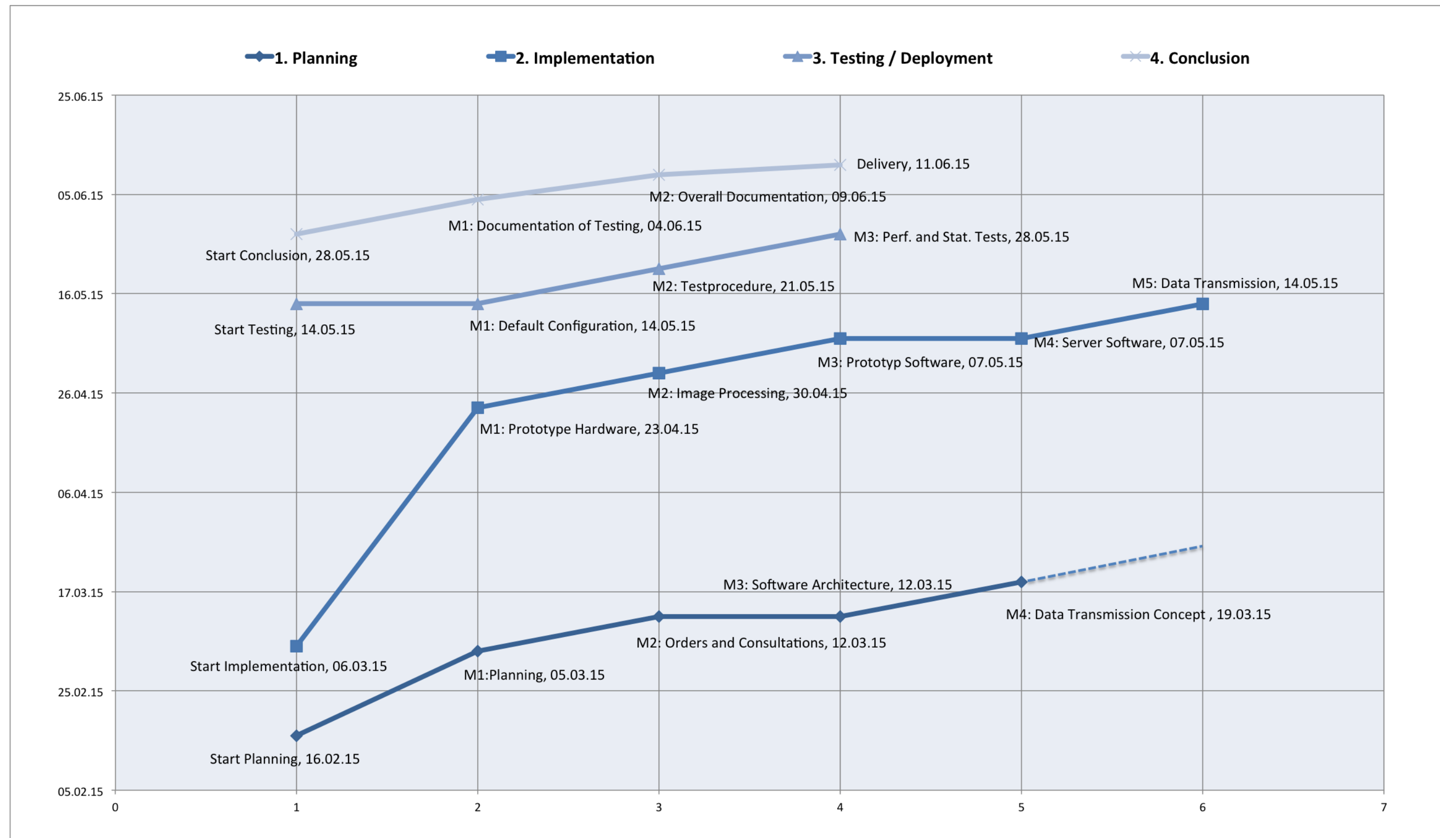



Figure 11: Project schedule

To create a project schedule, we started at the very end of the project and tried to determine which issues have to be solved before reaching that specific point. Traversing the whole project in that manner, enabled us to fix the most important points in our project. Of course this schedule is just a first approximation and will change during the project. Hence the task project planning will be a continuous task during the whole thesis.

## 2.10. Milestones

The resulted milestones can be characterized by two main properties. On one hand, every milestone stands for a set of issues or tasks that have to be resolved upon that point. On the other hand, every milestone signifies risks that could occur, if the milestone can not be reached in time.

### 2.10.1. Phase Planning

**Milestone 1.1 Planning**, due 05.03.2015

This milestone represents a stable first planning of our project. It includes fundamental questions such as requirements, goals and how they are going to be achieved. Furthermore we must provide a documentation that shows what initial situation we are facing in our project as well as defining its scope.

If we are not able to provide a systematic structure for our project, we are not going to be able to prioritize the different aspects of our project. This could lead to dead ends, or more fundamentally we most likely would not be able to achieve our goals in time.

**Milestone 1.2 Orders and Consultations**, due 12.03.2015

Some components of our project must be bought since they can not be provided by our University. Furthermore there are topics in our thesis in whom we do not have in depth knowledge, such as image processing or 3D-Printing. To minimize failures in such areas, we want to consult with corresponding experts at our university before we decide what measures we take.

If this milestone is not reached, this simply means that either we are missing some components for our prototype and will not be able to build the physical prototype in time. Or we did not manage to clarify all weak points, which would increase the chances of a failure in the corresponding areas.

**Milestone 1.3 Software Architecture**, due 12.03.2015

To increase the quality and efficiency of the software it is essential to create a concept first. We need to think about how our software will be used and hence how the architecture should look like. This includes thinking about which patterns could be of use and how they could be applied to the software.

If we do not have a concept for our applications, we might run into architecture based problems. Moreover, our code will not be as easily understandable and the co-working as

well as further development might get difficult. This all could end up in lacking efficiency and therefore loss of time.

**Milestone 1.4 Data Transmission Concept**, due 19.03.2015

Since we do have two different applications, it is indispensable to think about how they are going to communicate with one another. Moreover we need to think about how we can provide a secure communication between the two nodes.

If there is no clear concept on how the transmission is done, we may run into functional problems and we might not be able to provide the desired security.

### 2.10.2. Phase Implementation

**Milestone 2.1 Prototype Hardware**, due 23.04.2015

At this point, the prototype should be in its final state concerning the physical elements. This means that we are done with 3D printing and do not need any more physical components. We possess a stable physical environment.

If we do not have a physically stable prototype at that point, we are not going to be able to calibrate optimal configuration settings for further tests. Aspects such as brightness or density might influence image processing and its results, which will end up in inaccurate results and additional time expenses.

**Milestone 2.2 Image Processing**, due 30.04.2015

Reaching this milestone, means that we have informed ourselves about image processing. Furthermore we decided which image processing strategy is the most suitable for our project and implemented it in our software.

The image processing is the core function of our prototype software. It has a big influence on the performance of the prototype as well as on the quality of the derived entropy. If we fail to implement it at that point, we can not go on with the fine tuning of the configuration and we do not have any indication about how our prototype will perform and how much entropy we can gather.

**Milestone 2.3 Prototype Software**, due 07.05.2015

The configuration and administration of the prototype can now be done using the touchscreen application of our prototype.

If the administration via touchscreen application is note yet possible, future tests are going to take a lot more time since configuration changes will have to be synchronised on the raspberry PI first. Furthermore we need the touchscreen application to verify that the authorization during the transmission setup is working properly.

**Milestone 2.4 Server Software**, due 07.05.2015

The server application is up and running. The application is able to seed the /dev/random/ construct of the underlying server operating system with data received from the prototype.

If the server software is not running at that point, we will not be able to fully test and implement the data transmission as well as the verification process between server and prototype.

**Milestone 2.5 Data Transmission**, due 14.05.2015

The data transmission is implemented on both prototype and server and enables safe transmission of the gathered random data.

If this is not the case, we are unable to go on to the next phase. We will not able to fine tune our configuration an will not be able to make a statement about the performance and verfiability of the generator.

### 2.10.3. Phase Testing

**Milestone 3.1 Default Configuration**, due 14.05.2015

We have fixed an acceptable default configuration for both server and prototype.

The default configuration is important so that we can analyze the data as well as performance of our approach. We also need to run long-term tests and check how resistant to prototype is to interrupts.

**Milestone 3.2 Test Procedure**, due 21.05.2015

The test procedure has been verified and is now a straightforward, documented process rather than a concept.

If we fail to define the test procedure in a easily applicable way, we risk to discourage possible end-users in regularly performing it. This is particularly important because the test procedure stands for the most important property of our prototype, the verifiability.

**Milestone 3.3 Performance Tests**, due 28.05.2015

Various performance tests are done and corresponding results are added to the documentation. This includes the performance of the transmission, the image processing and the seeding on the target server. Furthermore we have seen how the application acts during long-term use.

The information about the performance is an important piece of our documentation. It can be used to situate our solution amongst other currently available random number generators. It also implies whether our random phenomena is suitable for a more advanced prototype or even for a product.

### 2.10.4. Phase Conclusion

**Milestone 4.1 Documentation of Testing**, due 04.06.2015

At this point we added the findings of our test phase to our documentation. This also means that all previous contents already have been added.

If we do not reach this milestone in time, we will be under pressure when it comes to providing a coherent documentation for our project.

**Milestone 4.2 Overall Documentation**, due 09.06.2015

The documentation has been reviewed, re-factored and the layout has been changed accordingly. This also includes the update of illustrations, references, acronyms and glossary.

If we fail to provide a coherent documentation we might not be able to present our project in a persuasive way. Furthermore there would be a big risk that the project will not be continued after our thesis and will fade.

**Delivery**, due 11.06.2015

The documentation is delivered in its final version to the expert and tutor.

# 3. Implementation

To enable a full understanding of how our system works, the following section documents how we implemented our prototype. It shows which hard- and software we used and how we realized the previously described concepts in that specific context.

## 3.1. Main Concept

Our prototype is based on the assumption that Styrofoam beads in a ventilated tube provide enough entropy for random number generation.

The principle itself is rather simple. We use an air box that produces a strong air flow. The air moves through a transparent tube and stimulates the contained Styrofoam particles. Apart from the air stream, the particles influence one another so that their movement becomes unpredictable.

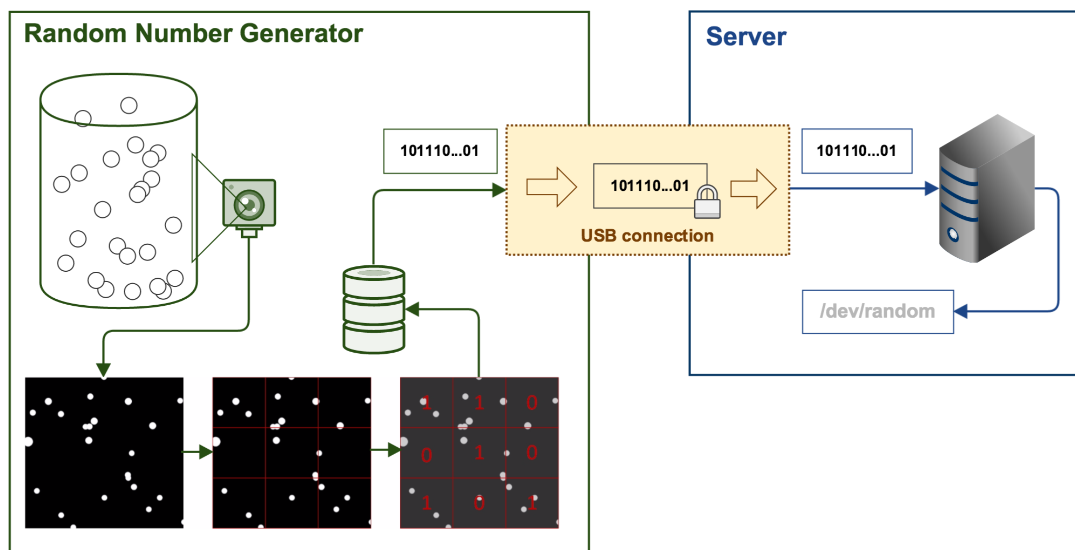



Figure 12: Main Structure

To extract the entropy, pictures are taken in a regular interval and processed into random bits. These random bits are then encrypted and transmitted via USB to the target server. Since it is highly probable that our prototype will not have a high performance, we assume that the target server will use the random input to feed a local PRNG. In our prototype setup we will seed the /dev/random/ device as described earlier.

## 3.2. Development Environment

**Setup Prototype**
As a generator platform, we used a Raspberry Pi 2 with a 32GB SD-Card, running the Raspbian operating system. The pictures are made with a PiNoir camera. The software is written in Java. We used a Git repository to for source code management and Maven for dependencies and the build process, including deployment.

**Setup Server**
To simplify the process of testing the server application, we decided to use a UNIX based operating system on a bootable USB-stick. We installed a Ubuntu 64-bit LTS Server on a 16 GB USB stick, which we used with common workstations at our university for test purposes. Since we did not have specific requirements concerning the server platform, we did not test the application on any other operating systems. Nevertheless we assume that applying the software to another UNIX-based system can be done without big efforts.

The software on the server is written in Java and organized in the same manner as the prototype. The source code is pulled from the Git repository and the build process is organized by maven.

**Repositories and File-sharing**
To enable a uncomplicated and flexible development process, we decided to store our data centralized and easily accessible.

The development of both prototype- and server application was managed by using two corresponding Git repositories. The repositories were hosted on the free source code hosting service bitbucket (`www.bitbucket.org`).

The whole documentation was written in LaTeX. Since manging the documentation via a repository is not ideal, the whole documentation was hosted on ShareLaTeX (`www.sharelatex.com`). ShareLaTeX is a free online LaTeX platform which includes an online editor with real time editing and multi-user access. This offered us huge flexibility, and since the service is free for up to 2 collaborators it did not involve any costs.

## 3.3. Prototype structure

The structure of our prototype can be divided into an air source and an enclosed tube.

In an early stage version of our semester project we implemented this using a simple tube covered by nets with a fan placed under it. This lead to several problems, for example the lack of turbulence and the inability to easily change parts and experiment in order to maximise entropy.

The intention of the next prototype was primary to gain modularity and stability. We wanted to have a structure that allowed us to change the tube part to experiment with different settings.

Moreover, the design should improve the overall stability of our system.

Since it is rather difficult to meet all these requirements while constructing something with existing parts we decided to design a model and print it on a 3D printer.

After doing some research we discovered the free accessible CAD tool tinkercad **??**. Tinkercad runs completely independent inside a browser and provides a rather easy way to design 3D designs. We worked trough the tutorials and spent some time learning the basic functionality needed to design our prototype.

The construct can be divided into the following parts:

**Air box**
A modular source of turbulent air is achieved by an air box. It should be constructed in a way that the fans can be be mounted so that they combined produce enough air pressure to move our particles in the tube. As we were advised from an automotive engineer that we contacted, the inside of the air box should provide a way to insert wind chimes so that the resulting air flow becomes turbulent.

We designed the air box so that it contains mounts for eight 80mm computer fans. The mounts consist of a hole that the air flows through and four screw-holes in the standardized dimensions for computer fans.

On top of the air box is an indentation that will hold the fitting tube connector. The indentation is the exit point for the turbulent air.

Inside the air box we added several anchoring supports which will hold the air chimes.

To enable a better understanding and for the sake of controlling the fans we added numberings in the design.

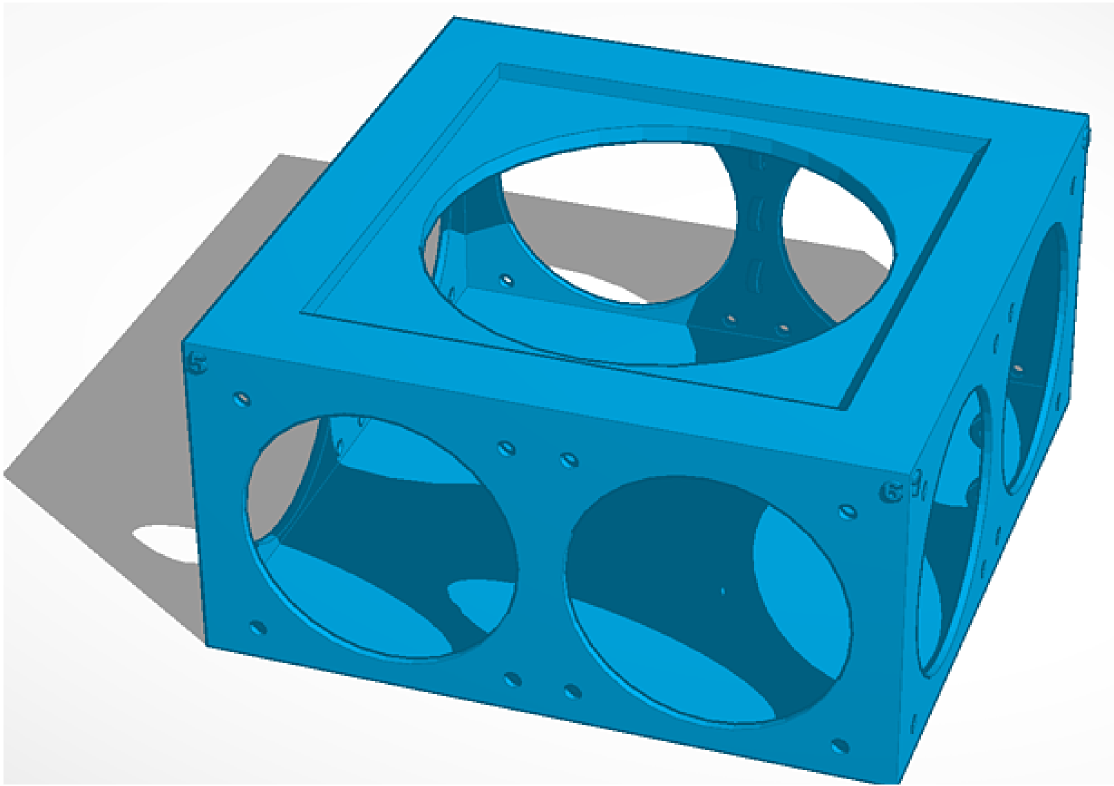


Figure 13: Airbox

**Tube**
The tube is the simplest part of all. It must be transparent and enclose particles.

We used an existing plexiglass tube that we cut into the desired size. It needs a modification so that the bolts of the enclosing parts can be locked in.

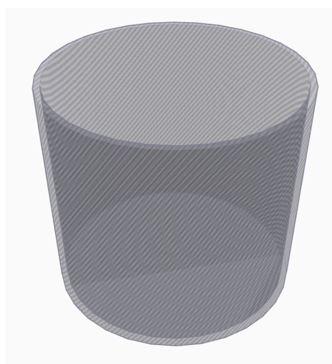


Figure 14: Tube

**Tube connector and tube hat**
The tube connector and the tube hat are the parts that enclose the tube. They have to hold back any particles inside the tube and enable air flow through it.

In a first version we constructed the grids of the tube hat and the tube connector using rings that slightly overlapped. In Tinkercad each object is made from a set of basic shapes. A Tube for example can be created using a cylinder and a smaller cylinder inside it that is defined as a hole. When the two objects are merged together, it is visualised as the resulting objects while still holding the individual parts.

This resulted in a disadvantage for our specific application. The grid was so finely coursed that the design held too many parts for it to compute. We had a finished design of the two tube parts that both refused to group since the browser application was on its limits. We tried exporting the STL files in order to open in it in a standalone application. Tinkercad however does not provide a version that can be used offline.

We did try to install a similar product from Autodesk called 123D Design but it was not possible to transform our designs using the Autodesk account.

Another option would have been the open source 3D software 3Dmax but considering the huge amount of training necessary to use it we decided to revise our design in Tinkercad. This resulted in a complete redesign since the existing designs were too unstable to fix them. The new grid is now designed with fewer components using long rectangles that intersect each other. The gap size in between them is chosen so that the Styrofoam balls can not escape. To enable a stable connection we added bolts to the hat and the connector so that they can locked inside prepared holes in the tube.

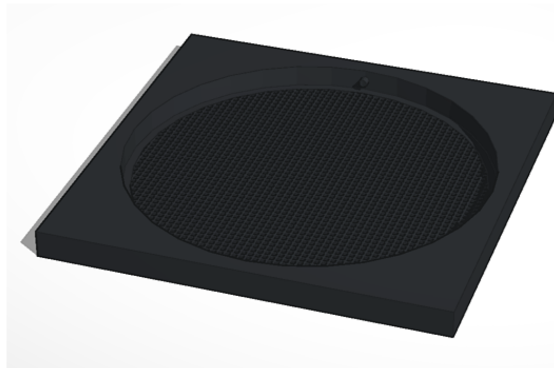


Figure 15: Tube connector

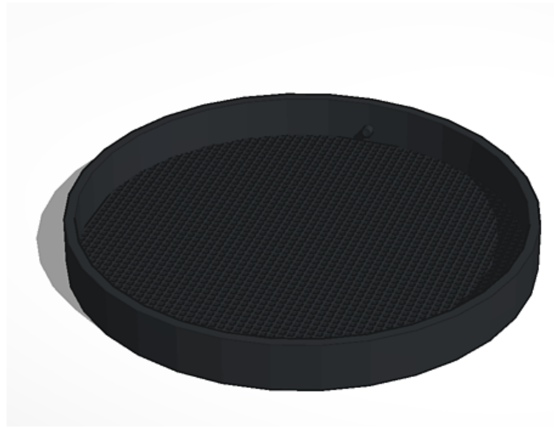


Figure 16: Tube hat

**3D printing the designs**

The printing of the finished designs proved to be quite challenging. According to our professor there is a 3D printer available at our department. We contacted the HFTM which is located inside the BFH estate since the were in possession of an advanced 3D printer. To get a quote and ensure the feasibility we sent in the STL files of our design. After some time one of the employees contacted us since two of the files could not be opened. We visited them in the lab and re-exported our files since they appeared to be damaged during compression. At this point we learned that the air box would take around 40 hours to print, depending on the resolution used. HFTM is in possession of a printer model called "Dimension Elite". This is a really expensive printer that seemed optimal for our prototype since it is able to print larger objects as well.

To compute the time and material necessary the software CatalystEx calculated each resulting layer that will be stacked by the printer.

The two files of the tube that were not opening in CatalystEx took a longer time to calculate which we explained by the fine grid. We agreed on letting it calculate through and the employee assured to contact us once the calculation was done and send us the complete quote containing the prices depending on the printing time.

The day after we got an email from him saying that he could not compute our two tube files. At a rate of CHF 12.-/hour of printing resulted in a price of 510.- up to 770.- just for the airbox, depending on the resolution used. This price was outside our budget and seemed unreasonable since this would be enough to buy our own printer. Furthermore they were not able to print the two key parts of the prototype.

After getting a hint from a classmate we contacted FabLab Bern to get a quote for our parts. The responded with two calculations:

Calculation based on CHF 1.-/minute of printing:

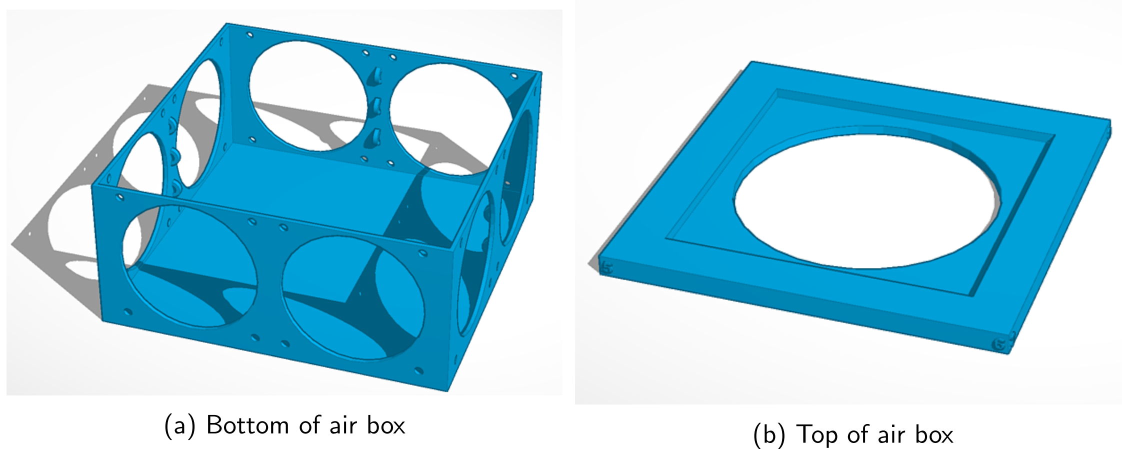
- tube connector: CHF 284.-
- airbox: CHF 916.-
- tube hat: CHF 230.-

Calculation based on CHF 2.-/cm2 of material

- tube connector: CHF 85.20
- airbox: CHF 358.60
- tube hat: CHF 89.30

He mentioned that with the overlaps of the air box and the fine grids of the tube parts the first calculation would be more realistic but also stated that we could buy our own printer for this amount of money. In order to save money he advised us to plan the air box with wooden parts and use a laser cutter to cut them out. This however, would again result in a complete redesign since our models were not made for laser cutting. Therefore we decided to print our models, but make another modification to save printing time.

The new air box is now divided into two separate parts that can be printed individually. This results in the advantage that it does not contain overlaps and therefore saves printing time, since the printer does not have to make supporting structures.



(a) Bottom of air box

(b) Top of air box

Figure 17: Resulting air box split in two parts

The final models were printed by FabLab in Berne.

## 3.4. Electronics

In addition to our structure we used several electronic parts.

- Power source
- Fans
- Light Emmition Diodes (LEDs)
- RaspberryPi 2
- Camera
- Touch screen display
- Speakers

**System overview**
We wanted to be able to control and configure our entire system from within the software. In order to achieve this we needed a way to control the speed of the fans, the brightness of the LEDs and the parameters for the camera.

**Power supply**
We developed the prototype in a way that it only needs one cable to power it.

The system needs both a 12V circuit for the fans as well as a 5V circuit for the RaspberryPi and the LEDs.

We modified a power supply from a router that has both voltages necessary. It is rated at 5V DC, 3A and 12V DC 2A.

This perfectly fits our needs. 5V, 3A is enough to power the RaspberryPi, the LEDs and the display on the 5V rail. 12V, 2A is enough to power all the fans which combine pull around 0.8A.

The power supply has a Remote On Off (ROF) connector that has to be grounded for the supply to work.

The 5V cable is directly connected to the 5V rail of the RaspberryPi which is also connected to the USB 5V line.

The 12V cable is connected to the plus pin of the fans.

Both circuits share the same ground which is inevitable for the fan control to work.

### Fan control

During earlier research we found out that fine tuning the air flow is an essential part of creating turbulence. Because of this reason and because it is hard to predict the necessary amount of air pressure in a changing system we decided to make the fan speeds variable. This comes with another advantage by means of modularity since the air box is not made for one specific tube but rather acts as a source of variable air flow that can be adapted to any part that connects to it.

The fans in our system are driven using Pulse-width modulation (PWM).

PWM is a method for controlling analog components using a digital signal This PWM signal is a square-wave with a fixed frequency that switches between On and Off.

The pulse width is the amount of time that the signal is On in the period given from the frequency. Altering or modulating the pulse width can change the signal between fully On and fully Off.

The ratio between On and Off is called the duty cycle and is commonly noted in percentage.

In most cases the PWM signal is supplied from a micro-controller which can not output high currents. Because of this reason most applications are not powered directly by the signal but rather by using an amplifier in form of a transistor or metal–oxide–semiconductor field-effect transistor (MOSFET) [20].

Varying the pulse width or duty cycle in this specific application results in changing the amount of time that the fan accelerates and therefore controls its speed.

The fans in our prototype are a 4-Pin version that holds two pins for power, one for tachometer and another one for the PWM signal. This has the convenience that they already have a built in amplification.

All PWM outputs are implemented using the RaspberryPi's GPIO pins and as a result we do not need additional periphery.

The GPIO pins are controlled using the PIGPIO library which is implemented in C and hence provides low-level, hardware-timed PWMs on all output pins [21].

Developing the fan controls took as a fair amount of time. At the beginning we considered using Tinkerforge and DC-Bricklets which can produce any voltage in between a certain range. After assembling eight DC-Bricklets however, we quickly realised that this would not be an ideal solution due to the many parts needed and the rather big size of the construct.

The next though was to to use the Arduino board, which contains enough outputs to control all of our electronics. We already did some work with it prior to this thesis. However, we decided to implemented the hardware control using only the RaspberryPi's GPIO.

During this development we tried to use the free Pi4J library that is written in Java. It is based on the WiringPi library that uses software-timed PWMs which are not as precise as hardware-timed PWM but seemed to nevertheless.

Unfortunately during implementation we discovered that PI4J only enables four PWM signals which was not enough for our prototype so we had to look out for another solution.

After some research we discovered the now used PIGPIO library. Because of the somewhat confusing documentation of this library we started to use it through Java Native Access (JNA), which meant for us to read into JNA and the C compilation and linking process. We implemented this native access and wrote the corresponding C code. The resulted implementation worked, but spontaneously crashed because of an unhandled kernel signal 11 in the PIGPIO library, which "is sent to a process when it makes an invalid virtual memory reference, or segmentation fault, i.e. when it performs a segmentation violation" [22].

By reading through several forums we figured that other people were dealing with this specific problem as well and finally discovered the current solution using a daemon and system calls.

After all we learned a lot about JNA and C code compilation which we could put to use in the server application.

**LED control**

We added dimmable LEDs to the prototype to enable a perfect adjustment of illumination inside the tube. With the same principle of PWM LEDs can be modified in brightness.

The LEDs are turned On and Off fast enough so that the human eye perceives it as a constant source of light. The pulse width or amount of time that the led is On determines how bright it will appear.

Different to the fans that already contain amplification circuits, we had to develop an electronic circuit that can power the LEDs without pulling the current from the RaspberryPi.

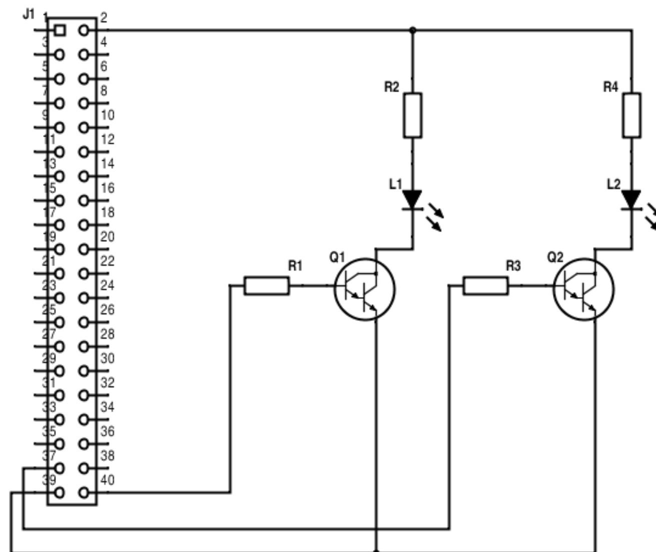



Figure 18: Schematic diagramm of LEDs

We designed a common collector circuit with a BC517 NPN Darlington transistor for each LED. This enables us to dim both LEDs independent from each other.

The LEDs L1 and L2 get the power from the RaspberryPi 5V power rail at pin 2, which is directly connected to the power source. Ground at pin 39 is directly to the emitter of the transistors Q1 and Q2.

The GPIO pins of the RaspberryPi are connected to the base of the transistors in serial with the resistor R1 and R2.

If the GPIO pin on the RaspberryPi is high (3.3V) a current flows from the base to the emitter of the transistor and as a result lets the current flow from the collector to the emitter which powers on the LED.

This way we can control a bigger amount of current with a small current from the RaspberryPi.

**Touch display**

The application on the prototype can be controlled using the 7 inch HDMI touch screen PI-ADA-2396 from Adafruit. It has a resolution of 1024x600 and includes the drivers, touch controller and buttons to configure the display. It is connected to the RaspberryPi using a HDMI cable for the signal and powered through a USB cable which connects it to the RaspberryPi's 5V rail. The touch controller is connected via USB.

This display works almost out of the box, we only had to configure the resolution.

At first attempt we ordered the 5 inch HDMI backpack touch screen PI-ADA-2260 from Adafruit but could not get the touch screen to work. Even after configuring, recalibrating, changing its behaviour from mouse to touch and installing manufacturer calibration software the coordinates were still wrong and the display not usable as a touch screen.

**GPIO pin-out**
The following table shows the Broadcom pin numbers of the RaspberryPi GPIO and the devices connected to it. The fan numbers are the ones printed on the prototype air box.

| *pin* | *device* |
|---|---|
| 4 | fan 3 |
| 5 | fan 6 |
| 6 | fan 7 |
| 13 | fan 8 |
| 17 | fan 4 |
| 19 | fan 1 |
| 21 | led 1 |
| 22 | fan 2 |
| 26 | led 2 |
| 27 | fan 5 |

## 3.5. Software overview

We wanted our software to be reusable, easily adaptable and extensible. Wherever possible, we used state-of-the-art software design patterns. For several functions we relied on open source libraries such as the many useful classes from Apache commons. This has the big advantage that the code is ample tested and maintained.

The software can be divided into two separate parts, the prototype application and the server application.

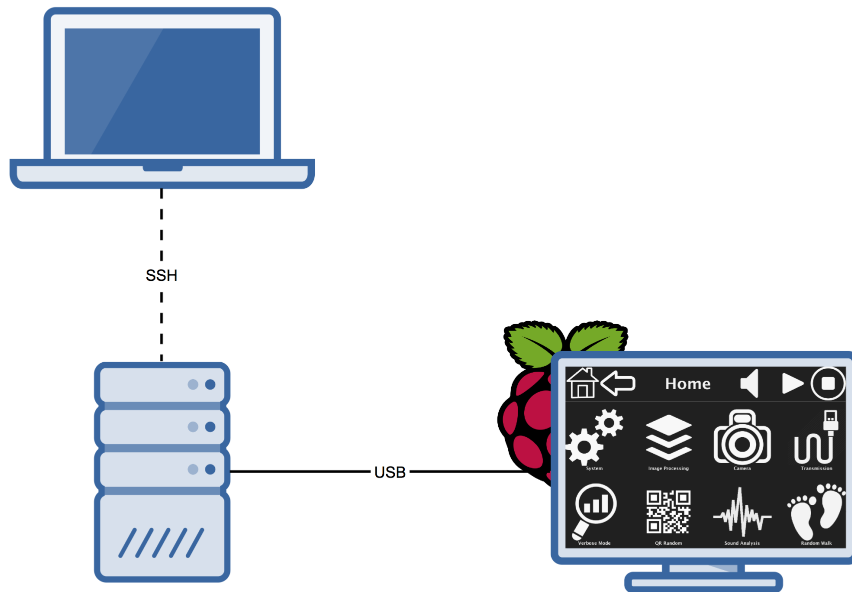



Figure 19: Interaction between applications

The prototype application implements a graphical user interface where the administrator can control the system and use our application without the need of a network connection.

The server application runs on the target server in command line mode. It does not implement any graphical interface and can only be accessed through command line inputs. As shown in 19 Interaction between applications the server application can be started and controlled through an SSH connection to the host server.

The two applications interact with a serial connection using a USB cable.

Note that the UML diagrams are not complete but rather contain the information that is necessary to understand the software architecture.

We would like to note a few things in order to understand the software documentation:

We used colors to exemplify the accentuate the different types.

- Blue: Superclasses or classes with abstract methods
- Yellow: Interfaces
- Green: Existing classes from libraries
- White: Regular classes, subclasses or Enums

The software elements contain a unified naming: Controllers and Views from the graphical user interface are named with "...Ctrl" and "...View" Controllers that are not part of the GUI are named "...Control"

In the following chapter class names are written as they are in Java and are therefore capitalized.

We divided the software into different packages, whereas the most part of the user interface is inside the truerandom.userinterface package, with the exception of specific implementations that provide an own view to avoid coupling.

## 3.6. Prototype Application

The prototype application enables the user to configure, administrate and analyze the generation of random numbers. Furthermore it provides a safe channel to exchange the inital secret used for the safe transmission of the random data. It is implemented in a way that even an inexperienced user can understand and configure the prototype.

The application is written in Java and uses native libraries that are written in C. The following chapter explains the different parts of the application.

### 3.6.1. Main

The Main class is the entry point of the application. It is the link for classes that are used by several different other classes as well as classes that implement a central functionality such as the transmission.

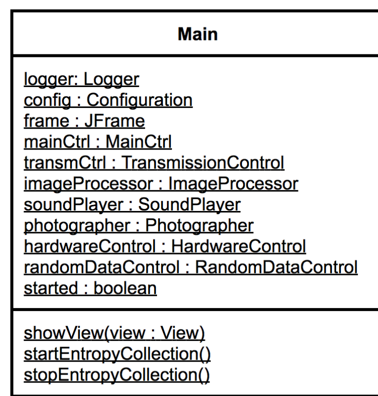



Figure 20: Main class

It holds instances of the logger, the configuration, the frame, the main control of the graphical user face, the instances used for transmission, the sound player, the photographer, the hardware control and the random data control. The Main class provides methods for starting and stopping the process of entropy collection.

The configuration of the generator can be easily adjusted using the GUI, without having to use the command line.

### 3.6.2. Graphical User Interface

The graphical user interface is implemented using the architectural software pattern mode view controller (MVC). This enforces a unified implementation of the user interface throughout our software.

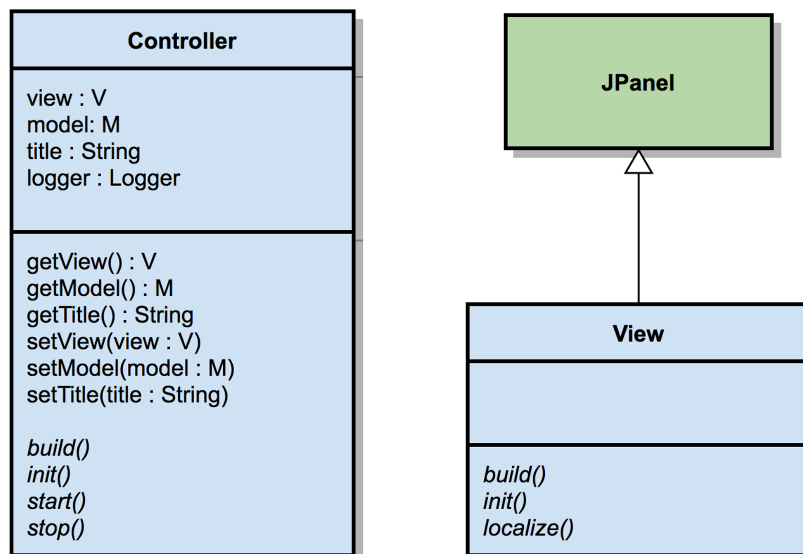



Figure 21: MVC UML

The two super-classes View and Controller are extended by the corresponding implementations.

The Controller class implements the basic functionality and methods like getters and setters for the view, the model and the title. Furthermore it can hold a logger.

The abstract methods build(), init(), start() and stop() are abstract and have to be implemented by the subclass. The constructor of the Controller calls the methods build() and

init() at instantiation.

The View class extends JPanel so it already is a Java Component and therefore can be directly added to a layout. In addition, the View already hold an instance of CellConstraint which we used for creating the layouts. Our Views in the application are mostly designed using the JGoodies FormLayout which provides an easy way to arrange different components [12].

More information about the Model-View-Controller pattern can be found online [11].

To unify the appearance of our application we implemented two classes, Cupboard and GUITools. Cupboard holds reusable things such as sizes, ratios and custom colors.

GUITools provides methods that automatically configure components such as buttons and sliders. Since our application is made to run on a touch screen, we designed buttons as icons. To configure a button the only thing necessary is to set its name. GUITools automatically changes its appearance and loads the image icon from the resource path.

### 3.6.3. Logging

Our logging implementation is done using Apache's log4j2 library, whereas the configuration of the Logger is made in the log4j2.xml file in the resource path. With help of two separate Appenders for the console and the file output we configured our software to output different logging levels for the file and the console output.

In short, the console output informs the user about general system actions without going into too much detail. The log-file contains verbose outputs and more detailed messages to investigate the behaviour of the software.

The log-files are written in the directory where the jar is executed, with the file-name "truerandom.log"

### 3.6.4. Configuration

Since we wanted to make our prototype easily configurable we needed a solid way to store and retrieve configuration data.

For this reason we created the Configuration class that holds an instance of Apache Common's PropertiesConfiguration.

For each configuration, the class holds a key, a default value, a getter and a setter method. With PropertiesConfiguration it is possible to store every possible type as a property value,

even an object.

When accessing a getter, both the key and the default value is passed as an argument. If no configuration has been made yet, the default value is automatically written into the configuration.

Instead of having to provide an initial configuration file for the application to work, default values can be set and the file will be generated on the fly.

Furthermore this centralises the methods, values and keys of the configuration and makes the software easier to understand and extend.

### 3.6.5. System / Hardware Access

The GUI enables the user to configure hardware system states such as the LED brightness and the individual fan speeds.

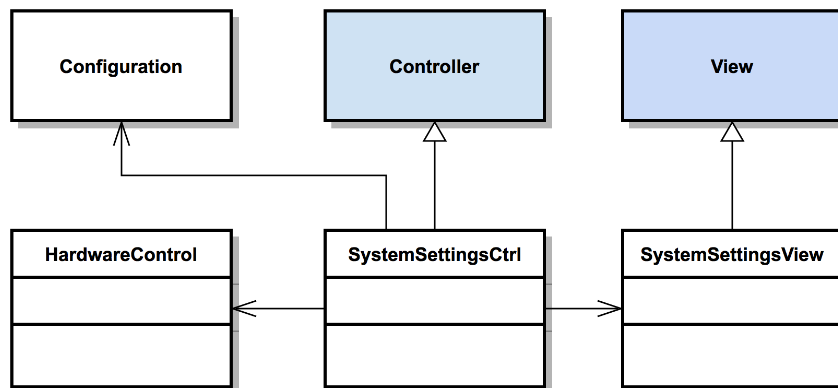



Figure 22: Hardware control UML

The class HardwareControl, which is instantiated in the Main class holds two ArrayLists for the LEDs and the fans which are filled with Integers of the Pin number on the RaspbberPi's GPIO port. The pin numbers are the numbers given by the Broadcom chip on the Raspberry. Publicly, it provides methods to set the fan speeds, led brightness and to turn all the hardware On or Off whereas it gets the configured values from the Configuration in the Main class. Privately it uses methods to configure PWM specific things on the GPIO pins such as the range, the frequency and the duty-cycle.

The actual calls to set the pin behaviour is done using the PIGPIO-daemon called PIGS which is described in [**?**]. HardwareControl starts PIGS at instantiation and sets off system commands for the appropriate commands.

The SystemSettingsCtrl uses a Configuration as its model. SystemSettingsView provides a user interface to configure fan speeds and led brightnesses using sliders. It automatically updates the values in its Configuration model and if the hardware is started it updates the pins as well.

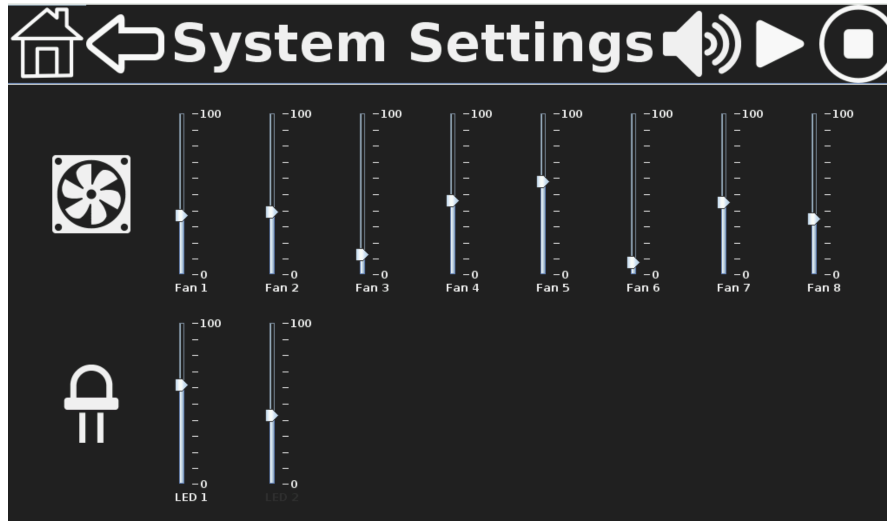



Figure 23: System settings view

### 3.6.6. Sound output and alarming

In order to give better user feedback in combination with the feedback we implemented the SoundPlayer class. It provides methods to output several system sounds that are loaded from the resource path as an AudioInputStream and played as a Clip.

Our prototype contains an alarming function that is set off when it remains in the offline state for too long. The alarming loop is implemented in the SoundPlayer as well.

### 3.6.7. Camera

The PiNoir camera can be comfortably accessed using the system commands raspistill and raspivid. In our case, adjustable camera setting are a vital part of good picture quality.

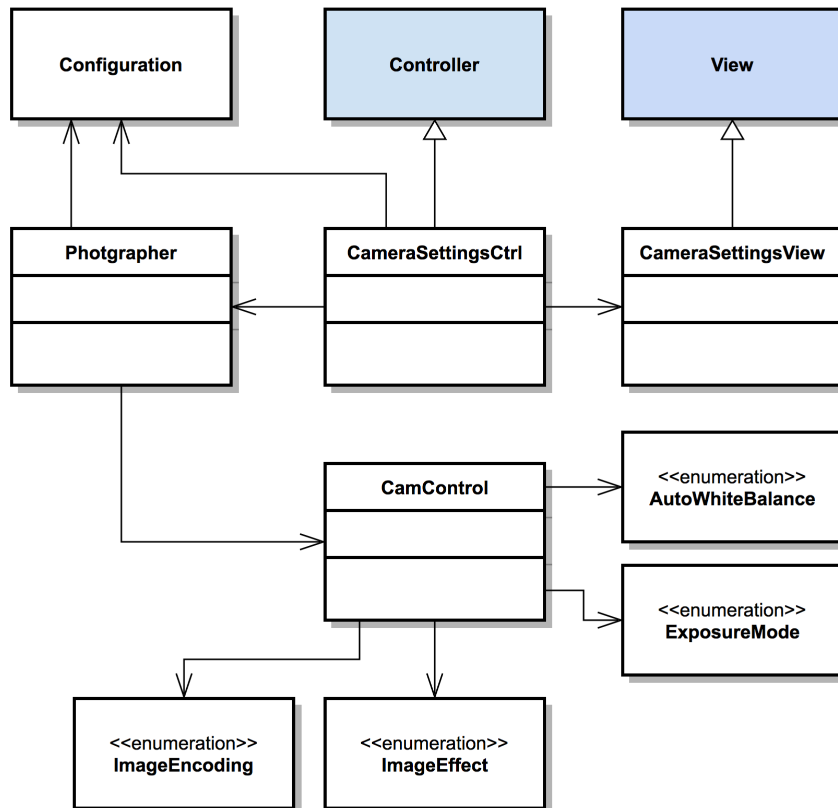



Figure 24: Camera UML

We implemented the CamControl class that allows to take pictures and videos with every possible command and options provided for the raspberry camera. This class can be reused later for a lot of different applications not just in our specific project and therefore comes in handy. The different modes for exposure, white-balance, image-effects and encodings are implemented using Enums.

The application demands a class that takes pictures with configured settings in a defined interval.This is represented by the Photographer class. The Photographer has an instance of the CamControl and provides methods to take pictures in regular intervals.

The Photographer gets its camera settings from the Configuration file and has a method to update its settings from the configuration

To make the configuration available to the end user we implemented the CameraSettingsC-trl and the appendant CameraSettingsView. This view enables configurations of camera parameters that are automatically stored in the configuration. When camera settings have changed, the Photographer gets told to update its settings.

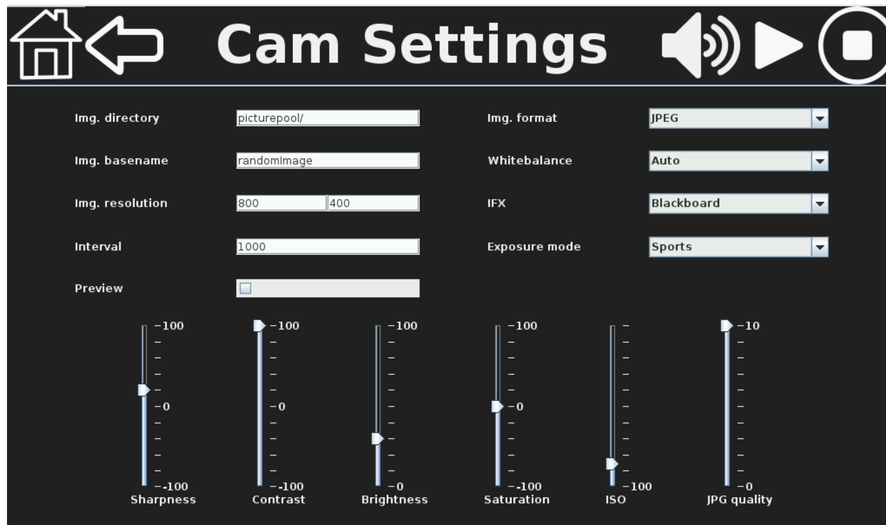



Figure 25: Camera settings view

### 3.6.8. Random Data Control

Instead of directly linking together classes and to avoid coupling we implemented an observer pattern for random data.

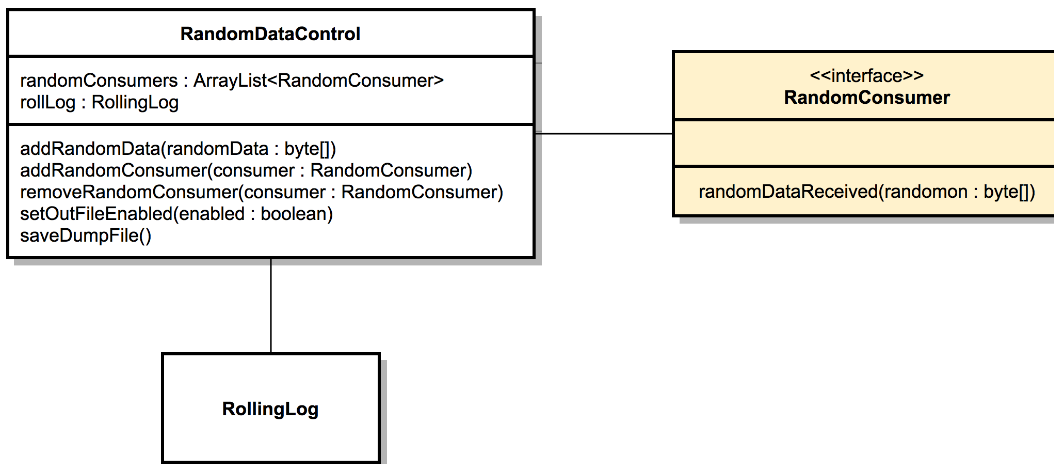



Figure 26: DataControl UML

All random data is fed to the RandomDataControl class. It is responsible to format and distribute the data.

Classes that are interested in the random data can implement the RandomConsumer interface and then subscribe for random data in the RandomDataControl class. Accordingly they can unsubscribe if they no longer wish to receive it.

RandomDataControl also contains a RollingLog, which is a class that we implemented to maintain a log file that is limited in the number of lines. It only needs a file, the maximum number of lines and a boolean whether it should write the file after every change or if the writing is initiated by the owner. If no maximum number is set, it uses a default value of 500 lines. Optionally a Logger can be set so that occurring errors are logged.

This output file of the RollingLog is an important part of the verification process since it is used to compare the data on the server with the one on the prototype.

### 3.6.9. Testing tools

As mentioned in 2.6.2 Human perception, a good overview over random data can be obtained by using methods that involve the human perception.

To make this form of testing available to the user, we implemented a random walk as well as a mode to analyze the sound output.

#### Random walk

This is the implementation of the principle explained in 2.6.2 Human perception.

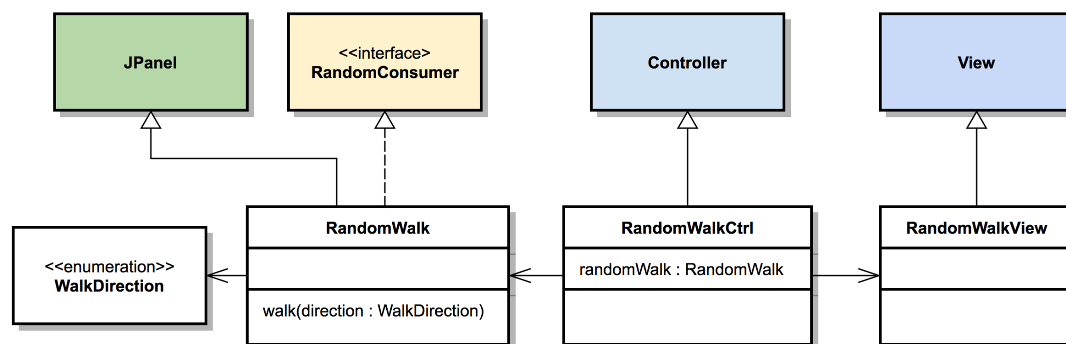



Figure 27: Random walk UML

Analogue to all other views the controller and view extend our abstract MVC classes. The RandomWalkView provides short instructions and buttons to start, stop and refresh the random walk.

We implemented the class RandomWalk which extends a JPanel. It holds an inner Enum with

the possible walking directions. RandomWalk implements our RandomConsumer interface and is subscribed to RandomDataControl as soon as the user touches the play button. Simultaneously it is unsubscribe when stop is touched.

Each time the RandomWalk receives random data, it interprets the data and chooses a walking direction based on it. At first we implemented a bitwise interpretation that takes two bits and chooses the direction based on the value of the two bits (0, 1, 2 or 3). Since this resulted in a lot of steps for a single image, we implemented another interpretation that compares the value of a whole byte to determine the direction.

To perform a step, the walk method is called and a WalkDirection is passed as an argument. The actual Component is a Path2D.Double that is initialized in the center and then draws a line from the last point to the new location.

Figure 28: Random walk view

### Sound analysis

This is the implementation of the principle explained in2.6.2 Human perception.

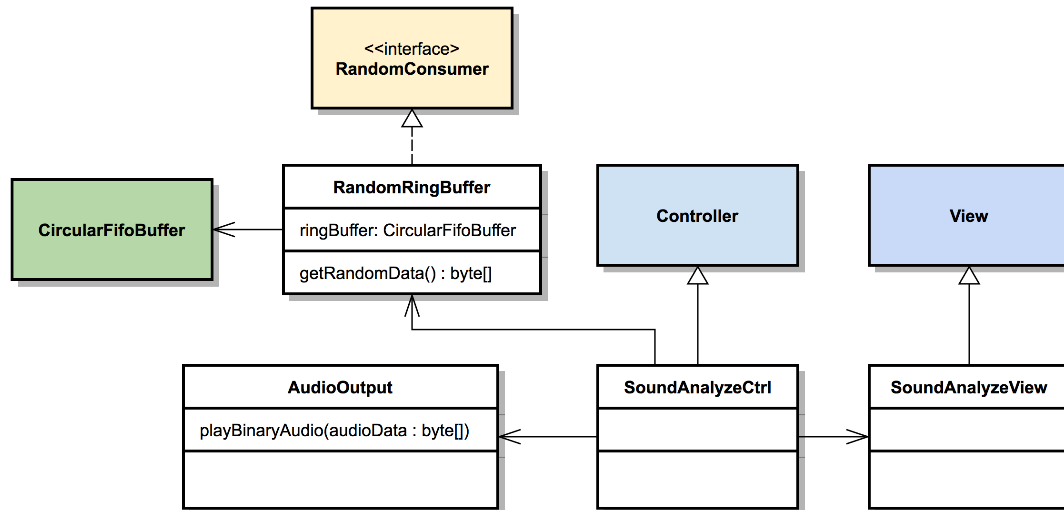



Figure 29: Sound analysis UML

SoundAnalyzeCtrl holds a RandomRingBuffer, which we implemented using a CircularFifoBuffer from Apache Commons.

The RandomRingBuffer implements a RandomConsumer and can therefore easily be subscribed to the RandomDataControl where it gets the random data from. This way it is automatically refreshed with the latest random data from our prototype.

To stream the binary data and play it on an audio device we created the class AudioOutput. It provides methods to play a byte array to the audio device as Pule-code modulation (PCM), which is a method to digitally represent analog signals [18].

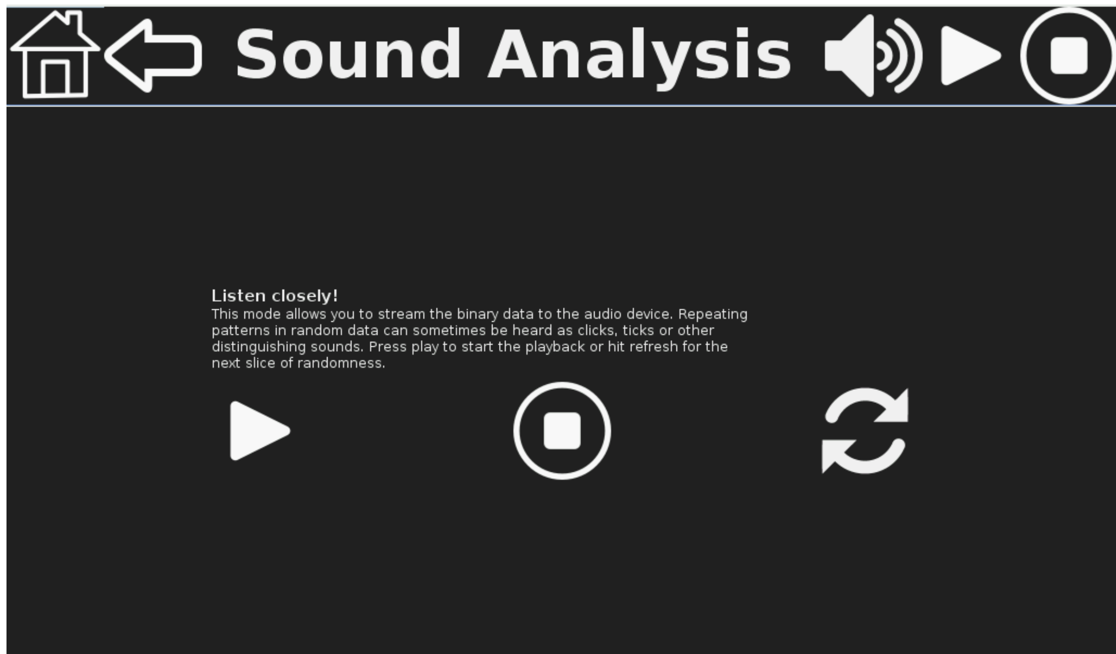



Figure 30: Sound analyze view

**Quick Response (QR) code**

In the earlier stage of development we had the idea of delivering an initial seed in form of a QR code and therefore implemented a view where the user can get a QR-code with random numbers.

As it turned out later we embedded this functionality in the transmission setup procedure. In the TransmissionCtrl the key required for authentication is displayed as text as well as a QR-code in case that the generator and the server console are too far apart. This way the user can scan the code, walk to the server console and type in the key.

We decided to keep the initial seed view but re-factored it so it became a neat feature where the user can get a random number for any purpose.

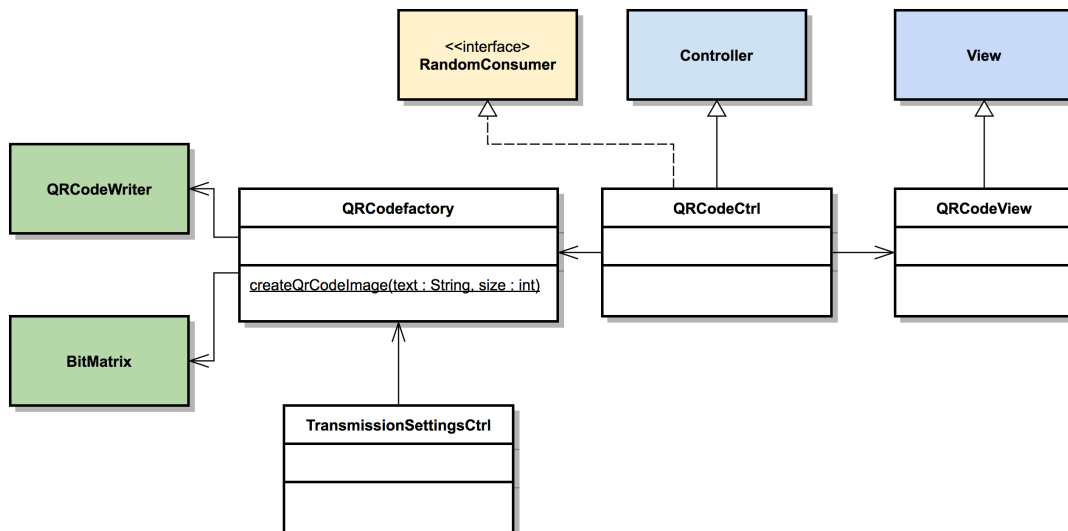



Figure 31: QR code UML

As already mentioned, the QRCodeView displays a QR-code with random data to the user. Therefore the QRCodeCtrl implements a RandomConsumer and subscribes itself for random data.

To generate the actual QR-code image, we made us of the factory design pattern and created a QRCodeFactory that can be reused for other projects.

The QRCodeFactory makes use of the Zebra Crossing (Zxing) library and the two classes QRCodeWriter and BitMatrix.

To generate a QR code, the String is encoded into a matrix that contains the bits. When encoding, an error correction map is needed. This map contains an error correction level that indicates how much of the destroyed data may be corrected when the QR code is not fully readable.

After encoding, the resulting BitMatrix holds the information whether or not a specific field is filled or not.

by iterating through the BitMatrix and filling rectangles, the QR code can be drawn as a BufferedImage.

More information about the principle or QR codes can be found on the here [19].

Figure 32: Random QR view

The same QRCodefactory is used by the TransmissionSettingsCtrl.

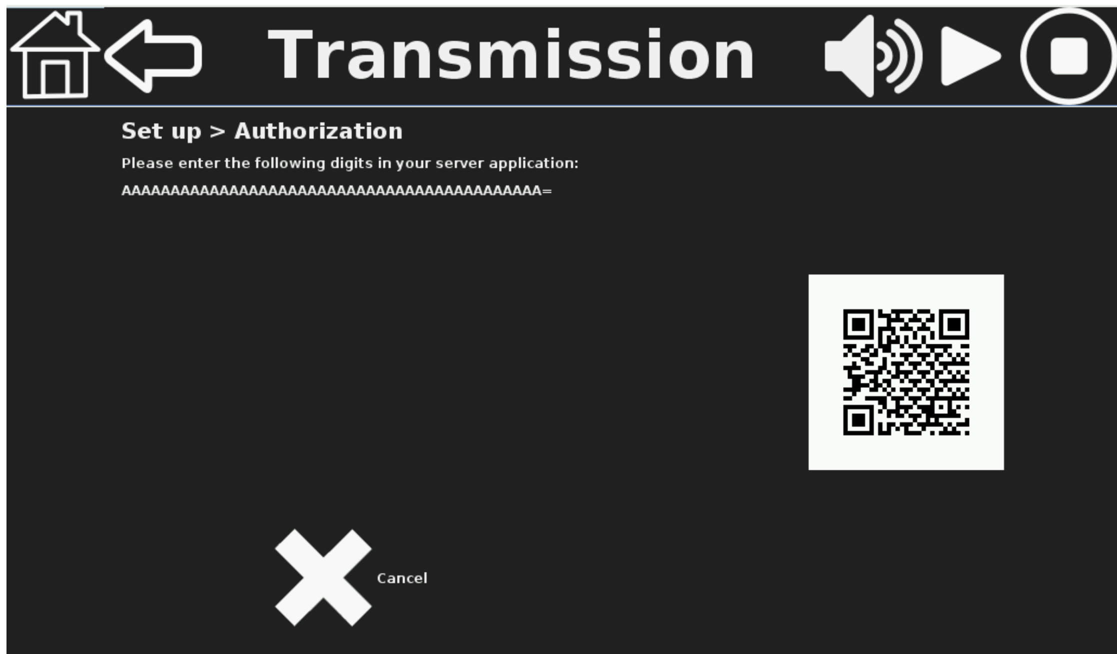



Figure 33: QR code in transmission settings view

### 3.6.10. Image Processing

The image processing is organized using the strategy pattern [8]. That way, we can easily switch between different image processing strategies and are not limited by one strategy. Besides of keeping the application open for further development, this decision was also based on the fact that we do not possess in depth knowledge of image processing.

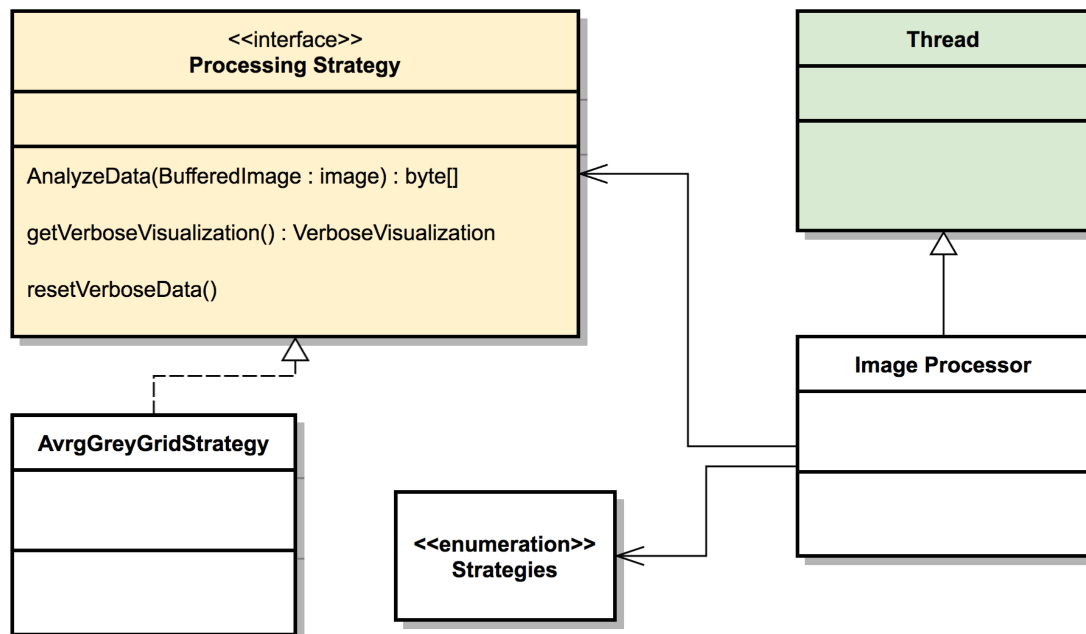



Figure 34: Image processing strategy setup

The image processing and the distribution of the resulted values is managed by the class image processor. To enable multi-threading, the class implements the standard Java Thread class. Which strategy should be used, can be defined in our configuration file and is mapped to a corresponding Enumeration class Strategies.

In our prototype application we implemented only one really trivial image processing strategy, the average grey-scale grid strategy. This strategy simply appends a grid to the given pictures and measures the average grey-scale value of the different grid fields.

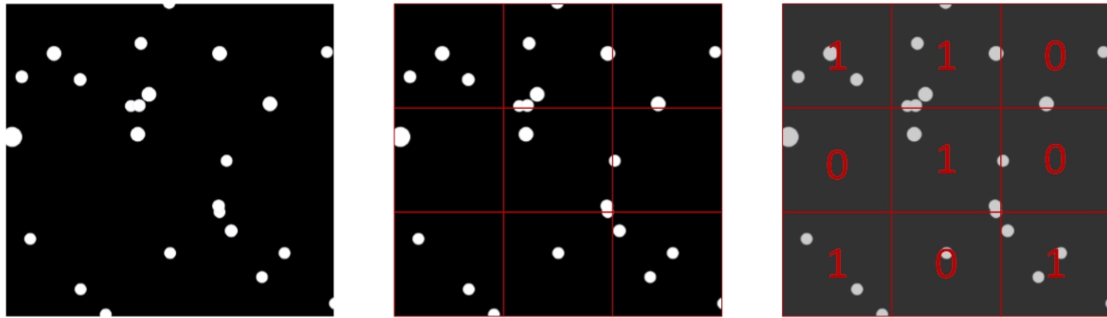



Figure 35: Average grey-scale grid strategy

If the average of a field exceeds a certain threshold we derive a bit of the value 1, if it equals or undercuts the threshold we derive a bit of the value 0. Whereas the threshold and the size of the grid are parameters definable in our configuration.

### Image Processing Verbose Views

Since humans are are way better in detecting patterns than computers are, it was evident for us that a visualization of the gathered data would be a great benefit for the testing and the verification of the random data generation. Therefore we decided to implement a so called verbose view, which displays the average values gathered by the image processing. Because future image processing strategies may vary in the way of how they analyze the provided images, we had to assume that the visualization will differ depending on which strategy is used. Therefore we extended the strategy pattern with the previously described verbose view.

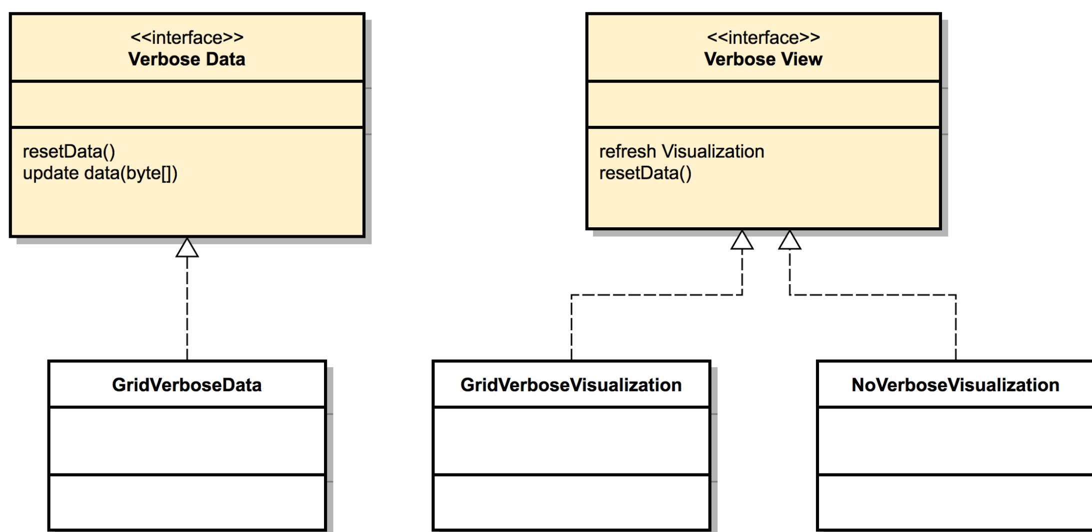



Figure 36: Image processing strategy setup

Hence every image processing strategy also must provide a verbose view / visualization. To enable an easier refresh of those views, we also implemented a corresponding verbose model class (VerboseData) containing the displayed verbose information. Therefore a strategy must only update the corresponding verbose model to update the data displayed on the view. This corresponds to the model view controller pattern used on the graphical user interface.

In our implementation, the visualization of the average grey scale grid strategy simply consists of the average value derived from the different fields. If a average value tends towards the value 0 or 1, it is obvious that this specific field is not completely unbiased. To underline this fact, we added a colour transition from green to red, whereas green implies a better average value (e.g. 0.5) and red implies biased bits (e.g. an average of 0.98). Of course this depends strongly on the time and the set up of the measurement.

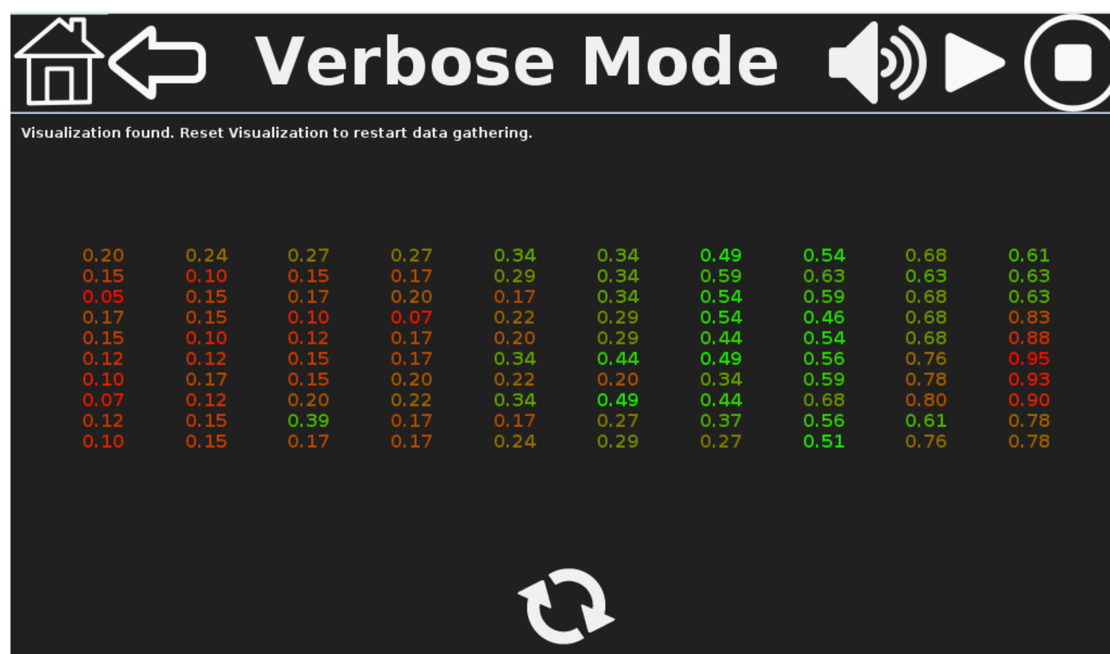



Figure 37: Verbose visualization on the user interface

For example a five minute measurement will not give great hints about how reliable the random numbers are. But if a system administrators takes a look at this view every time he passes by the generator, he might discovers that some field are always tending towards 0 or 1. He can therefore easily discover if the generation of the random numbers is flawed and start further investigations.

## 3.7. Server Application

The main tasks of the server application is to interact with the incoming transmission. The server must properly authenticate and receive transmitted data. This includes answering with an acknowledgement packet for each data packet received. After that the server application must forward the random data to the corresponding target application.

The two main features of the server application, transmission and seeding, are handled in the corresponding controller classes. Most classes are based on the generator setup.

The main difference apart from seeding and the transmission behaviour, is that the whole server application is command line based.

### 3.7.1. User interface

Since it is highly unlikely that a server possesses a graphical user interface, we decided to base the server application solely on command line based arguments. The interaction with the user is supposed to be limited to the authentication during the start up procedure. All interactions with the user are managed in the class CLIControl.

The reason why we need to have an interaction with the user, is to retrieve the USB device and to exchange the secret for encryption. Once the transmission as well as the seeding is running, there is no more need for the user interaction. For now, our application solely informs the user about the success of the transmission setup and keeps running. For further versions it might be a good idea to release the InputScanner and run the application as a background process. This setting however was entirely sufficient to fulfil our requirements, furthermore using the GNU screen manager the current application could also be started in a separate session.

### 3.7.2. Seeding Strategies

Since we wanted to keep our application architecture as open as possible, we implemented the strategy pattern which we already used for the image processing on the generator. Therefore every seeding target is implemented as a Seeder class which implements the SeederStrategy interface.

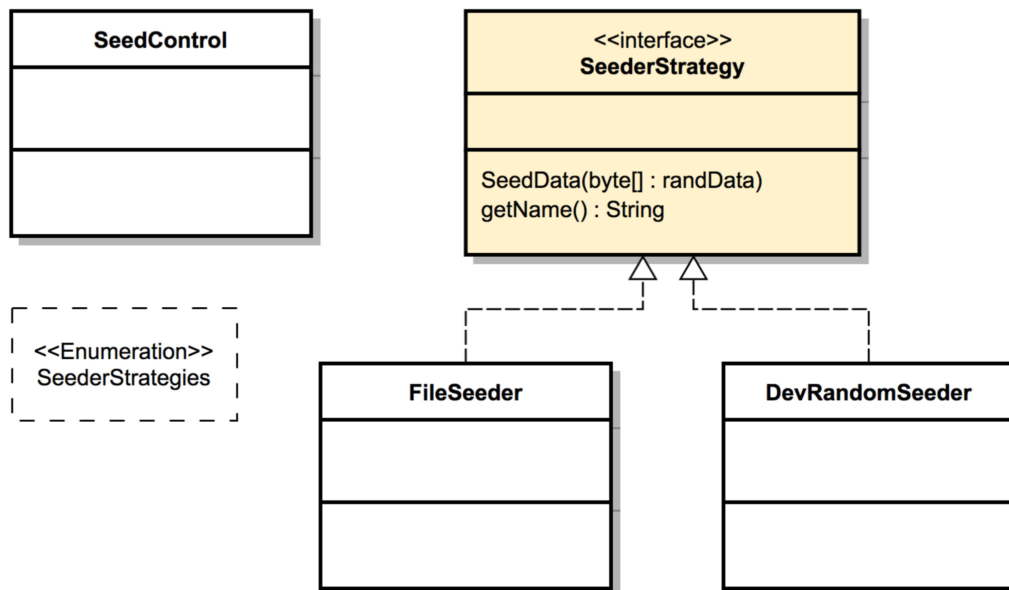



Figure 38: Setup Strategy Pattern for Seeding

This enables a swift and easy addition of new seeding targets. One can simply add a new seeder class and add the corresponding identifier to the configuration as well as to the switch statement in the class SeedControl. In the future the switch statement could be replaced by a corresponding enumeration class as we have done it on the generator application.

**Seeding /dev/random**

As we previously introduced, one of the main functionalities of our server application is seeding into /dev/random. The underlying construct of /dev/random, the Fortuna PRNG has already been discussed in detail in the conception. To seed the /dev/random construct, there is a predefined IOCTL call (RNDADDENTROPY):

```
struct rand_pool_info {
                    int    entropy_count;
                    int    buf_size;
                    __u32  buf[0];
                };
```

Since this a kernel level system call, we were not able to access it directly with our java application We had to write a corresponding small C instruction, which will get called by our java program using the Java Native Interface (JNI).

**Seeding To File**

To enable easier testing and to show that our generator could also be used to provide entropy for other services, we also implemented a Seed to File strategy.

It is a quite trivial approach which is based on the Java standard class FileOutputStream. To avoid overflows and enable long term tests, we created a manual rolling feature by counting the lines and creating a new output file once the predefined maximal length is reached.

The parameters such as the file location and max length of a output file are defined in the configuration /properties file).

## 3.8. Data Transmission

According to the definition of the transmission concept, we decided to describe the transmission using the states DOWN, AUTHORIZATION, UP and OFFLINE. For a more detailed description on what actions are taken during the different states, see appendix: A Transmission Flow.

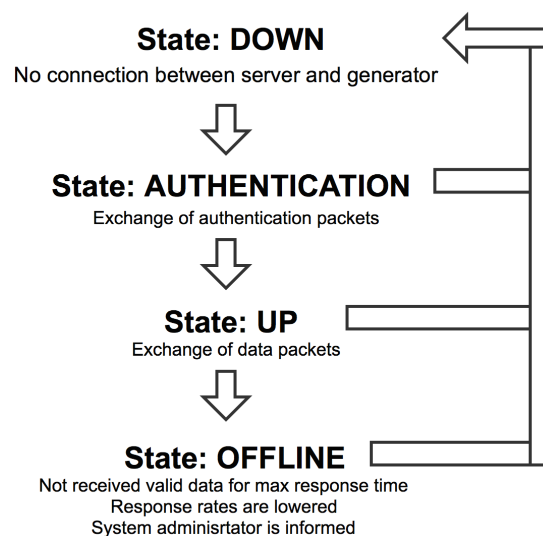


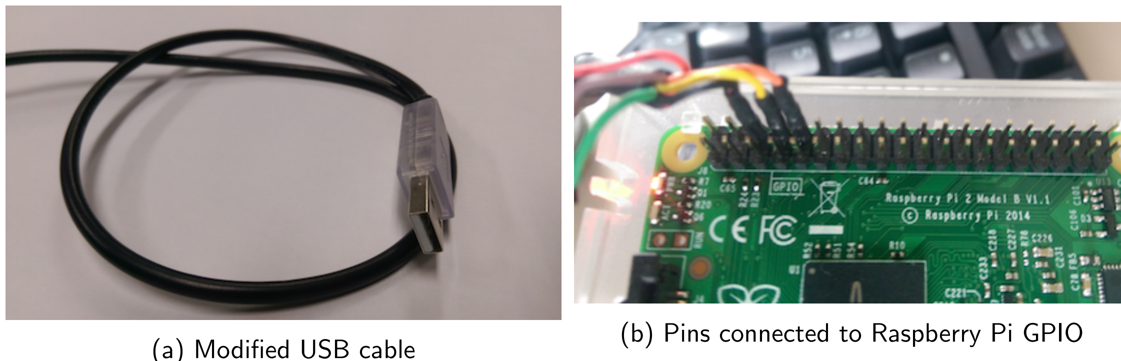

Figure 39: Transmission states

The main functionality of the software is to react to To make our transmission implementation as flexible as possible, we included key values such as response times in our configuration file. Hence the transmission rate can be adjusted according the demands of the generator (e.g. if the generator is able to provide more entropy due to a better configuration).

### 3.8.1. Hardware Implementation

At the start of the project, we ordered a simple peer to peer USB cable to connect the Raspberry Pi to the server. The declarations of most USB cables were not that detailed and we did not encounter a lot of corresponding setups during our research. It turned out that the cable was not crossed, which it should have been to enable a peer to peer USB connection. Luckily we were able to obtain a corresponding crossed USB cable from our electronics division.

We directly attached the USB pins to the Raspberry Pi GPIO pins and connected the other side of the cable to the server.




(a) Modified USB cable

(b) Pins connected to Raspberry Pi GPIO

Figure 40: Setup of the USB cable

The device was immediately detected by the Raspbian operating system via the Advanced Micro-controller Bus Architecture (AMBA) and displayed as device ttyAMA0. Once connected to the server, we were able to exchange data using simple shell commands. This enabled us to go a step further and implement a corresponding application in Java.

### 3.8.2. Software Implementation

First we tried to set up the transmission using a quite primitive approach. We simply executed the echo command to write to the corresponding Unix devices (ttyAMA0, ttyUSB*) and used a BufferedReader to read from the devices. We observed that during the read process, the machine on the other end was constantly receiving blank lines as an input. This then led to a deadlock, since sometimes both sides are supposed to listening / read from the device.

We therefore decided to look for a cleaner solution to manage the transmission. Since the few Java based solutions we stumbled upon (e.g. usb4java[9]) were too complex and not fully applicable to our situation, we decided to build a trivial serial connection. We chose the rxtx library [10] to do so, because we have been using it in a previous project and it turned

out to be simple and robust.

All classes that are used for the transmission are located in the package truerandom.transmission, respectively truerandomserver.transmission. The main features are represented by the TransmissionControl class, which is taking actions accordingly to the current state of the transmission. For more information on the exact transmission flow, please consider appendix A Transmission Flow.

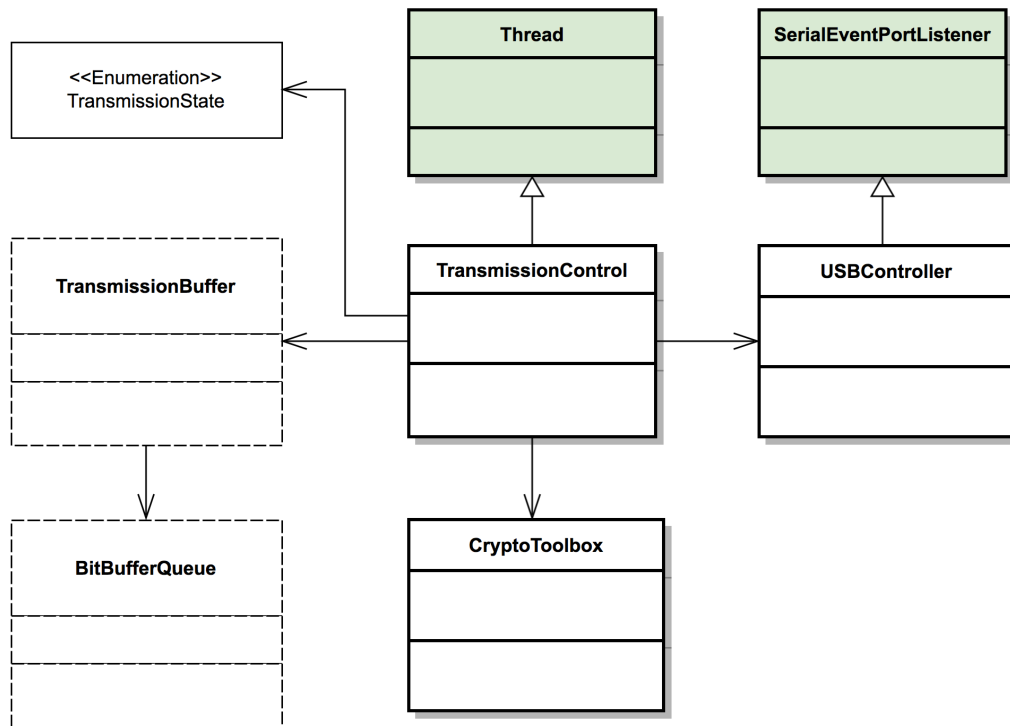



Figure 41: Setup Transmission

To describe the different states, we created a corresponding enumeration class TransmissionState. The actions concerning the set up of the serial connection as well as sending and receiving of the transmitted data are represented by the USBControl class. Lastly, there is the CryptoToolbox which provides cryptographic functions such as SHA512 hashing and AES encryption. The CryptoToolbox currently handles all exceptions directly. In a future version these should be passed on to the upper classes, which would make it possible to include the CryptoToolbox in a commons library.

On the generator, we still use a so called TransmissionBuffer class and a BitBufferQueue class. Theses classes originated during the beginning of the project, before we applied the data control system (see section 3.6.8 Random Data Control).

It is to say, that apart from the TransmissionControl all transmission related classes include almost the same features on both server an generator, hence they have both been described in this section.

## 3.9. Test Procedure (Challenge Response)

We added a corresponding opening to our prototype to make sure that the insertion of a test module is physically possible.

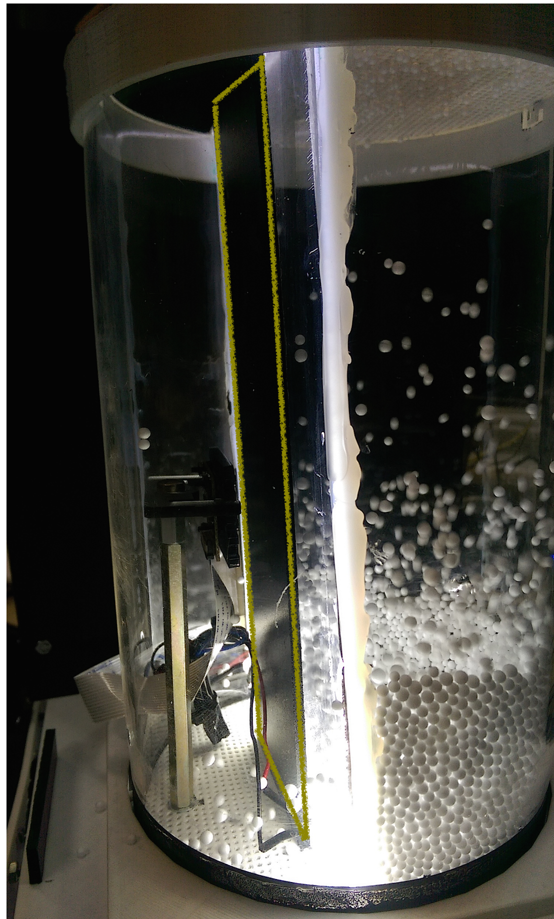


Figure 42: Insertion of a test module

As we can see in the figure, the test module neatly covers the motive of the camera and therefore static output has to be expected on both the generator and the server. Once inserted, we have various possibilities to verify the random numbers generated during the test procedure.

The most trivial attempt would be to analyse the output of the RollingBuffer. By comparing the raw random bits on both server and generator with the expected test values, the whole generation process can be verified. Although this is the safest way, it is not really user-friendly since every bit has to be compared manually. For future version one might try to create a corresponding log parser, who automatically compares the log files of server and generator for a more accurate comparison.

As we described in our concept (see section 2.6 Testing randomness), humans are way better in detecting patterns than computers are. With our additional features, one can us a more pleasant way to verify the current output. For example by taking a look at the verbose view.

(a) Verbose view during test

(b) Verbose view normal runtime

Figure 43: Using the Verbose view for the testing procedure

It is to say that the verbose view is not always a good option, since if the generator has been running for some time it takes extremely long to see the changes happening on the visualization. Another way would be to play the random bits via speaker using our sound buffer feature or by taking a look at the random walk feature. During our tests however, the verbose view turned out to be the most intuitive way to get information about the currently produced random numbers.

# 4. Conclusion

## 4.1. Progress and Time Management

Until we started with the development phase, we were able to keep to our project schedule and achieve the milestones defined during the planning phase of our thesis. But once we started with the implementation we quickly realized that we had to restructure our project planning.

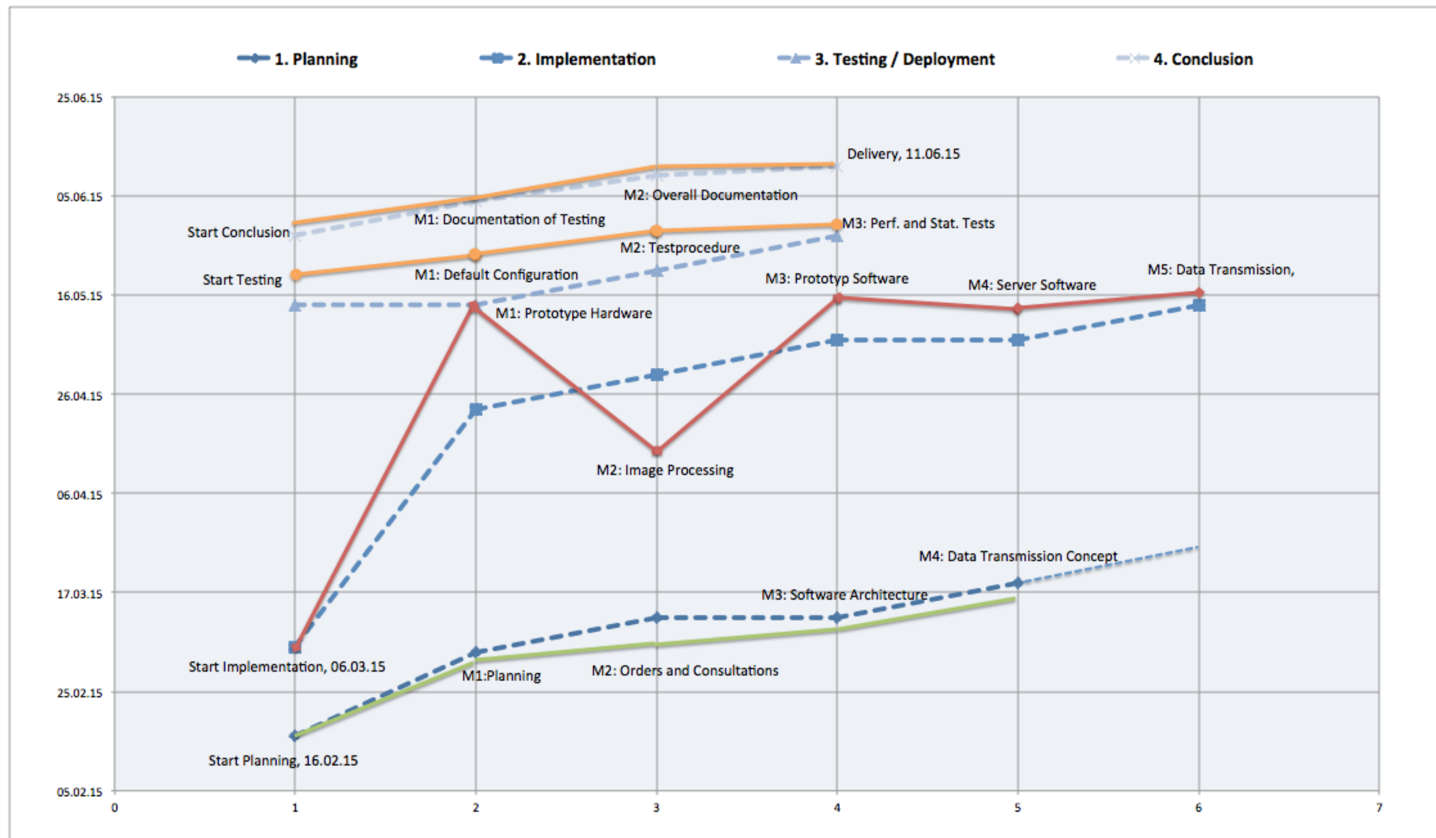



Figure 44: Actual project schedule

The main reason was that we encountered a lot of issues that required more attention and provided us with a lot more questions than we expected. For one thing, the physical prototype included a lot more planning, clarifications. Especially the 3D prints caused a big delay and urged us to rethink the real goal of our generator prototype.

The main reason for this delay was because we realized that we are going to create a prototype that is fixated on one specific random phenomenon. We then decided to lay a bigger focus on the prototype and the application in order to build it a open, more general applicable way. Thus if our specific phenomenon turns out to be either incapable of providing enough entropy or if we fail due to mechanical aspects, our concept and software could be applied to any other random phenomena that is detectable by a camera.

## 4.2. Open Issues

As our project went on, we steadily discovered parts of our software that could be improved as well as possible new features or unfavorable implementations. To make sure that these open issues are not lost track of in the future development, we decided to point them out in this chapter.

**Safe shutdown of the server application**
The server application should be implemented as a background process to enable a better handling. This includes the ability for a user to safely shutdown the service at any time.

**Verbose view reset button**
Currently the reset action is called when the lower part of the verbose view is clicked, rather than just if the button is clicked. This is a rather specific detail that can be resolved easily.

**Adjustment of the random walk system**
Currently the orientation of the random walk is based on an int value in between 127 and -127. Hence the value of one byte. This proved to be a problem since higher bits are influencing the outcome a lot stronger than the smaller bits are. In a future version we would create a new concept that combines the two implementations of bit-wise and byte-wise interpretation.

**JUnit tests in further development**
During our development phase we did not use JUnit test classes for in depth testing. Although we did not face any problems due to that, we suggest to include JUnit testing into further development processes. This is mainly due to the fact that the application steadily grows and deploying the application to the prototype might become time consuming.

**Refactoring of TranmissionBuffer / BitBufferQueue**
Due to the DataControl concept, the classes TransmissionBuffer and BitBufferQueue on the generator application could be shrinked to one instance, maybe even included in the TransmissionController.

## 4.3. Prospect

Even though we initially focused on a specific phenomenon, we recognized that we needed to stop focusing on a single approach and developed a powerful platform for randomness extraction and testing. During our thesis we built a solid prototype that is open for a lot of options and an ideal platform to continue the development of true random number generators.

We sincerely hope that our prototype will be developed further and that its abilities to adapt to many other phenomena will be recognized and put to use. During the last semester we learned a great deal not just in computer science but interdisciplinary in engineering, what makes us look back to a very successful and interesting bachelor's thesis.

## Declaration of primary authorship

I / We hereby confirm that I / we have written this thesis independently and without using other sources and resources than those specified in the bibliography. All text passages which were not written by me are marked as quotations and provided with the exact indication of its origin.

| | | |
|---|---|---|
| Place, Date: | Biel, June 11, 2015 | |
| Last Name/s, First Name/s: | Morandi Matteo | Rothen Tobias |
| Signature/s: | ...................................... | ...................................... |

# References

[1] Bruce Schneier, "The Strange Story of DUAL_EC_DRBG" 2007 [Online], Available: https://www.schneier.com/blog/archives/2007/11/the_strange_sto.html

[2] Tom McNichol, "Totally Random" 2003, Wired, [Online], Available: http://archive.wired.com/wired/archive/11.08/random.html

[3] Ian Goldberg and David Wagner, "Randomness and the Netscape Browser" 1996, CS Berkley [Online], Available: http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html

[4] Module Computer Science Seminar, Bern University of Applied Science, http://www.ti.bfh.ch/fileadmin/modules/BTI7302-de.xml

[5] Module Project 2, Bern University of Applied Science, http://www.ti.bfh.ch/fileadmin/modules/BTI7302-de.xml

[6] Glenn Greenwald, "How the NSA tampers with US made internet routers" 2014, The Guardian, [Online], Available: http://www.theguardian.com/books/2014/may/12/glenn-greenwald-nsa-tampers-us-internet-routers-snowden

[7] Cryptographic Key Recommendation, Blue Crypt, [Online], Available: http://www.keylength.com/

[8] Strategy Design Patter, sourcemaking.com, [Online], Available: https://sourcemaking.com/design_patterns/strategy

[9] usb4java, a Java library based on libusb 1.0, [Online], http://usb4java.org/

[10] RXTX,native interface to serial ports in Java, [Online], https://github.com/rxtx/rxtx

[11] Model view controller Wikipedia, [Online], http://en.wikipedia.org/wiki/Model_view_controller

[12] JGoodies FormLayout, [Online], http://www.jgoodies.com/freeware/libraries/forms/

[13] Statistical Randomness, [Online], http://en.wikipedia.org/wiki/Statistical_randomness

[14] Random walk with 25000 steps, [Online], http://upload.wikimedia.org/wikipedia/commons/e/ea/Random_walk_25000_not_animated.svg

[15] Random walk square root of two, [Online], http://www.jeffreythompson.org/blog/2012/01/04/random-walk-square-root-of-two/

[16] Hash Visualization: a New Technique to improve Real-World Security, [Online], http://users.ece.cmu.edu/~adrian/projects/validation/validation.pdf

[17] Pseudo-random vs. True random, [Online], http://boallen.com/random-numbers.html

[18] Pulse-code modulation, [Online], https://en.wikipedia.org/wiki/Pulse-code_modulation

[19] QR code API, [Online], http://goqr.me/api/doc/create-qr-code/

[20] MOSFET Wikipedia, [Online], http://en.wikipedia.org/wiki/MOSFET

[21] PIGPIO library, [Online], http://abyz.co.uk/rpi/pigpio/

[22] Unix Signals Wikipedia, [Online], http://en.wikipedia.org/wiki/Unix_signal

[23] Tinkercad [Online], www.tinkercad.com

*Content of all web pages dated June 11, 2015*

# A. Transmission Flow

## A.1. Server Startup

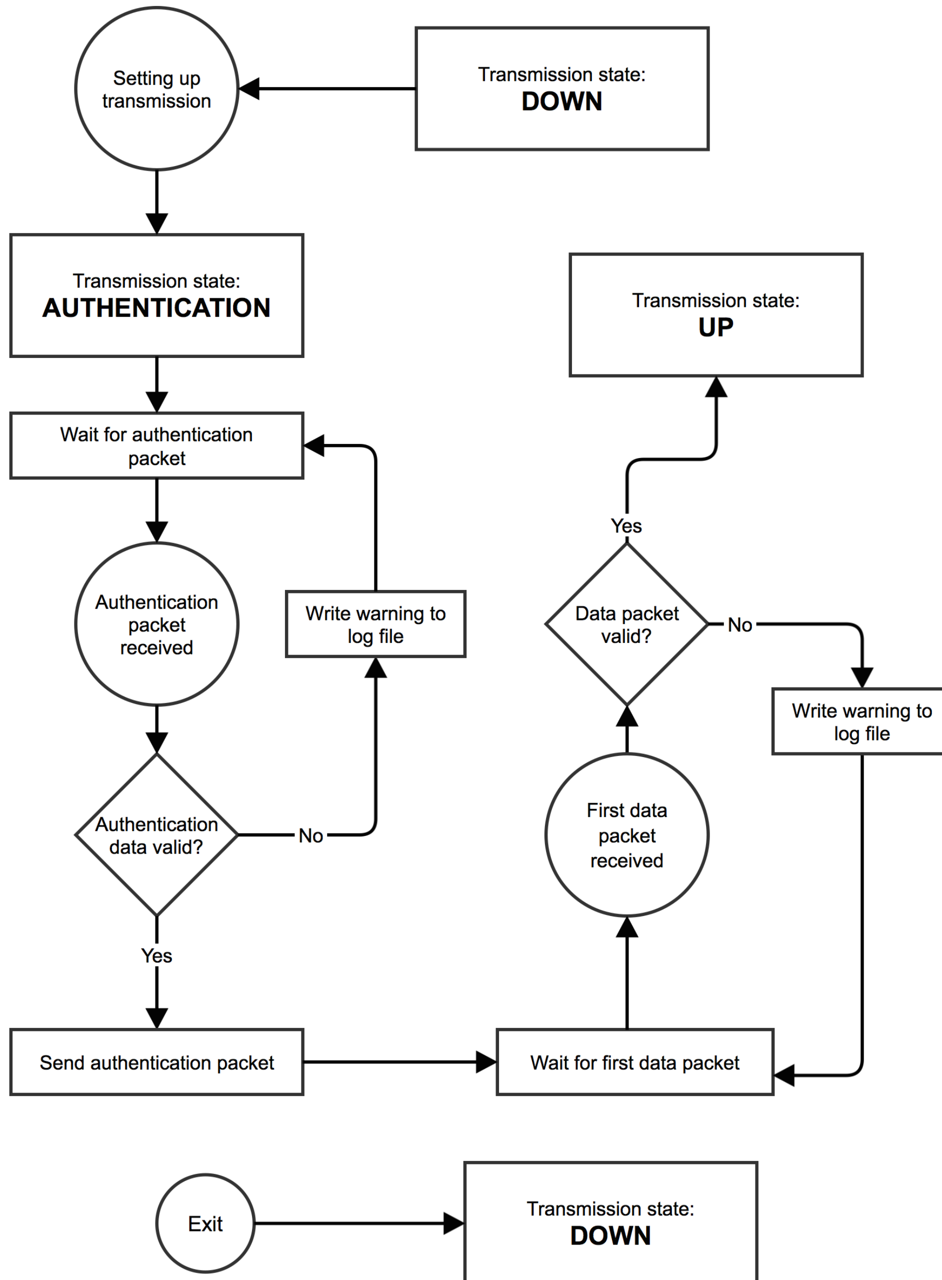



Figure 45: Network Flow Server Startup

## A.2. Server Data Transport

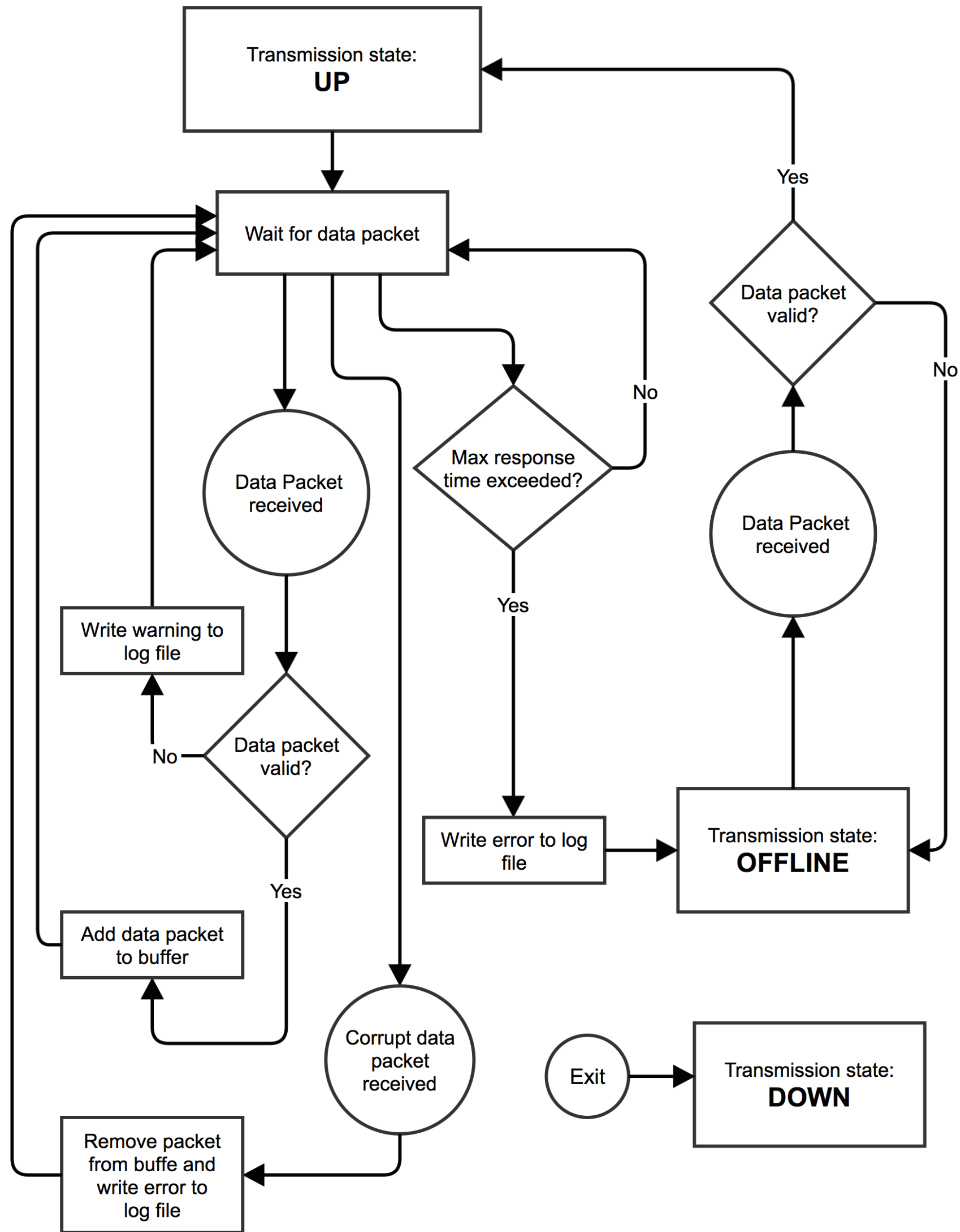



Figure 46: Network Flow Server Data Transport

## A.3. Generator Startup

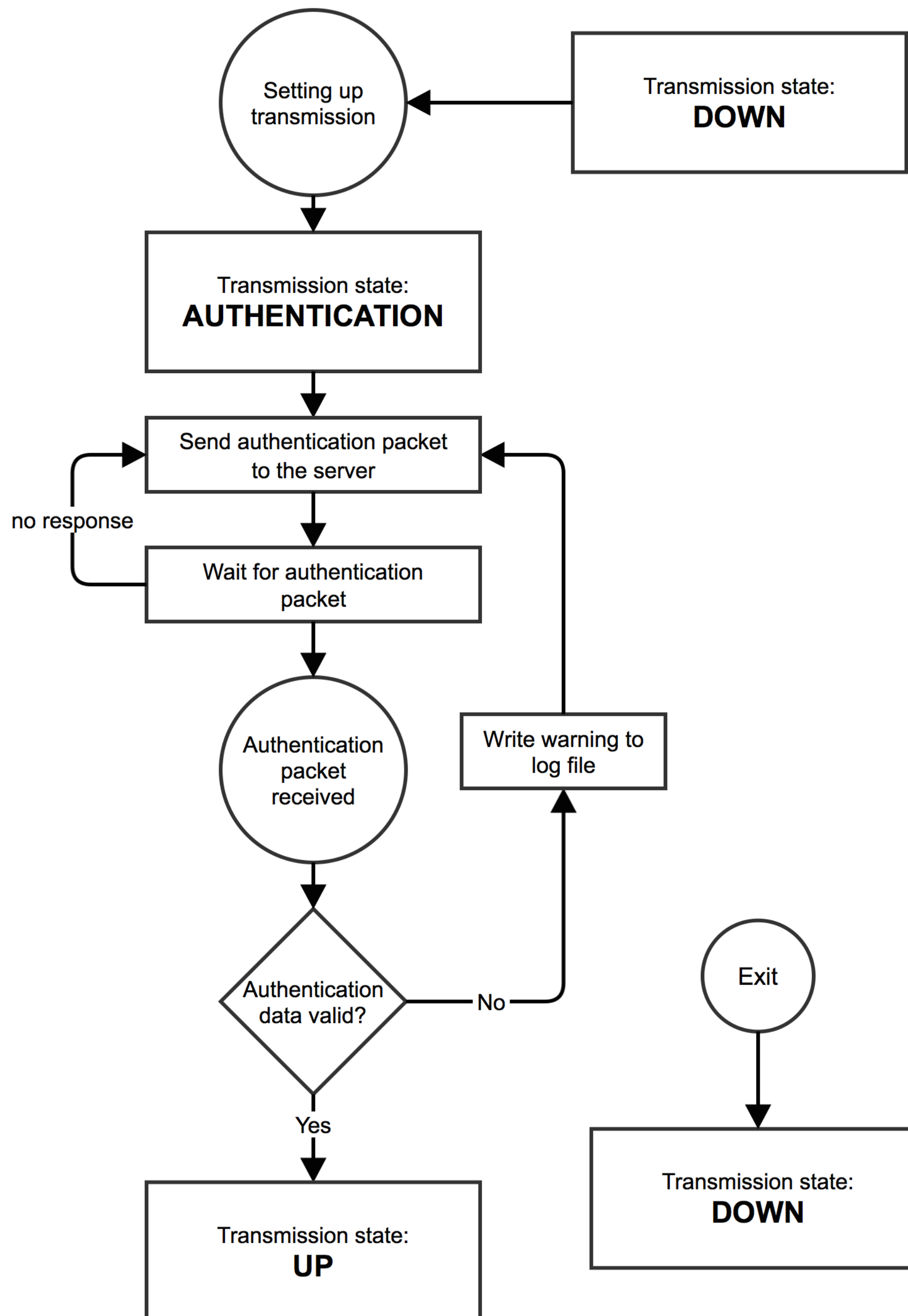



Figure 47: Network Flow Generator Startup

## A.4. Generator Data Transport

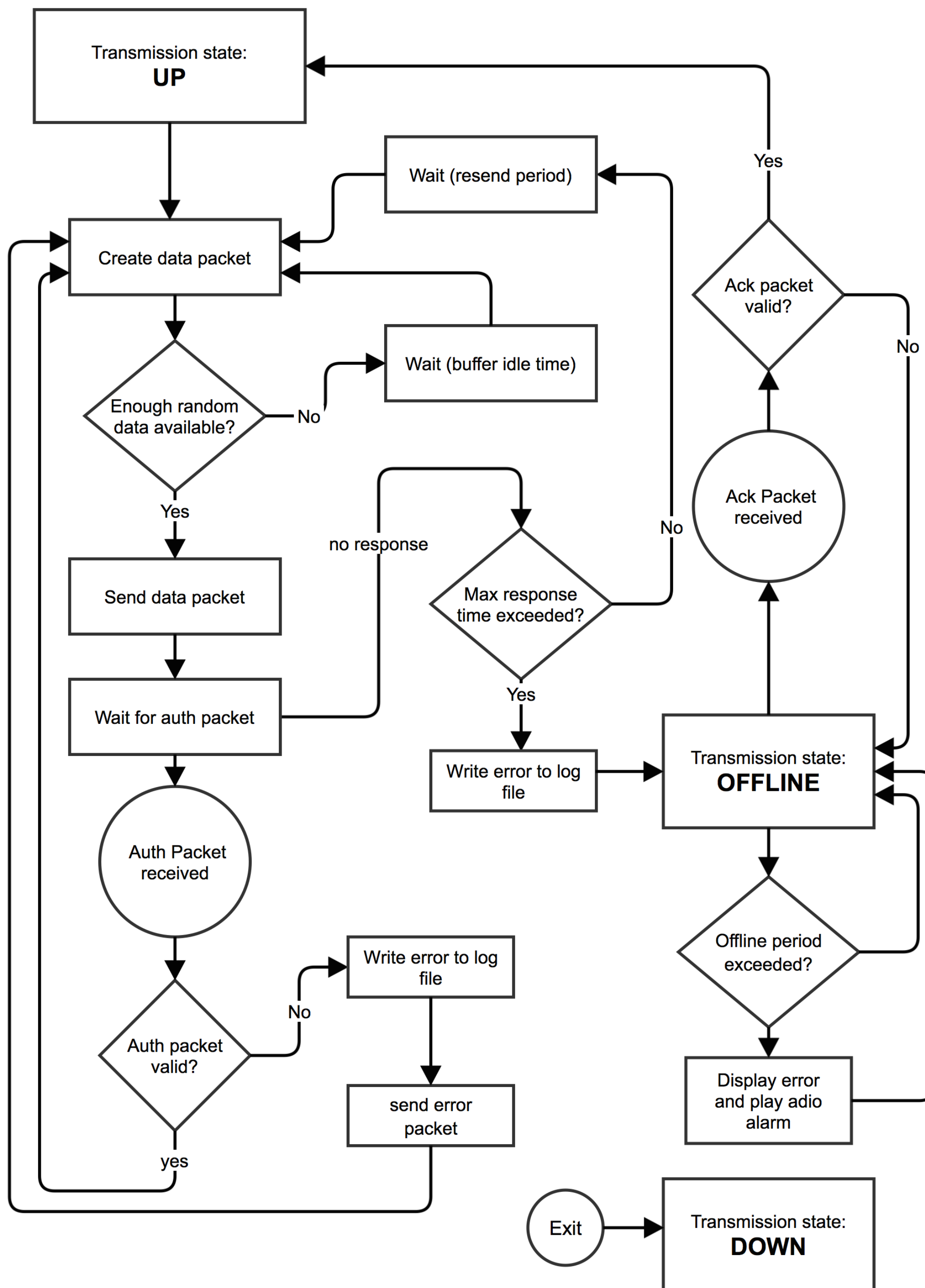



Figure 48: Network Flow Generator Data Transport

# B. Installation Manuals

## B.1. (Installation Generator(Raspberry Pi)

### B.1.1. Setup

The application on the server is possessing the following folder structure:

| | |
|---|---|
| /*.jar | executable application |
| /picturepool/ | contains the pictures taken by the data |
| /conf/ | contains the properties file for configuration |
| /out/ | contains the dumps of the produced random numbers |
| /truerandom.log | log file for verbose output |

Although the folders are supposed to be created once the application starts, this should give an oversight on how the applications looks like in a vital state.

### B.1.2. Prerequisites

**1. Raspbian**
Download and install Raspbian on a RaspberryPi 2 as instructed on:
`https://www.raspberrypi.org/downloads/`

**2. Java JDK8**
Java will most likely already be installed in the latest Raspbian image. However, you may need to configure the Java version in order to run the software. Set the java and javac to version 8:

```
sudo update-alternatives --config java
sudo update-alternatives --config javac
```

The application has only been tested with Oracle JDK 8, hence running the application with other java versions is not supported.

**3. Git and Maven (for developers)**
Git is used to pull the source code from the repository and Maven to build and package the software. Alltough our software comes in a Jar, you may need this to pull and build future updates.

```
sudo apt-get install git maven
```

**4. Java Native Interface**
JNI is used for calls to native libraries

```
sudo apt-get install libjna-java
```

**4. RXTX**
This library is used for serial communication between the generator and the server.

```
sudo apt-get install librxtx-java
```

**5. Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy**
Installation needed to enable strong cryptography (256 bit keys in AES encryption). Copy and overwrite the two files local_policy.jar and US_export_policy.jar.

```
sudo cp US_export_policy.jar $JAVA_HOME/jre/lib/security/US_export_policy.jar
sudo cp policy.jar $JAVA_HOME/jre/lib/security/policy.jar
```

**6. PIGPIO Library**
This library is needed to control the fans and LEDs using the GPIO pins of the RaspberryPi.

```
wget abyz.co.uk/rpi/pigpio/pigpio.zip
unzip pigpio.zip
cd PIGPIO
make
make install
```

**7. RaspberryPi / Display / Camera**
Raspbian needs some changes to its configuration for the touchscreen display, the camera and overall performance:

Add the following lines to /boot/config.txt:

```
hdmi_force_hotplug=1
hdmi_group=2
hdmi_mode=1
hdmi_mode=87
hdmi_cvt 1024 600 60 6 0 0 0
max_usb_current=1
disable_camera_led=1
```

Furthermore start raspi-config and make the following changes:

```
sudo raspi-config
->Enable Camera -> Enable
```

```
-> Overclock -> Modest
->Advanced Options -> Serial -> No
```

### B.1.3. Starting the Application

Use the startup.sh script to start the application:

```
source start.sh
```

NOTE: The application has to be started with root privileges!

## B.2. Installation (Server)

### B.2.1. Setup

The application on the server is possessing the following folder structure:

| | |
|---|---|
| /*.jar | executable application |
| /picturepool/ | contains the pictures taken by the data |
| /conf/ | contains the properties file for configuration |
| /out/ | contains the dumps of the produced random numbers |
| /truerandom.log | logfile for verbose output |

Although the folders are supposed to be created once the application starts, this should give an oversight on how the applications looks like in a vital state.

### B.2.2. Prerequisites

**1. Java JDK8**
Java will most likely already be installed on the Ubuntu server. The version can be checked using java -version. If not already installed, install the newest version of java and make sure it is configured as standard:

```
sudo update-alternatives --config java
sudo update-alternatives --config javac
```

The application has only been tested with oracle JDK 8, hence running the application with other java versions is not supported.

**2. RXTX**
This library is used for serial communication between the generator and the server. Install it with the command:

```
sudo apt-get install librxtx-java
```

**3. Java Cryptography Extension(JCE) Unlimited Strength Jurisdiction Policy**
Installation needed to enable strong cryptography (256 bit keys in AES encryption). Copy and overwrite the two files local_policy.jar and US_export_policy.jar.

```
sudo cp US_export_policy.jar $JAVA_HOME/jre/lib/security/US_export_policy.jar
sudo cp policy.jar $JAVA_HOME/jre/lib/security/policy.jar
```

### B.2.3. Starting the Application

Use the startup.sh script to start the application:

```
source start.sh
```

NOTE: The application has to be started with root privileges!

### B.2.4. Application Recovery

The Application will create a so called hook-file when terminated unplanned. This enables the server to maintain the serial connection with the generator after a unplanned reboot. If you want to start the application normally and re-authenticate the generator, use the cleanup script prior to start up:

```
source cleanup.sh
source start.sh
```