University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Performance Analysis of Blockchain Off-chain Data Storage Tools

Lukas Eisenring
Aarau, Switzerland
Student ID: 14-709-455

Supervisor: Bruno Rodrigues, Dr. Thomas Bocek
Date of Submission: January 31, 2018

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

1. Einleitung

Das Blockchain Signaling System (BloSS) ist eine Applikation zur Entdeckung, Weiterleitung und Bekämpfung von verteilten Denial of Service (DDoS) Angriffen. Es arbeitet mit weiteren System für das Netzwerkmanagement zusammen. BloSS verwendet die Ethereum Blockchain für die Kommunikation zwischen seinen jeweiligen Instanzen. Da die Speicherung von grossen Mengen an Daten auf der Blockchain aufwendig und teuer ist, sollten die Daten ausserhalb der Blockchain gespeichert und mit ihr verknüpft werden. Die Verknüpfung wird vorgenommen, um weiterhin von den Vorteilen der Blockchain, wie Unveränderbarkeit und Rückverfolgbarkeit, profitieren zu können.

2. Ziele

Basierend auf den Anforderungen des BloSS an eine Speicherlösung ausserhalb der Blockchain soll ein geeignetes Produkt evaluiert werden. Das Produkt wird in das BloSS integriert. Mit dem dabei entstandenen System wird eine Überprüfung der Leistungsfähigkeit durchgeführt. Dies geschieht sowohl in lokalen Tests innerhalb des Netzwerkes der Universität Zürich als auch in einer weltweit verteilten Testanordnung, um die Auswirkungen von hohen Verzögerungen und instabilen Netzwerkverbindungen bestimmen zu können.

3. Resultate

IPFS wurde als Speicherlösung für BloSS ausgewählt. Es erfüllt die Anforderungen am Besten. Mittels IPFS, der Etherum Blockchain und dem existierenden BloSS Programmcode wurde ein System entwickelt, welches für die Testläufe um weitere Funktionen, wie die Möglichkeit zur Generierung von Fülldaten, erweitert wurde. Das System ist in Python3 geschrieben und basiert auf dem Betriebssystem Ubuntu. Für die Kommunikation mit den jeweiligen Netzwerken wurden die Implementierungen geschrieben in der Sprache Go genutzt.

Basierend auf dem erstellten System und dem Ethereum Testnetzwerk Rinkeby wurden mehrere Tests durchgeführt. In lokalen Tests innerhalb des Netzwerkes der Universität Zürich wurde das System mit Dateigrössen von bis zu 4 GB und zehn Dateien gleichzeitig getestet, wobei keine Fehler auftraten.

Die Testanordnung wurde anschliessend auf insgesamt acht weltweit verteilte Instanzen erweitert. Die dazu genutzte Hardware wurde bei unterschiedlichen Cloud-Providern an verschiedenen Standorten zu gemietet. Bei der Übertragung von jeweils 10'000 IP-Adressen über IPFS zeigten sich bei der Übertragungsdauer Mittelwerte im Bereich von 0.5 und 2.6 Sekunden. Generell erhöhte sich die benötigte Übertragungsdauer bei grösseren Distanzen. Der Mittelwert der benötigten Zeit reduzierte sich durch die Anwesenheit von weiteren Instanzen in geografischer Nähe. Die Zwischenspeicherung der Daten nach dem Empfangen im Peer-to-Peer Netzwerk erlaubt es gelegentlich, die Daten von einer naheliegenderen Instanz zu beziehen, was in kürzeren gemessenen Zeiten resultierte.

Um die Möglichkeit von grösseren Listengrössen zu prüfen, wurde ein Durchgang mit 150'000 IP-Adressen durchgeführt. Dabei zeigten sich die gleichen Tendenzen bezüglich dem Verhältnis zwischen den einzelnen Übertragungswegen wie bei den vorherigen Tests und Medianwerte bei der Übertragungsdauer von bis zu 20 Sekunden.

Bei einem Vergleich der Übertragungsdauer von HTTP und IPFS war die Übertragung mittels IPFS durchwegs schneller als über HTTP bei jeweils identischen Sendern und Empfängern.

Die gesamte Transaktionsdauer vom Hinzufügen der Datei zu IPFS, über die Transaktion auf der Blockchain bis zum Empfangen der Datei auf einer anderen Instanz dauerte in über 95% der Fälle zwischen 15 und 35 Sekunden.

4. Weitere Arbeiten

Weitere Optimierungen am entwickelten Programmcode lassen sich bezüglich Stabilität, Sicherheit und Zusammenarbeit realisieren. Der Client zum IPFS Netzwerk stürzte in den Tests einige Male ab, da dem System der Arbeitsspeicher ausging. Die Sicherheit des Vertrages auf der Blockchain und der übertragenen Daten wurde in dieser Arbeit nicht berücksichtigt und müsste vor einem Produktiveinsatz berücksichtigt werden. Des Weiteren fehlt im momentanen Zustand eine Verknüpfung zu den anderen Teilen von BloSS, da diese nicht ausserhalb der Testumgebung, basierend auf Asus Tinker Boards, verfügbar sind. Eine dezentralisierte Verwaltung der einzelnen Teilnehmer, respektive der Verträge auf der Blockchain, würden die Zusammenarbeit vereinfachen.

Abstract

An off-chain data storage tool unburdens a blockchain of storing large datasets. An evaluation based on the requirements of the Blockchain Signaling System (BloSS) appointed IPFS to be the most suitable tool.

In a local setup inside the network of the University of Zurich, IPFS managed files up to 4 GB size and ten files simultaneously. The setup was enlarged to eight worldwide distributed instances. In the test, 10'000 IP-addresses were submitted in between 0.2 and 2.6 seconds in median depending on the pair of instances. Overall, including the transaction on the Ethereum test-network Rinkeby, 95% of the data was transmitted in between 15 and 35 seconds.

Acknowledgments

I would like to thank Professor Stiller for providing me the opportunity of writing my bachelor thesis at the Communication Systems Group of the University of Zurich and Dr. Thomas Bocek for the supervision of my thesis.

I am especially thankful for the various inputs, hints, and feedback from my supervisor Bruno Rodrigues about my work and giving me additional information about BloSS. Your feedback and reviews were very constructive for me to complete this thesis.

Also many thanks to Andreas Gruhler and Mervin Cheok working simultaneously on their thesis in the BloSS environment for the helpful interchange of ideas.

Contents

Zusammenfassung **i**

Abstract **iii**

Acknowledgments **v**

1 Introduction **1**

1.1 Description of Work 1

1.2 Thesis Outline 2

2 Background and Related Work **3**

2.1 Blockchain 3

2.1.1 Consensus Protocols 3

2.1.2 Ethereum 4

2.2 Off-Chain Storage 4

2.2.1 Interplanetary File System 4

2.2.2 Swarm 6

2.2.3 StorJ 6

2.2.4 Constrained Application Protocol 6

2.3 Related Work 7

3 System Design **9**

3.1 Blockchain Signaling System 9

3.2 Requirements for the Off-chain Storage Solution 10

3.3 Evaluating a suitable Technology 11

3.3.1 IPFS 11

3.3.2 Swarm 11

3.3.3 StorJ 12

3.3.4 CoAP 12

3.3.5 Selected Solution 12

3.4 Integration of the Off-chain Storage Tool into BloSS 12

3.5 Operation System 14

3.6 Programming Language 14

3.7 Other possible Limitations 14

3.8 File Format for Storing the Data 15

3.9 Blockchain 16

3.9.1 Smart Contracts 16

3.10 Measurement 17

3.10.1 Overall Duration 18

3.10.2 Duration of collecting Data from IPFS 18

3.11 Module Design 19

4 Evaluation **21**

4.1 Underlying Hardware 21

4.1.1 Instances for Local Tests 22

4.1.2 Instances for the Worldwide Test 22

4.1.3 Ping Requests 23

4.2 Local Tests 23

4.3 Overall Duration 25

4.4 IPFS in worldwide Usage 27
4.4.1 Acting with larger Datasets 27
4.4.2 Comparison with HTTP 28
4.5 Failures 29
4.5.1 Lack of Memory 29
4.5.2 IPFS cannot find hash 30

5 Summary and Conclusions **31**

Bibliography **33**

Abbreviations **35**

List of Figures **35**

List of Tables **37**

A Installation Guidelines **41**

B Contents of the CD **43**

Chapter 1

Introduction

The Blockchain Signaling System (BloSS)[17] is designed for preventing and reducing Distributed Denial-of-Service attacks. BloSS is working with distributed instances each of them generating large lists consisting out of IP-addresses. These lists have to get communicated between the instances of BloSS for signaling and mitigating attacks on assigned networks.

BloSS is designed to work together with the Ethereum blockchain, which is a decentralized approach to store, verify, and enforce transactions. Ethereum can store transaction records using smart contracts. However, the usage of smart contracts for storing data is expensive, slow, and limited. Therefore, another solution has to be found for the purpose of BloSS.

The usage of off-chain data storage tools like the Interplanetary File System (IPFS)[2] or Swarm[9] is an approach to store data outside of the blockchain. These tools unburden the blockchain from the load of storing large datasets while still offering some of the advantages of the blockchain technology like immutability, persistence, retraceability, and indisputableness.

Because BloSS is working in a real-time environment the transfer of data and the interaction with other instances of BloSS is time-critical in certain boundaries. Nowadays no literature is available to appreciate the behavior of IPFS in a global environment similar to BloSS.

1.1 Description of Work

The goal of this thesis is to find a solution for storage and communication desires of BloSS. Available off-chain storage tools have to get evaluated according to the requirements of BloSS. Afterwards, an integration of the chosen tool into BloSS has to get designed and implemented. Based on the developed system, performance measurements are made for demonstrating the usability of the solution and gather insight into the capabilities of the tool.

1.2 Thesis Outline

As a starting point, the evaluated and used concepts, technologies, and products are shortly described in Chapter 2.

In Chapter 3, a possible approach for an implementation of an off-chain solution into BloSS is discussed. The proposed settlement utilizes the Interplanetary File System as external storage for the data. The data gets added to IPFS, and the unique hash value of the data is inserted into a smart contract on the blockchain to signal the action to the other participants of the BloSS consortium. The other instances retrieve the data over IPFS using the communicated value from the smart contract.

Based on the system described in the previous chapter, Chapter 4 contains the process and result of testing the developed system in a locally and worldwide spread approach. First, the local tests inside the network of the University of Zurich explore the boundaries of IPFS with attention on data size, on simultaneous transfer of multiple files, and on the delay of transfer time. In addition, the influence of the file size on the transmission time is described. In the worldwide test with eight nodes spread all over the world, the behavior of IPFS and Ethereum transmitting data over wide distances and through uncontrolled networks is investigated. Measured are the values of the total transmission time containing the publication of the address list to IPFS, updating the contract on the blockchain, and receiving the file on another instance. Furthermore, the used time for IPFS and the blockchain are measured separately. The measurement values are compared with transmission over HTTP. A typically used file was containing an IP address list of 10'000 addresses resulting in a file size of 173 kB. This measurement is compared with a more massive list containing 150'000 addresses. This amount of addresses correspond to the number of involved devices in one of the most massive DDoS attacks in the past.

Finally, Chapter 5 summarizes the work done and gives suggestions for further improvements to the created solution and the utilized technologies.

Chapter 2

Background and Related Work

In this chapter, the used tools and their underlying technologies are shortly described. As a starting point, Blockchain and the used implementation Ethereum is presented in Section 2.1. Then four different off-chain storage tools are presented in Section 2.2 and for finishing in Section 2.3 related work is mentioned.

2.1 Blockchain

A blockchain is a technology of distributed datasets stored in blocks. Together these blocks form a record in which the blocks are connected and secured against mutation by using cryptographic procedures[14]. The users are creating transactions for writing data on the blockchain. These transactions get saved in blocks generated by miners utilizing a consensus finding algorithm as described in Section 2.1.1. If the same block is generated multiple times with different content, various branches of the chain are resulting. In this case, the longest chain is valid because there is the most effort done for calculating this chain[14]. The blocks are stored on distributed nodes and get exchanged using peer-to-peer technology[24].

2.1.1 Consensus Protocols

Consensus protocols are used to build consensus on the blocks and are therefore crucial for a blockchain. Many different protocols and implementations exist. In this section two consensus finding protocols are described. The public Ethereum main network uses the proof of work protocol while the Rinkeby test network works with proof of authority implementation.

2.1.1.1 Proof of Work

By using the proof of work consensus protocol, a mathematical problem has to be solved by the miner to create the block[14]. The first miner which resolves the problem attaches

his block to the chain. While multiple branches of the chain could exist, the longest chain is always selected as the valid one because there is the most computing time in it[14]. A block in the chain can not get changed after its creation unless it is possible to recalculate the previous blocks faster than another node can attach a block to the original chain[14]. In consequence the system is secure against mutation as long as a possible attacker has less than half of the computing power of the whole network[14].

2.1.1.2 Proof of Authority

In contrast to the proof of work method, the proof of authority consensus algorithm does not rely on solving mathematical problems. Instead, it uses defined sets of authorities, which are allowed to create and sign blocks[25]. A block has to get signed by a majority of the authorities to become part of the permanent chain[25]. Proof of authority based blockchains are often used for private blockchains or test networks.

2.1.2 Ethereum

Ethereum was started in 2015 by Vitalik Buterin, designed as a blockchain for supporting decentralized applications (DApps)[20]. The DApps are communicating over so-called smart contracts on the blockchain, and the smart contracts get executed by the miners of the blocks[24]. For this effort, the miners receive an incentive in the own cryptocurrency Ether (ETH), which can also get used for financial transactions[24].

Ethereum uses proof of work consensus mechanism for creating the agreement on the block[24]. The time between the creation of two blocks is influenced by the difficulty which regulates the complexity of the mathematical problem to stabilize the block time[24]. The block time is designed to be seven seconds[20].

2.2 Off-Chain Storage

An off-chain storage solution unburdens the blockchain from part of the load of saving large datasets onto the chain. The challenge is to swap out the data but retain as many advantages, like immutability and traceability, of the blockchain as possible. In the following section four approaches for off-chain storage solutions are presented.

2.2.1 Interplanetary File System

The Interplanetary File System (IPFS) is a distributed file system based on peer-to-peer technology from BitTorrent and versioning techniques of Git[2]. In IPFS, files get addressed by their hash value of their content[2]. IPFS is a peer-to-peer system, like the in Section 2.1 presented blockchain and has, therefore, the advantage that it “has no single point of failure and nodes do not need to trust each other.”[2]

The management of the owners, and therefore potential provider of a file, is implemented by using a combination of the distributed hash table (DHT) systems Kademlia and Coral[2]. Kademlia provides with an average of $log_2(entries)$ queries for finding an entry an efficient lookup system[2]. Furthermore, it has a low coordination overhead, and it prefers long-living nodes to increase resistance against various attacks[2].

The data of IPFS is stored in a structure using an object Merkle directed-acyclic graph (DAG)[2]. The DAG is content addressed because it uses the hash value of the file as identifier[2] and this provides security against mutation. Furthermore, the objects get deduplicated because the same content results in the same hash value and gets therefore stored at the same place.

IPFS can transport large files, splitted into multiple parts. These parts are called chunks and have, by default, a size of maximal 256 kB[2]. A more massive file gets split up into multiple chunks, and in place of the file on the DHT, a list of the chunk addresses is saved.

Data exchange between different nodes is performed over the, from BitTorrent inspired, protocol BitSwap[2]. BitSwap has a `want_list` for files it is looking for and a `have_list` including the files it could provide to other nodes[2]. For preventing the network from free-loaders (systems only leeching), BitSwap has a credit-based incentive system implemented to stimulate nodes providing their resources and punish uncooperating nodes with exclusion from the network[2].

```
/ip4/130.60.73.41/tcp/4001/ipfs/QmdjWtStehBS5nMor2NLomnW2d8RaixjmEvJLQ59C3aZrf
```

Figure 2.1: Example of an IPFS multiaddress.

A `nodeId` uniquely identifies a node in the IPFS network. The `nodeId` is the hash value of the corresponding public key. Including the connection information, a node has a mutable `multiaddr` assigned. This address can change by time and is used to build a swarm. Figure 2.1 shows an example of a `multiaddr`. It includes the `nodeId`, IP-address, port and protocols to connect to this node. A `multiaddr` is extendable for supporting encapsulation and using proxies[2].

2.2.1.1 Interplanetary Name System

The Interplanetary Name System (IPNS) provides mutable links to immutable content. It behaves roughly like the domain name system: Every user gets a mutable namespace assigned. This namespace is linked to a hash value representing the file in IPFS[22]. While changing the content of the file, a new hash value results and the pointer has to get changed to the new file.

2.2.2 Swarm

"Swarm is a distributed storage platform and content distribution service, a native base layer service of the ethereum web 3 stack."[9] It stores the data redundantly and distributed over multiple nodes[9]. This approach allows Swarm to be DDoS-resistant, fault-tolerant, and censorship-resistant[9] during large-scale operation. Swarm is primarily designed to store the Ethereum's public records but also provides the opportunity to store other files[9]. For motivating the peers to offer their resources, an incentive system is integrated which can even penalize a peer for losing the hosted data of another party[9].

2.2.3 StorJ

StorJ is a network of distributed and encrypted data storage[23]. A file gets split up into different so-called shards, and the shards get encrypted on the client site for uploading to different storage nodes[23]. As the identifier for the file, the hash value of the original file is used and supplemented with ordering information of the blocks[23]. As data storage personal computer or servers can be used[23] and the shards are stored redundantly on multiple machines. The storage owner gets an incentive payment after checking the availability and integrity of the data from time to time[23]. Splitting, hashing and encrypting has to be done by the publisher of the file[23]. This is computation intensive for the client.

2.2.4 Constrained Application Protocol

"The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained networks."[19] The peer-to-peer approach supports multicasting over UDP [19] for delivering the payload simultaneously to multiple recipients which save network resources. The communication based on a REST model working with stateless transactions and HTTP connectors are available as well[19]. Furthermore, the CoAP specification allows caching the content and provides security mechanisms for encrypting the communication between nodes[19].

2.3 Related Work

There is only a small number of publications available in the domain of globally spread off-chain solutions connected with a blockchain. In the year 2017 Confais, Lèbre & Parrein[6] made research comparing the behavior of IPFS to Rados and Cassandra in local test-networks for evaluating the performance in fog and edge computing. They recognized that IPFS is comparably slow in writing or adding files to their network, but faster in reading them over the network[6]. The receiving of a 256 kB file lasts about three seconds on average, and the writing process to a non-local IPFS gateway takes about five seconds[6]. Additional findings are that IPFS creates a lot of requests between the different IPFS nodes because of the Kademlia hash table[6]. Furthermore, IPFS is much faster in requesting the same file a second time, because it caches the data locally after the first delivery[6].

Later in the year, the same authors published an additional work[5] about a performance analysis of optimizing a scale-out network attached storage using IPFS. For comparing their result of the combined system, they did also performance measurements with IPFS alone. Table 2.1 shows their results in reading and writing time while varying filesize and latency.

	Mean **writing** time (seconds)			Mean **reading** time (seconds)		
Latency / Workload	5ms	10ms	20ms	5ms	10ms	20ms
100 x 256 KB	0.473	0.490	0.730	0.397	0.531	0.776
100 x 1 MB	1.647	1.644	1.855	1.403	1.515	1.934
100 x 10 MB	15.260	15.240	15.410	14.102	14.763	15.326

Table 2.1: Mean time using IPFS in function of the latency (from Confais *et al.*[5]).

In this tests, files were transferred between local situated machines and the latency between the different servers was varied. An increasing latency has a more significant impact on the transfer time for smaller files than for bigger files[5]. Furthermore, for smaller files and 20ms latency, it was faster to write the file as to read the file. The problem of IPFS in this setup was that the distributed hash table is not entirely local available. This issue means that for finding an entry in the DHT sometimes a request has to be made outside of the test setup and for this request a high latency results[5]. Also, it is not possible to detach the local network from the internet without losing access to information on the DHT.

Chapter 3

System Design

This thesis is based on the Blockchain Signaling System (BloSS) presented in Section 3.1. The requirements regarding the storage and communication of data of BloSS are discussed in Section 3.2. Based on these needs, four possible off-chain storage tools are discussed in Section 3.3.

As outlined in Section 3.3, IPFS is chosen as the storage solution. The integration of IPFS into BloSS is presented in Section 3.4, followed by considerations about the underlying operating system and programming language. Furthermore, problems that might occur while using the system in combination with BloSS are discussed in Section 4.5 and a data format for interchanging the data is suggested in Section 3.8. The counterpart of the storage solution IPFS is the smart contract on the Ethereum blockchain. The created contract and the underlying Ethereum blockchain is presented in Section 3.9. The measurement methods for performance analysis, as the key finding of this thesis, are discussed in Section 3.10. The last Section 3.11 presents the design of the source code implementing the system described in the previous chapters.

3.1 Blockchain Signaling System

The Blockchain Signaling System (BloSS) is designed for preventing and reducing Distributed Denial-of-Service (DDoS) attacks. It is coupled to network management systems analyzing the network traffic to detect DDoS attacks against an administrated network and allows filtering the network traffic according to rules given by BloSS. The system will report the occurrence of attacks to the BloSS, which signalizes the attack and provides information to other instances of BloSS by using smart contracts on the Ethereum blockchain.[16] [17]

3.2 Requirements for the Off-chain Storage Solution

The off-chain storage solution has to get evaluated according to the prerequisites on the BloSS system described in the Section 3.1. The primary requirement is the possibility of collaborating with the blockchain through BloSS. Further requirements are described in the next sections. These requirements were used to evaluate a suitable off-chain storage tool.

Decentralised System
As the blockchain is a distributed ledger and has no central instance on which it relies on, the off-chain storage solution should also operate as an independent, decentralized system. This requirement also means that there should not be a central point of failure, to prevent systematic attacks against this point.

Data Integrity and Immutability
Modifying the address list would be a new attack method because the BloSS would ban devices from the network. Therefore, it is essential that the data cannot get modified or that corruption is detectable.

Unique Signifier
The off-chain storage solution has to provide a unique signifier representing a pointer to the data. The signifier is required to integrate this pointer into the blockchain to expand the advantages of the blockchain onto the storage solution.

Retrievability & Durability
An instance of BloSS has to be able to retrieve the data from the off-chain solution only having given the information out of the contract on the blockchain. Retrieving data should also be possible after a few hours, *e.g.* in the case an instance was offline due to an attack or another issue.

Transmission Time
Because the situations in DDoS attacks can change fast, the mitigation solution has to keep pace with this changes. Therefore, it is required that the signaling of an attack with the corresponding information about the attackers should get transmitted within several minutes.

Extensibility
The BloSS system is under development, and the storage solution has to be capable of transmitting more than only address lists. Transmission of additional information or trust information might be needed in the future.

Stability
The storage solution, like the whole BloSS ecosystem, has to be designed for a continuous runtime. The system is expected to be highly available and to have low crash rates.

Communication Connections & Scalability
During an attack on the system, the availability of the communication channel could

be influenced. The setup has to ensure that downstream communication is still possible, in order to get new blocks from the blockchain. Upstream communication has to be ensured in order to publish transactions on the blockchains and deliver the files of the off-chain storage solution. Beneath the networking capabilities, also the computing capacity for performing the necessary calculations must be guaranteed. The application should be scalable to adjust to the desires of the corresponding network.

3.3 Evaluating a suitable Technology

Many possible storage tools for off-chain storage exist. In the next sections, the solutions IPFS, CoAP, Swarm, and StorJ are discussed on their applicability and compliance with the requirements named in Section 3.2.

3.3.1 IPFS

IPFS complies with most of the requirements, at least in theory: It is designed as a decentralized system working as a peer-to-peer system. The hash value of the corresponding file is used as the unique identifier belonging to the file. This approach declares the file as immutable because a change in the content of the file would result in another hash value. The hash value can be added to the blockchain, and therefore the nonrepudiability of the blockchain is extended to the storage solution. A file typically gets cached by all receiving nodes for up to 24 hours. During this time period, the file should be available for retrieval.
The drawback of this solution is the generated overhead for building the swarm of connected devices, hashing, and building a hash table for retrieving the nodes which are in possession of the desired data.

3.3.2 Swarm

Swarm is a part of the Ethereum ecosystem and collaborates perfectly with the blockchain. But at the time of the creation of this thesis, swarm is not fully developed and not publicly available as an enlarged network. The primary advantage of Swarm is, that after a file is stored onto Swarm, like submitting the transaction to the blockchain, the reporting instance does not need to be available anymore. This fact would be a big benefit for systems that are unavailable due to a DDoS attack. Swarm is designed as a financial incentive system, which means that the file is available as long as you pay for it. With the payment, the number of redundant storage nodes is also manageable[9]. Drawbacks are the large overhead for storing the files redundantly and the costs for using this solution in a production environment.

3.3.3 StorJ

StorJ provides a decentralized and encrypted storage solution based on an incentive system. The files are spread across multiple hosts redundantly, and all information is stored in a dedicated blockchain. To interact with the files, keys are needed, and therefore access control is guaranteed.

StorJ does presently not allow new members to join the network[21], and the function of sharing data is constricted due to changes in the architecture[1].

3.3.4 CoAP

CoAP is a transmission protocol and not a storage solution, unlike the other discussed solutions. Its advantages are the lightweight operation, low overhead as well as secure and efficient transmission. Drawbacks are the missing durability, because it allows only caching and not storing, and the missing signifier for connecting the data to the blockchain. An off-chain storage solution for BloSS could be built by using CoAP, but the additional effort, like creating a unique pointer, would be needed to implement such a solution.

3.3.5 Selected Solution

Based on the arguments in the preceding sections, IPFS was chosen as the off-chain storage solution for this thesis. It fits best into the requirements, and it is actually in operation. It is not an incentive-based system, which allows saving costs in operation, and it offers the possibility to build a closed and local network for testing. Furthermore, there is a python library available for the integration of IPFS into the BloSS system.

3.4 Integration of the Off-chain Storage Tool into BloSS

The Blockchain Signaling System produces datasets containing lists of IPv4-addresses. These lists have to get transmitted to other instances of BloSS using IPFS and the blockchain.

Figure 3.1 displays the interaction of BloSS with IPFS and the blockchain as a sequence diagram.

BloSS is connected to a system monitoring the network and reports attacking addresses to a file which contains lists of attackers. This file is pushed to the part called connector in the figure. This connector is in the duty of interacting with IPFS and the blockchain. First, it adds the file, which contains the list of attackers, to the IPFS network using the local IPFS gateway and the corresponding python library. This operation returns the

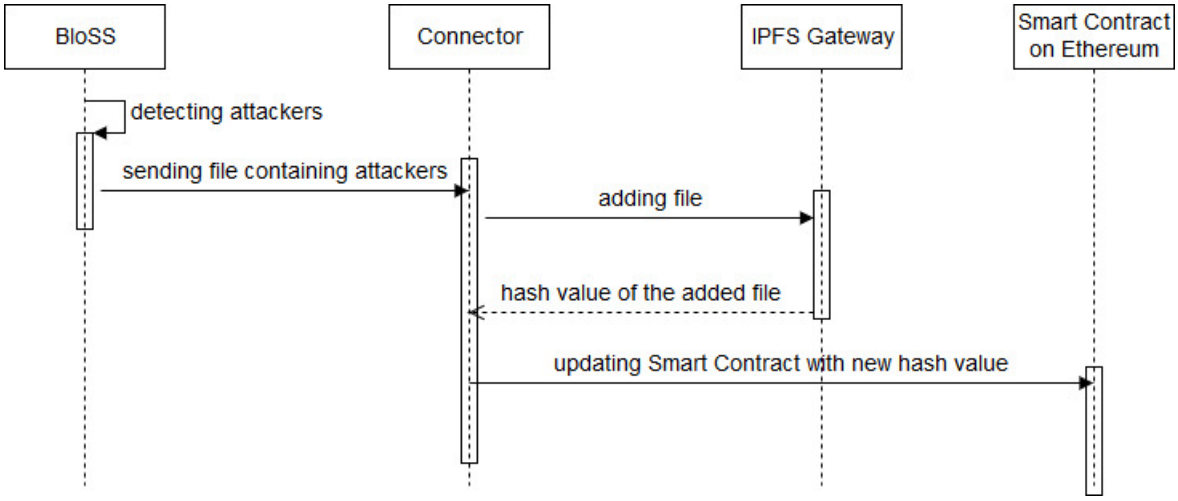

Figure 3.1: Sequence diagramm of publishing a file on IPFS and the blockchain.

hash value of the file under which the data can, later on, be retrieved by other instances. In the end, the smart contract gets updated with the hash value as an identifier for the file from the previous step.

Opposite to the previous mentioned sending instance, some instances of BloSS need to access these lists of addresses. Figure 3.2 shows the sequences of actions on an instance receiving address lists.

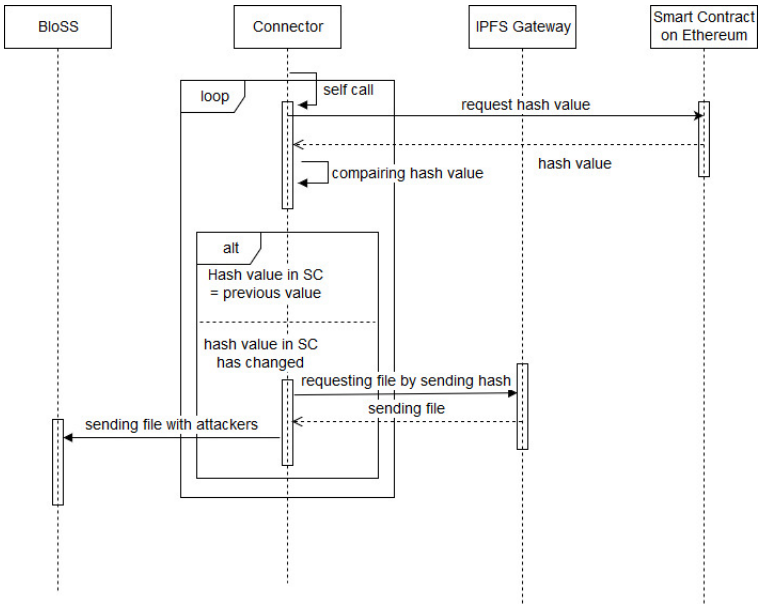

Figure 3.2: Sequence diagram of receiving a file from the blockchain and IPFS.

The connector is polling the smart contract and looking for a new hash value. If it recognizes a new value, the connector demands the IPFS gateway for searching and delivering the file with the given hash value. The IPFS gateway returns the data to the connector who will forward it to the BloSS system. The BloSS system can then use the information for blocking the attackers and therefore reduce malicious network traffic.

3.5 Operation System

BloSS is currently only executable on a development and presentation system using Asus Tinker Boards. However, for the usage in larger scale networks and distributed setups, the usage of single-board computers is no longer appropriate.

For this thesis, Ubuntu was chosen as the underlying operating system. It provides several advantages as it is easy to use and available on all major cloud providers. Furthermore, Ubuntu offers easy access to the network interface for BloSS and all clients for IPFS and Ethereum are available on this platform. To be precise, the actual long-term supported version 16.04 of Ubuntu was chosen.

3.6 Programming Language

The existing BloSS system is written in Python 2.7. In November 2017 the support of the Python module web3 for the communication with the geth client was announced to be discontinued at the end of the year 2017[4]. Because of this fact, the used parts were rewritten to Python 3.5 in order to have an actual codebase until the end of the thesis.

The smart contracts are written in the language Solidity version 0.4.19 which was the most recent released version at the time of creation of this application.

The additional scripts for simplifying and automating the installation process are written as bash scripts for Ubuntu.

3.7 Other possible Limitations

While BloSS is performing with IPFS and the Ethereum blockchain, many problems are imaginable to occur during an attack. Both of the systems, IPFS and Ethereum, are relying on access to the network for exchanging information with other nodes in their network. However, during a Distributed Denial-of-Service attack, the possibilities of the system to communicate with the network can be heavily limited due to the overload of network capacity, and it is the moment for BloSS to signal an attack for receiving help from other instances of BloSS.

The data has to get transmitted over IPFS thereby the other instances of BloSS can react to the attack. Therefore, the IPFS gateway has to be reachable for the other instances. The sender has to provide the data to at least one instance, and afterwards, the original sender can disappear out of the network. The file is still reachable from at least one other instance. Once an instance has retrieved the data, it should provide the data to other instances demanding for it. Also, the blockchain must be able to send a transaction to the network for spreading the identifier of the file in IPFS to the other instances.

A possible solution would be to use dedicated internet access for the needs of BloSS. Another approach, more suitable for larger setups, is to run the IPFS gateway on a different server system and therefore disburden the system hosting BloSS of the load from IPFS. This system should not be located behind the network access regulated by BloSS.

Furthermore, an idea is the usage of the Interplanetary Name System (IPNS). IPNS allows to address a regularly available namespace, and therefore another system could poll and check the defined namespaces for updates. If it detects updates, it will add the file to the IPFS network, and because the same hash value as the directly attached file will result, the file is reachable on the IPFS network.

3.8 File Format for Storing the Data

BloSS generates datasets out of two IPv4-addresses or subnets: The address of the victim and the address of the attacker. Both of these addresses have to get transmitted. Multiple approaches are evaluated in the following paragraphs like saving the data in a JSON format, comma separated or in a file-based database like SQLite.

The most straightforward way would be the use of a comma separated text file separating attacker and victim in multiple columns. The drawback of this method is the high redundancy of the victims' address. This issue would have to get eliminated by compression, which is an additional computing effort.

The usage of a file-based database like SQLite provides advantages like simple searching and subsetting, but it also produces a lot of data overhead and additional filesize for this information is needed.

The JSON[3] format was developed for exchanging object information among applications. It has a lightweight design and allows minifying operations. It enables the creation of a list of victims and the assignment multiple attackers to a victim. Many tools and plugins are available for handling JSON data.

For this application, the JSON data format was chosen, because it results in the smallest file size to transmit data and it provides an easy data handling in the python ecosystem around BloSS.

Figure 3.3 shows an example how a possible JSON file could look like.

For every host, there is a list of attackers assigned. The example in Figure 3.3 shows two attacked hosts listed behind the keyword `host` and each of them has a list of four `attacker` assigned to them. The code can also get minified by escaping all shiftings and line breaks to reduce the data size.

```
{
  host: "130.60.7.61",
  attacker: {
   "123.123.125.123",
    "190.23.65.86",
    "169.56.85.74",
    "69.58.74.85"
    }
},
{
  host: "130.60.7.62",
  attacker: {
    "123.1.123.123",
    "190.23.65.85",
    "169.56.85.72",
    "69.53.74.95"
    }
}
```

Figure 3.3: Example of storing IP-address lists formatted as JSON objects

3.9 Blockchain

The BloSS uses the Ethereum blockchain for its operations. Besides the IP-address list, many other features are based on the blockchain. Therefore, Ethereum is also used as the blockchain for an off-chain storage solution.

For performance measurement purposes, the Ethereum test network Rinkeby was used to measure the performance of the off-chain storage setup. Rinkeby provides us the same possibilities as the Ethereum main network, but it is free of charge to use. Five nodes mine the blocks on Rinkeby in a proof of authority manner[8]. Every 15 seconds a new block will get mined. The main network uses a proof of work principle and therefore the time between two blocks and the block containing the transaction cannot be predicted exactly. Compared to the Ethereum main network, the Rinkeby network has much fewer participants to interact and exchange blocks with.

As a client to the Ethereum network figures the application `geth` written in Go. This client is a reference implementation of the functions of Ethereum[13].

3.9.1 Smart Contracts

The smart contract (SC) is deployed on the Ethereum blockchain and therefore every instance of BloSS can interact with it. The implemented solution uses an SC to communicate the identifier of the file in IPFS to other instances. The Figure 3.4 shows the implementation of a simple SC for measuring the performance of the off-chain storage[13].

In the smart contract, four variables are defined and allow to store data. Each instance of BloSS has deployed its own SC. In the variable `owner`, the account number of the owner of this contract is saved. The variable `ipfsHash` stores the actual path to a file on IPFS. For future usage, it is also possible to add a hash value of a directory to transfer multiple files (*e.g.*, with related materials).

```
pragma solidity ^0.4.0;

contract Offchain {
  address owner;
  string public ipfsHash;
  string public sendTime;
  string public ipfsAddress;

  function Offchain() {
   ipfsHash = 'initialHash';
   sendTime = 'initialTime';
   ipfsAddress = 'initial';
   owner = msg.sender;
  }

  function setHash(string _hash, string _time) public {
    ipfsHash = _hash;
    sendTime = _time;
  }

  function getHash() constant returns (string) {
    return ipfsHash;
  }

  function getSendTime() constant returns (string) {
    return sendTime;
  }

  function setIPFSAddress(string _addr) public {
    ipfsAddress = _addr;
  }

  function getIPFSAddress() constant returns (string) {
    return ipfsAddress;
  }
}
```

Figure 3.4: Smart Contract for communicating informations on the blockchain

While `owner` contains the address in the blockchain ecosystem, `ipfsAddress` includes the address for the IPFS as a multiaddress. This multiaddress consists of the protocol, the IP-address, the port number, and the unique identifier of this node. The first part of the address can change when using dynamic IP-addresses. The hash value at the end is connected with the IPFS instance.

The variable `sendTime` is only used for the performance tests. For every update of the `ipfsHash` this variable is filled with the actual timestamp, and therefore the receiving node can calculate the time required for the whole process.

In the constructor of this contract, the variables are set to initial values for preventing errors before the first send operation. The functions below are responsible for interaction with the SC. These interactions can involve changing the hash value or retrieving information out of it.

3.10 Measurement

The performance analysis is a key aspect of this thesis. Measurements were made to gather sufficient evidence for the behavior of the system.

To determine the performance of the solution, running time is the crucial measurement variable. The following Chapter 4 provides precise information about the resulting values. In the next sections, different possible measurement approaches are discussed.

3.10.1 Overall Duration

The overall duration of transaction transmission is the critical value from the point of view of the application implementing it. It quantifies the time used between sending the file to IPFS, receiving a hash value in return, updating the smart contract with this hash value, receiving the updated value out of a mined transaction on another instance, and retrieving the file over IPFS. This time depends on many parameters. Some of them are influenced by the setup. The file size, for example, depends on the number of addresses to report and therefore influences the transmission time significantly. Another factor is the location of the instances resulting in different distances and networking resources in between. The block time, which is the time used until a block is created and added to the chain, could have a significant influence on the total duration of this process.

To provide correct time measurements, the local times of the involved instances have to be synchronized.

3.10.2 Duration of collecting Data from IPFS

In the beginning, adding a file to the IPFS ecosystem is a local operation. The file to add gets hashed and stored in the local IPFS datastore. Then the distributed hash table is updated with the value of the attached file.

For the local operations, we can assume a static time used because this differs only in available local system resources. The time used for transferring the file to another instance can vary a lot. It depends on the location and on the sender of the file as well as of the networking capacities between these instances.

3.11 Module Design

The implemented prototype for measuring the performance of the integration of IPFS into BloSS is split into different modules. The modules `ipfsConn.py` and `ethConn.py` are designed for future usage in BloSS and contain all required functions. The other modules are specific for the measurements and not designed for further usage outside of this thesis. In the next sections, the modules and their public methods are shortly described.

3.11.0.1 ipfsConn.py

This module is responsible for the communication with IPFS. It provides three functions presented in Table 3.1 for interacting with the IPFS gateway and requires an available gateway. The connection details are configured in the header of the file.

Method	Description
ipfs_add	Adds a file to the IPFS network and returns the hash value figuring as a pointer to this file.
ipfs_get	Retrievs a file from IPFS by using a given hash value as identifier.
ipfs_swarm	Adds a multiaddress to the IPFS swarm to establish a connection to this node.

Table 3.1: Methods of ipfsConn.py

3.11.0.2 ethConn.py

This module is responsible for the interaction with the Ethereum blockchain. Like the IPFS module, it needs a connection to a gateway, whose connection details are configured in the header of the file. This module contains the smart contract used for the signaling of the files and methods to modify or retrieve the variables in the contract. The Table 3.2 presents and describes the available methods.

Method	Description
deploy_contract	Deploys a contract to the blockchain and saves the address as the own contract.
get_own_contract	Returns a list of all known contract addresses from BloSS systems.
set_ipfs_hash	Returns the hash value of the own contract.
get_ipfs_hash	Returns the actual hash value stored in the smart contract.
set_ipfs_address	Writes a multiaddress representing the current address of the IPFS gateway to the contract.
get_ipfs_address	Returns the multiaddress of the IPFS gateway from the contract owner.

Table 3.2: Methods of ethConn.py

3.11.0.3 receive.py

In this module, the receiving instance continually polls the smart contracts for changes. If a new transaction is detected, it tries to load the file from IPFS. The time used for all these steps is measured and written to a file for later evaluation.

3.11.0.4 send.py

While this module is running, it creates dummy files representing the lists from BloSS, publishes these to IPFS and creates a new transactions on the blockchain.

Chapter 4

Evaluation

Based on the system design presented in the Chapter 3, this chapter will outline the methodology and results of the empirical test runs.

At the beginning of this chapter, in Section 4.1 the used systems, providers, and hardware specifications are presented. System measurements like ping roundtrip time and bandwidth of the servers to the internet are presented in Sections 4.1.2 and 4.1.3.

The setup specification and results from locally performed tests to discover the limits of this setup relating file size and simultaneous transfer are presented in Section 4.2.

Also, worldwide measurements were made to reveal performance at a higher scale. The setup was enlarged to eight nodes, spread all over the world, to identify the challenges and influences of vast distances, high delays, network crossing, and cloud computing. The findings of these tests are presented in Section 4.4 and the occurred problems discussed in Section 4.5

4.1 Underlying Hardware

The measurements used many different types of computing and networking capacities. Table 4.1 presents the hardware specifications of the underlying hardware for these tests.

Type	local machine*	Server Irchel*	VPS M*[10]	t2.micro[11]	t2.small[11]	B1S[7]	B1MS[7]	small[12]]	medium[12]
Provider	own Hardware	own Hardware	Contabo	Amazon EC2	Amazon EC2	Microsoft Azure	Microsoft Azure	DigitalOcean	DigitalOcean
(v)CPU	4	12	2	1	1	1	1	1	2
Memory	4 GB	32 GB	6 GB	1 GB	2 GB	1 GB	2 GB	1 GB	2 GB

Table 4.1: Specification of the used hardware for the tests

The locally available machines have better capabilities regarding central processing power and memory. The instances marked with a star have a simultaneous load of other services or a graphical interface in case of local desktop machines.

The instances of the cloud provider are chosen to have as similar specifications as possible. The first trial runs were performed with instances equipped with instances having 1 GB

of memory like the Amazon t2.micro, Microsoft Azure B1S, and DigitalOcean small. Due to a lack of memory in certain situations of the previously mentioned instance types, the worldwide test runs on servers of the type Amazon t2.small, Microsoft Azure B1MS and DigitalOcean medium. All of these instances are equipped with an amount of 2 GB of memory each. The difference between AWS, Azure, and DigitalOcean is the computing power. While AWS and Azure have one virtual CPU, DigitalOcean is equipped with two vCPU and can, therefore, process two threads simultaneously. A virtual CPU on cloud infrastructure is not comparable with a hardware processor core and describes the claim of using a physical core for a certain time. A so-called CPU-credit is used for regulating the CPU usage. The credit gets charged for usage and recharged by the hourly payment fees.

4.1.1 Instances for Local Tests

For the local tests inside the network at the University of Zurich, overall six instances were used.

The local machines were situated at the Institute of Informatics and the servers in the central server room of the University of Zurich. The distance in between amounts to 3 km, and the bandwidth is about 200 Mbit per second in both directions. Located at the Institute of Informatics are two local machines attached to the network over Wireless LAN. At the server room four instances were running. They are virtualized using LXD and connected to each other over GBit ethernet.

4.1.2 Instances for the Worldwide Test

For testing the behavior of the system in a network spread globally, eight instances were set up for measuring data transfer times among themselves. The Table 4.2 shows their location, resources according to Section 4.1, and their bandwidth.

Location	Australia	Brazil	Brazil (San Paulo)	Germany (Munich)	Asia (Singapore)	Japan	USA (New York)	Switzerland (Zurich)
Instance type	AWS t2.small	Azure B1MS	AWS t2.small	Contabo VPS M	Azure B1MS	Azure B1MS	DigitalOcean medium	own hardware
Bandwith	350 Mbit/s	225 Mbit/s	180 Mbit/s	90 Mbit/s	400 Mbit/s	180 Mbit/s	170 Mbit/s	310 Mbit/s

Table 4.2: Specifications of the used instances for the worldwide test.

Two own servers, located in Switzerland and Germany, were used for the tests. The server which was farthest away from these was located in Australia. In Asia, two servers were run in Singapore and Japan. Three other servers were located in America. One of them was located in North America, close to New York, and two servers were located in Brasil, near São Paulo. These servers were from the different providers AWS and Azure. At the time of the test, the major cloud providers were not capable to provide instances in the continent of Africa.

The bandwidth was measured using the packet `speedtest-cli` from the Ubuntu repository. The speed test was made by using the server with the smallest latency from the list of `speedtest.net`. The value represents the approximately and minimum measured

value out of up- and download in a batch of ten measurements. The lowest measured bandwidth is from Contabo with 90 Mbit/s. Contabo is restricting the available bandwidth by contract to 100 Mbit/s. All other instances have at least a bandwidth of 170 Mbit/s. Regarding the small filesizes of under 200 kilobytes in the test, the differences in bandwidth connections are insignificant.

4.1.3 Ping Requests

The time used for a ping roundtrip gives a estimated lower bound for needed time to transfer a file between two instances. The measurements in Table 4.3 are average values of 100 request at the interval of 10 seconds.

from \ to	Australia	Brazil (AWS)	Germany	USA	Switzerland
Australia	-	344 ms	315 ms	248 ms	351 ms
Brazil (AWS)	337 ms	-	229 ms	130 ms	220 ms
Brazil (Azure)	366 ms	6 ms	224 ms	116 ms	204 ms
Germany	333 ms	227 ms	-	103 ms	21 ms
Asia	197 ms	326 ms	279 ms	227 ms	260 ms
Japan	153 ms	266 ms	334 ms	171 ms	170 ms
USA	247 ms	128 ms	104 ms	-	96 ms
Switzerland	347 ms	221 ms	20 ms	94 ms	-

Table 4.3: Ping time between different instances in the worldwide test.

Microsoft Azure does not allow incoming ICMP traffic. Therefore, a ping request is not possible for every node.

The measurement results are reflecting the distances and network hops between the different servers. Noticeable is the fast time between the nodes of Azure and AWS in Brasil. Despite being rented from various providers, they seem to be located in the same datacenter. Based on this example, we can also see the influence of direct links between the datacenters from a provider. The measurement transfer time between servers located in Brazil and Australia was faster when they are rented from the same provider Amazon than when they are from different providers. Within European servers and American servers, fast connections seem to exist.

The measurements show quite symmetric results. This means that the required time from Brazil to Switzerland is approximately the same as from Switzerland to Brazil.

4.2 Local Tests

Before performing worldwide tests, some local tests were made to determine the performance limits of IPFS and the Rinkeby Ethereum test network. Figure 4.1 shows four

test runs with different file size and 100 measurements each. The red line represents the median value of the corresponding test set.

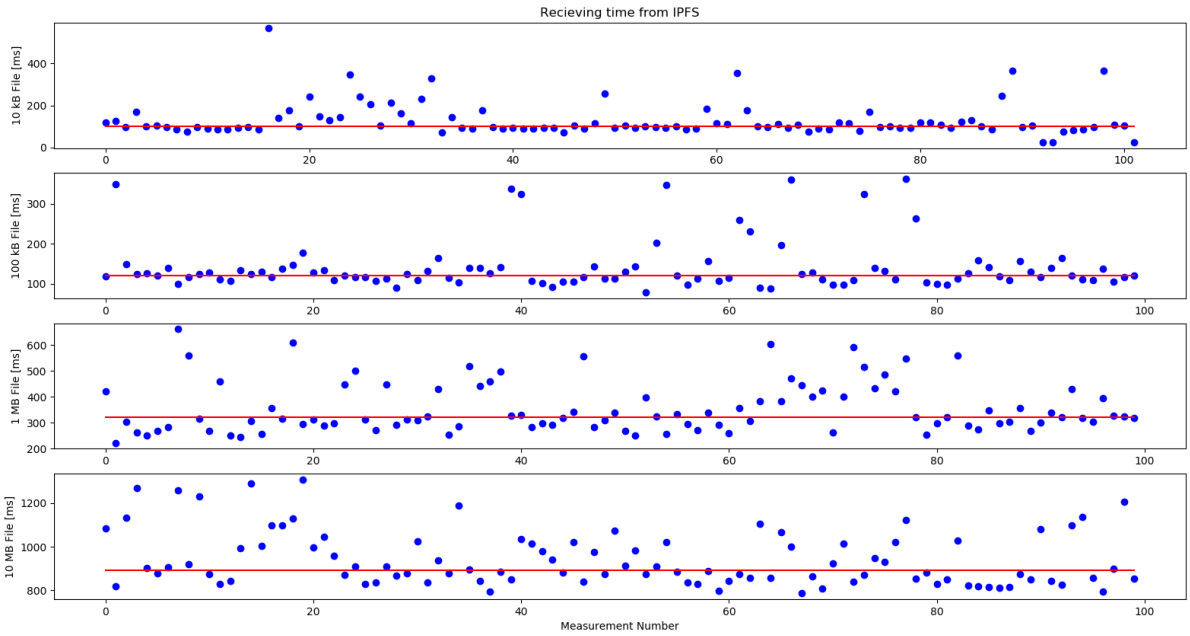

Figure 4.1: Time used for a transaction in a local network depending on the filesize.

The median time for transporting a file of 10 kB size is about 100 milliseconds and for a file of 100 kB approximately 120 milliseconds. For the 1 MB block, around 320 ms is required. As a pattern, we can extract a base time of 75 ms and additional 25 ms per 100 kb.

The maximum block size is 1 MB. Therefore, a file of 10 MB size needs to be split up into multiple blocks. A serialized transmission would thus result in 10 times 320 ms, resulting in 3200 ms, but the measurements captured only 900 ms. The transmission time improvement is due to the simultaneous transfer of multiple files which increased the transfer rate.

The tests run up to filesizes of 4 GB without causing problems on the transfer of the data.

On the sending side, between 40 and 100ms is used to add the hash value to the contract and send the transaction to the blockchain network. IPFS uses another between 70 and 140 ms to publish the file. This time includes copying the data locally to the IPFS directory, retrieving the hash value of the file, splitting blocks and updating the hash table.

An additional test was made by adding ten files of 1 MB size simultaneously to IPFS, writing the hash value to the same contract, and retrieving the data. The measured time was with 950ms on the median only slightly over the transmission time of a regular 10 MB file.

4.3 Overall Duration

The most relevant value for the end-user, respectively in this case the BloSS, is the time used over the whole process, from adding the file to IPFS until the other node has received the file from IPFS. Figure 4.2 shows the distribution of time used for this process. In the bottom part of the figure, the range below 40 seconds of the upper histogram is zoomed in.

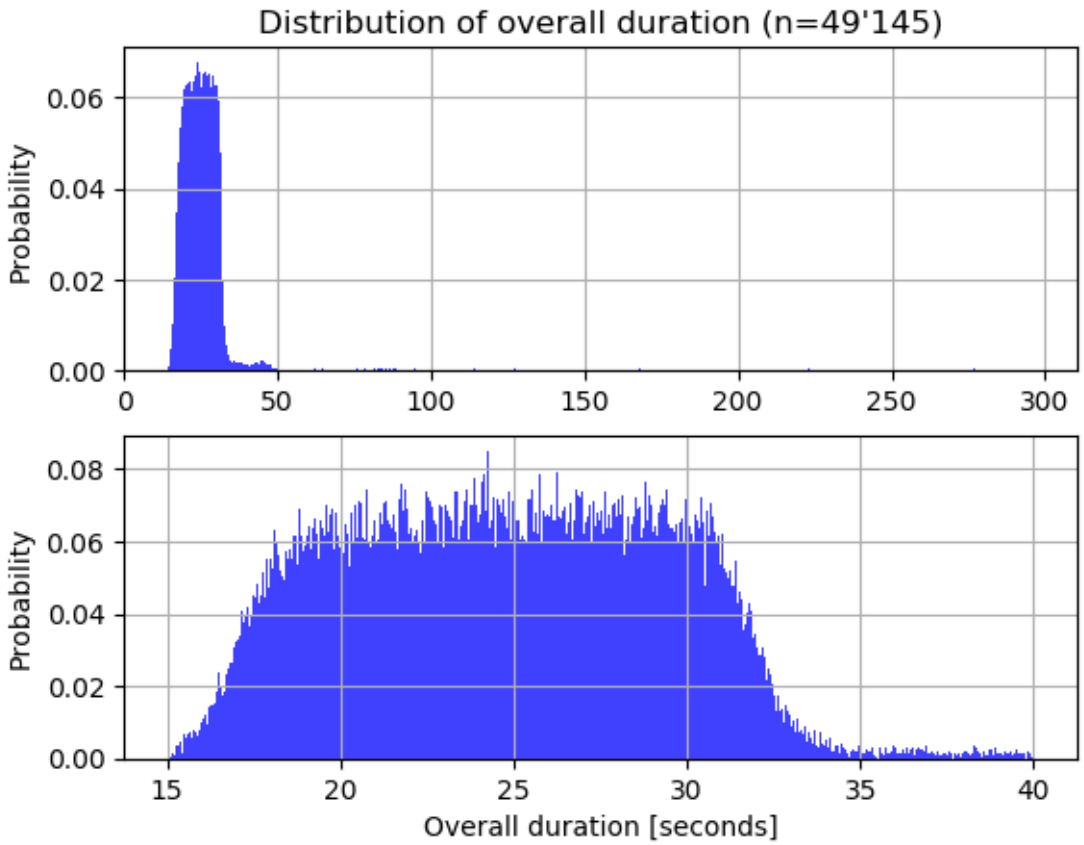

Figure 4.2: Distribution of the overall time used for a transaction.

More than 95% of the measurement values are between 15 and 35 seconds. The fastest transmission used 14.1 seconds, and the slowest ones had up to 3 minutes. Considering that, every 15 seconds in the Rinkeby network, a block gets mined, most of the transacts got mined in the first or second block after transaction creation.

Looking at the long-lasting transactions, a failure on the receiving node in most cases caused the problem because of a drop-out or a transaction reorganization of the `geth` client.

A measurement of more than 300 seconds is not possible because the value stored in the contract would be overwritten by the next transaction of the still sending node. Other sources of errors were the restart of the operating system and restarting the instance after adding the instance in Brazil on the Azure Cloud.

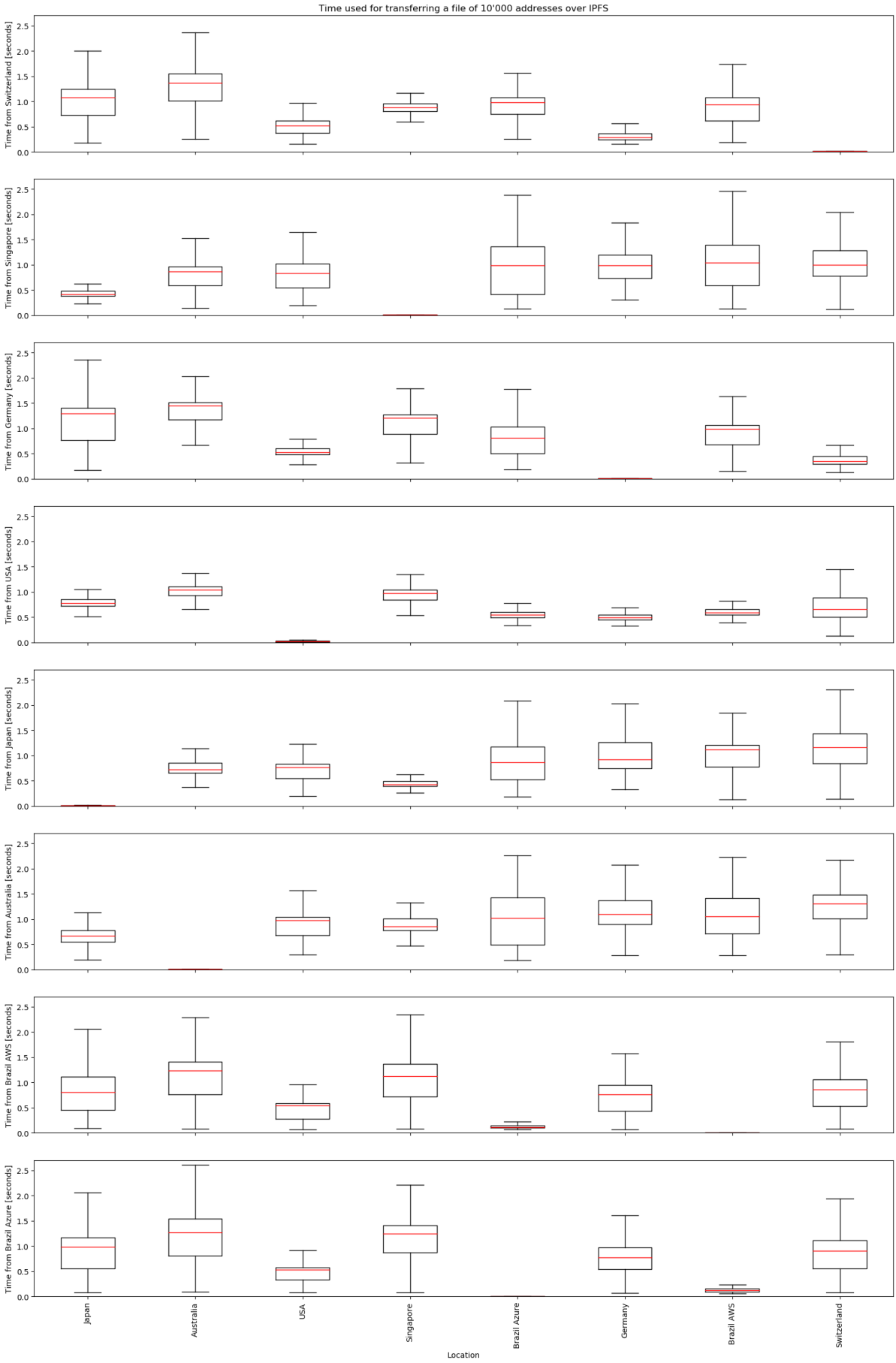

Figure 4.3: Used time for transferring a file of 10'000 addresses between two locations.

4.4 IPFS in worldwide Usage

The primary measurements to evaluate the performance of IPFS as off-chain storage tool is the time used for transferring a file between two instances distributed over different places on the world. In Figure 4.3 the time used for successfully transferring a file from a sender to another receiver is displayed. Every file contains 10'000 randomly generated IPv4-addresses and has a size of about 173 kB. The file got published by an instance, and all other instances will retrieve this file. In the BloSS environment, this would represent a BloSS instance with an attacked network assigned to as the publisher of the file and the other instances of BloSS retrieving the file to support the mitigation of the attack.

In general, the proportions between the transfer times are distributed like the results from the ping measurements in Section 4.1.3. The transmission time of nearer located nodes is, in general, lower than between farther distanced instances. However, in contrast to Section 4.1.3 the results are not in all cases symmetric. The difference in transfer time could be rooted in the distributed peer-to-peer approach. If the block is already available from a nearly located node, the desired data could get retrieved from there and hence a lower transfer time could result. The fast measurements values from these transmissions are reducing the mean values. The constellation of nearby located instances is the case for the server pairs Switzerland-Germany, Brazil-Brazil, and, benefitting from the fast overseas cables, USA-Germany and USA-Switzerland.

For excluding measurements resulting from system failures, the outliers more than 1.5 interquartile ranges away from the second, respectively third quartile, are escaped. However, a significant spread of the transfer times results. The maximum time used is 2.6 seconds. There are a lot of much lower results going down to about 200 ms in the median. The average values by instance regardless the author of the files are less varied between 0.9 and 1.2 seconds.

Notable is the rapid transmission between both nodes in Brazil transferring the file in below 200 ms most of the time. These nodes seem to be nearly located to each other even though different providers provide them.

The measurements for retrieving a file from itself indicate the portion of the time an instance uses for its own operations and includes a query with one hop on the DHT. The transfer time is typically between 4 and 80 ms in this tests.

4.4.1 Acting with larger Datasets

The previous measurements contain a dataset of 10'000 IPv4 addresses as a payload. This number of addresses is a rough estimate of the data that BloSS normally generate. In times of a heavy attack, like the DDoS attack on the French provider OVH in 2016 with over 150'000 IoT devices involved[15], larger files have to be handled. The Figure 4.4 shows the measured values of four instances in comparison of 150'000 to 10'000 IP addresses as in the previously mentioned tests.

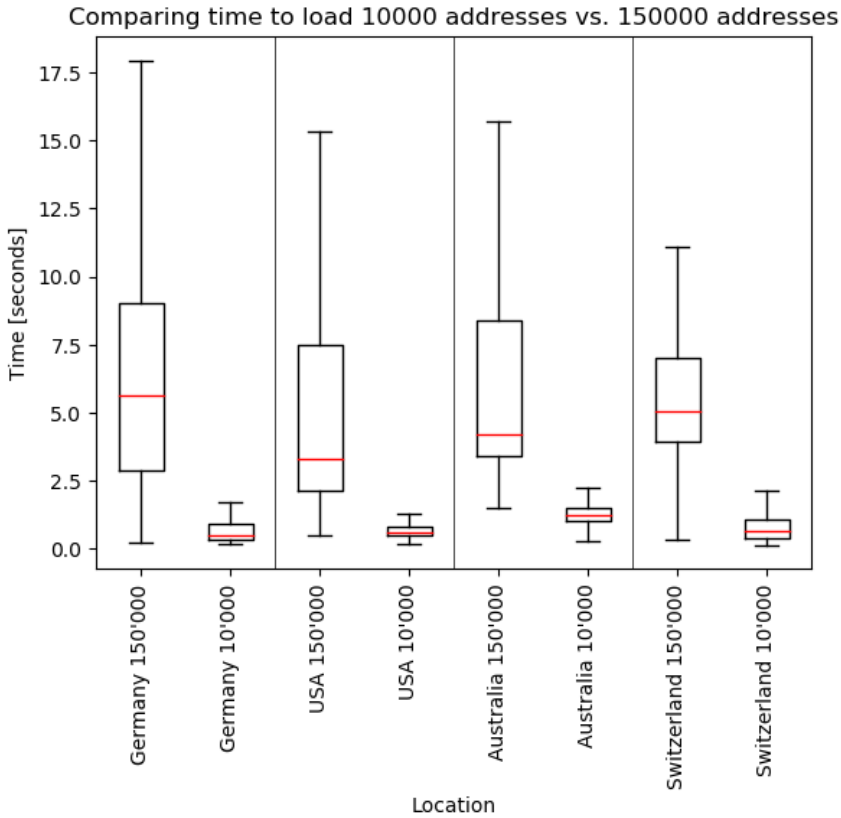

Figure 4.4: Difference in time transmitting a file of 150'000 compared to 10'000 addresses.

As we see in the box plots of Figure 4.4 the mean values for the larger file is between four and ten times higher for transmitting a file which is 15 times larger. With files of over 1 MB size, we have the advantage of IPFS splitting such files into multiple blocks. These blocks are transported independent of each other and can be loaded from a different node. For example, for a file published in Australia, the node in Switzerland can retrieve the first block from the USA, which already has loaded this block, some blocks from Australia and another block from Germany, which has charged this part in the meanwhile. This behavior allows to minimize the usage of distant nodes and interact with close-by nodes more often. On the other hand, the spread of the measurements is much higher. In the best case, all blocks are available from a close-by instance and the transfer time will be relatively low. It can happen that all nodes try to receive the data at the same time from one node which gets congested at this moment and can serve only two requests simultaneously. In this case, some nodes have to wait until resources of the sender are available. As a result, the used time increases strongly.

4.4.2 Comparison with HTTP

For comparing the results with other technologies, measurements using the data transmission over HTTP were made. Figure 4.5 shows the used time comparing HTTP and IPFS.

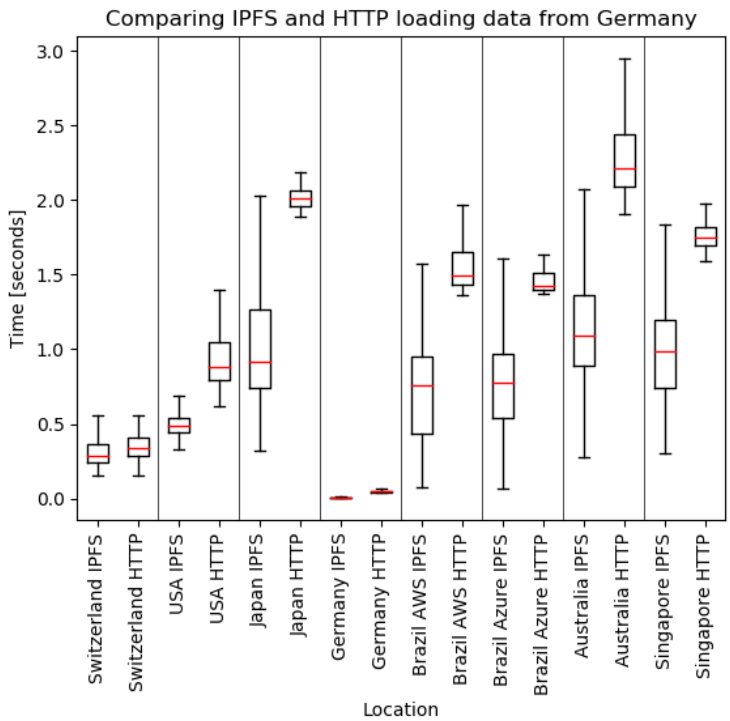

Figure 4.5: Comparison of HTTP and IPFS

The measurement was made from the instance in Germany providing the file and all eight nodes requesting the file. Every node requested the file more than 120 times. In general, the transmission over HTTP was slower than over IPFS. The difference between the time used over HTTP and IPFS is strongly increased by the distance between the two instances. Noticeable is the difference in the spread of the two nodes in Brasil, probably caused by the different overseas cable of the providers.

4.5 Failures

During the test runs, different failures occurred. In the next two paragraphs, the main problems, the outage of memory and missing peers for IPFS, are described.

4.5.1 Lack of Memory

The first try was to run the test on cloud servers with 1 GB of memory. In most cases, the IPFS daemon crashed shortly after starting the tests. This was caused by a lack of memory. The geth client and IPFS daemon allocated a lot of memory and the IPFS daemon requested far more memory than unallocated memory is available and the geth client will not release any memory during block retrieval. In this situation, the system is in a locked state and the operating system kills the IPFS daemon process after a few minutes to escape the deadlock state.

At the beginning the nodes were running on Amazon EC2 t2.small, Microsoft Azure B1S and DigitalOcean small instances with 1 GB of memory as shown in Table 4.1. After over 20 failures on all affected nodes, the instances were upgraded to Amazon EC2 t2.medium, Microsoft Azure B1MS and DigitalOcean medium instances which provide 2 GB of memory. This size was enough to run the geth client and IPFS daemon simultaneously on the same machine on the AWS. On the Microsoft Azure cloud, the five servers crashed in total seven times. As we can see in the screenshot of the console from the DigitalOcean droplet, shown in Figure 4.6, even after the upgrade it crashed several times.

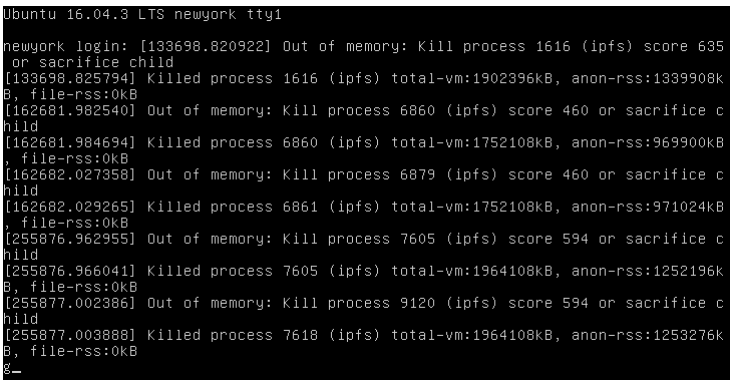

Figure 4.6: Screenshot of the console showing killed processes of IPFS.

The distribution of the used memory (besides the desires of the operation system) is spread in the following proportion: two parts for the IPFS daemon and one part for the geth client with a deviation of up to 15 percent of this ratio. In the time of block indexing of the geth client, a lot more memory was used for this process.

The problem of the memory usage is already reported to the developers of the IPFS Go implementation on Github[18]. At the moment a more stable system can be built by starting the IPFS daemon as a service. This service would restart the daemon by itself after killing the process and therefore resulting in an outage of only about 10 seconds.

4.5.2 IPFS cannot find hash

In the first operation, a node located in Australia tried to receive a file from Switzerland, but IPFS was not able to find the file in less than an hour. The peer-to-peer system could not find an entry in the distributed Kadmlia hash table to resolve the request of the file to an owner. To solve this issue, in the installation script a bootstrap node will now get added to the IPFS swarm manually. This improvement resolved the issue at it's best. For other nodes located nearer and in environments of higher IPFS usage like from Germany to Switzerland, it was no problem to find the corresponding node owning the file. An additional improvement made after the tests to the codebase. It is now possible to define the multiaddress of IPFS in the smart contract, and all instances can add this address directly to their connection list in the swarm. Therefore, a dedicated network with connections between the BloSS instances can get established fast.

Chapter 5

Summary and Conclusions

The purpose of this thesis was to evaluate a suitable solution for an off-chain data storage tool for the Blockchain Signaling System. The Interplanetary File System was found to be the solution which, compared with Swarm, StorJ, and CoAP, fulfills the requirements of BloSS best. A system based on IPFS and the Ethereum test network Rinkeby was built using Python3 and the clients of IPFS and Etherum written in Go on the operation system Ubuntu. This system was extended by a data generator for the test data and used for performance measurements.

These performance measurements were conducted in a local and a worldwide spread setup. The local tests inside of the network of the University of Zurich showed that it was possible to transfer files up to 4 GB size and ten files simultaneously without failure.

For the globally distributed test, the setup was enlarged to eight instances. These instances were running on cloud servers rented from various providers distributed over five continents. A typical task in the test for the setup was to transmit over IPFS a file containing 10'000 IPv4-addresses (173kB). The resulting transfer time in this test varied between below 0.2 and 2.6 seconds. A finding is that the time to transfer a file increases with the distance between the instances and is reduced if two instances are located near to each other or have fast network connections in between. Nearer located instances benefit from the usage of a peer-to-peer network. It is sometimes possible to retrieve the data from an instance which is nearer located than the original author.

A more massive test set with 150'000 addresses was, according to the number of attackers in one of most intensive attacks in the past, used to test the capability of reporting high amounts of addresses. In this test, the transfer time increased up to 20 seconds without interfering the service.

In comparison of transmitting data over HTTP, the transmission time using IPFS was shorter. With longer distances, the advantages of using IPFS in contrast to HTTP increases.

Overall, from adding the file to IPFS until receiving it again from IPFS on another instance (including blockchain operations) more than 95% of the measurements lasted between 15 and 35 seconds.

Improvements can be made in stability, security, and collaboration between the instances. Regarding the stability, the IPFS client crashed several times, mostly because of a lack of memory on the host system in combination with the memory usage of the Ethereum client. Improvements can be made directly in the application or, combating the symptoms, by controlling the service management. In the implemented solution, the security of the smart contract and the file content on IPFS was not considered and must be addressed prior to future productive usage. For the handling of the smart contract addresses, a solution using an main contract could simplify the management of the participants.

In general, IPFS is a good solution for off-chain storage. Even though, many opportunities to improve this setup exist. IPFS works very well for the BloSS use case, and the existing problems of IPFS are already getting addressed by a large open-source community.

Bibliography

[1] Kevin J Baird. *Can I share a file with another Storj user?* June 2017. URL: https://docs.storj.io/v1.1/discuss/5942dd69e0a850000ffd92f9.

[2] Juan Benet. *Ipfs-content addressed, versioned, p2p file system.* 2014.

[3] Tim Bray. *The javascript object notation (json) data interchange format.* 2017.

[4] Jason Carver. *Remove py2 from supported languages in pypi.* Nov. 2017. URL: https://github.com/pipermerriam/web3.py/commit/2822ab889bda4c5fe7079595d12a9678c4ba1771#diff-2eeaed663bd0d25b7e608891384b7298.

[5] Bastien Confais, Adrien Lebre, and Benoît Parrein. "An object store for Fog infrastructures based on IPFS and a Scale-Out NAS". In: *RESCOM 2017.* 2017, p. 2.

[6] Bastien Confais, Adrien Lebre, and Benoît Parrein. "Performance Analysis of Object Store Systems in a Fog and Edge Computing Infrastructure". In: *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXIII.* Springer, 2017, pp. 40–79.

[7] Microsoft Corperation. *Virtuelle Linux Computer.* Jan. 2018. URL: https://azure.microsoft.com/de-de/pricing/details/virtual-machines/linux/.

[8] Etherscan. *Ethereum Top Miners Stats.* Jan. 2018. URL: https://rinkeby.etherscan.io/stat/miner?range=365&blocktype=blocks.

[9] Ethereum Foundation. *Swarm Distributed Data Storage.* Dec. 2017. URL: https://github.com/ethersphere/swarm.

[10] Contabo GmbH. *Cheap Windows & Linux hosting at an affordable price.* Jan. 2018. URL: https://contabo.com/?show=vps.

[11] Amazon Web Services Inc. *Amazon EC2 Instance Types.* Jan. 2018. URL: https://aws.amazon.com/ec2/instance-types/?nc1=h_ls.

[12] DigitalOcean Inc. *Compute on DigitalOcean.* Jan. 2018. URL: https://www.digitalocean.com/products/compute/.

[13] Martin Michelmayr. *ethereum/go-ethereum: Official Go implementation of the Ethereum protocol.* Nov. 2017. URL: https://github.com/ethereum/go-ethereum.

[14] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system.* 2008.

[15] Pierlugini Paganini. *150,000 IoT Devices behind the 1Tbps DDoS attack on OVH.* Sept. 2016. URL: http://securityaffairs.co/wordpress/51726/cyber-crime/ovh-hit-botnet-iot.html.

[16] Bruno Rodrigues, Thomas Bocek, and Burkhard Stiller. *Enabling a Cooperative, Multi-domain DDoS Defense by a Blockchain Signaling System (BloSS).*

[17] Bruno Rodrigues et al. "A Blockchain-Based Architecture for Collaborative DDoS Mitigation with Smart Contracts". In: *IFIP International Conference on Autonomous Infrastructure, Management and Security*. 2017, pp. 16–29.

[18] Hector Sanjuan. *ipfs daemon memory usage grows overtime: killed by OOM after a 10 12 days running*. Dec. 2016. URL: `https://github.com/ipfs/go-ipfs/issues/3532`.

[19] Shelby, Zach and Hartke, Klaus and Bormann, Carsten. *The constrained application protocol (CoAP)*. 2014.

[20] Elfriede Sixt. "Ethereum". In: *Bitcoins und andere dezentrale Transaktionssysteme*. Springer, 2017, pp. 189–194.

[21] Storjblog. *Getting From Petabytes to Exabytes: The Road Ahead*. Jan. 2018. URL: `http://blog.storj.io/post/169896892413/getting-from-petabytes-to-exabytes-the-road-ahead`.

[22] Viktor Tron et al. *Swarm documentation Release 0.2rc5*. 2017.

[23] Shawn Wilkinson et al. *Storj a peer-to-peer cloud storage network*. 2014.

[24] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum Project Yellow Paper* 151 (2014).

[25] Gavin Wood. *Proof of Authority Chains*. May 2016. URL: `https://github.com/paritytech/parity/wiki/Proof-of-Authority-Chains`.

Abbreviations

AWS	Amazon Web Service
Azure	Microsoft Azure Cloud Services
BloSS	Blockchain signaling System
CoAP	Constrained Application Protocol
DAG	Directed acyclic Graph
DApp	Decentralized Applikation
DDoS	Distributed Denial of Service
DHT	Distributed Hash Table
ETH	Ether (Currency)
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IoT	Internet of Things
IPFS	Interplanetary File System
IPNS	Interplanetary Name System
IPv4	Internet Protocol version 4
JSON	Javascript Object Notation
NTP	Network Time Protocol
REST	Representational State Transfer
SC	Smart Contract
UDP	User Datagram Protocol

List of Figures

2.1 Example of an IPFS multiaddress. 5

3.1 Sequence diagramm of publishing a file on IPFS and the blockchain. 13

3.2 Sequence diagram of receiving a file from the blockchain and IPFS. 13

3.3 Example of storing IP-address lists formatted as JSON objects 16

3.4 Smart Contract for communicating informations on the blockchain 17

4.1 Time used for a transaction in a local network depending on the filesize. . 24

4.2 Distribution of the overall time used for a transaction. 25

4.3 Used time for transferring a file of 10'000 addresses between two locations. 26

4.4 Difference in time transmitting a file of 150'000 compared to 10'000 addresses. 28

4.5 Comparison of HTTP and IPFS 29

4.6 Screenshot of the console showing killed processes of IPFS. 30

List of Tables

2.1 Mean time using IPFS in function of the latency (from Confais *et al.*[5]). . 7

3.1 Methods of ipfsConn.py 19

3.2 Methods of ethConn.py 19

4.1 Specification of the used hardware for the tests 21

4.2 Specifications of the used instances for the worldwide test. 22

4.3 Ping time between different instances in the worldwide test. 23

Appendix A

Installation Guidelines

This installation guideline guides you through the setup of the used measurement system.

1. Setup operation system Ubuntu 16.04.
2. Load source code onto the machine.
3. Make the file install.sh executable and run it as a bash script.

   ```
   chmod +x install.sh
   ./install.sh
   ```

   This script installs the needed module, a client for the Ethereum network (geth) and IPFS. A geth instance on the Rinkeby testnet will get started as a screen session.
4. Create an account on the Rinkeby testnet.

   ```
   geth attach http://127.0.0.1:8545
   personal.createAccount('Password')
   ```
5. Adjust the password in ethConn.py to the password set in the previous step.
6. Add some ether on the account (*e.g.* by using `https://faucet.rinkeby.io/`)
7. Run setup.py

   ```
   python3 setup.py
   ```

   This script creates skeleton files for adding the measurements values, deploys a Smart Contract and writes this information to the configuration file.
8. Run the IPFS daemon

   ```
   ipfs daemon &
   ```
9. Run send.py and receive.py either in different windows or screen sessions.

   ```
   screen -d -m -S geth send python3 send.py
   screen -d -m -S geth receive python3 receive.py
   ```

   This starts the scripts in different screen sessions.

Appendix B

Contents of the CD

The enclosed CD contains the following content:

Code
This folder contains the python source code of the scripts used for the measurements. Furthermore, the installation scripts used in Appendix A are included in this folder. In the subfolder `BloSS` the provided source code of BloSS is extended by the off-chain storage.

Documentation
The LaTeX source code for this document is in this folder and all images included in this document are in the subfolder `images`.

Literature
This folder contains all in the thesis referenced papers.

Measurements
Comma seperated measurement data captioned by the instances during the tests.

Presentation
The slideset for the final presentation in February 2018 as PowerPoint document.