

KEIO UNIVERSITY

MASTER THESIS

Direct File Transfer System via WebRTC
*An Alternative to E-mail Attachments with
Improved Security*

Author:
Robin LUNDE

Chief Examiner:
Dr. Keiji TAKEDA
Co-Examiners:
Dr. Jun MURAI
Dr. Osamu NAKAMURA

*A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Media and Governance [M.M.G.]
in the*

Graduate School of Media and Governance

July 5, 2018

KEIO UNIVERSITY

Abstract

Graduate School of Media and Governance

Master of Media and Governance [M.M.G.]

Direct File Transfer System via WebRTC*An Alternative to E-mail Attachments with Improved Security*

by Robin LUNDE

Utilizing WebRTC's P2P technology, this thesis will suggest an alternative to the way e-mail attachments currently work. In today's technical environment, there are countless new and secure ways to send files over the Internet. Yet most people still use the traditional, outdated e-mail attachment technology to share files. This thesis will propose a new system that seeks to improve overall security and usability of transferring files.

The system will be evaluated against the current e-mail system, as well as cloud and SNS based solutions. The comparison will largely be focused on security and usability. This thesis will clearly show that the current e-mail system is not sufficient when it comes security, usability, or both. It will propose a solution that raises the standard of security and confidentiality, which improves the current conditions and offers an alternative solution.

It suggests a system utilizing the web of trust model combined with WebRTC's P2P functionality. By using the web of trust model, the end-user gets more control over whom he trusts, while simultaneously avoiding the problem of having to authenticate users against a central server or service. The P2P functionality of WebRTC allows for the direct transfer of files between users, avoiding the need for servers to store the files in transit. This reduces the risk of an attacker gaining access to the file, while also optimizing transfer-speed. This system was implemented in a prototype to demonstrate the feasibility of the proposed system.

Information Security

Keywords: Usable security - Trust management -
Real world cryptography - WebRTC - File transfer

Acknowledgements

“Without a struggle, there can be no progress.”

-Frederick Douglass

I would like to start off by thanking my advisor, Keiji Takeda, for his patience and contributions to my research. His advice and guidance has helped me a lot, and allowed me to do the best I could. His encouragement and belief in my research is also the main reason I have been able to get this far. I especially appreciate his willingness to help in situations not related to research!

Next, I would like to thank my family. Thank you Mom and Dad for letting me chose my own path! Going to Japan was a big change for me, and you have been nothing but supportive and optimistic, and for that I am really thankful. I would also like to thank my little brother, Benjamin, for giving me the idea of going to Japan. I would also like to give a special message to my Grandma:

Kjære Mormor:

Tusen takk for all hjelp og oppmuntring under studietiden. Vi ungdommen setter stor pris på at du alltid holder deg oppdatert og følger nøye med på hva vi driver med. Det er utrolig koselig og jeg setter stor pris på alle meldinger og postkort som kommer med tilfeldige mellomrom. Jeg er også takknemlig for alle bidrag og all støtte du har gitt oppgjennom årene. Tusen takk!

Thank you to all my friends for all the good times. You have made me feel welcome and at home in Japan and given me someone to share my experiences with. I appreciate that a lot!

In addition, I also want to thank the members of the MAUI-project as well as the members of Takeda-ken for their advice, friendliness and for suffering through my endless PowerPoint presentations.

I would also like to thank my co-advisors Jun Murai and Osamu Nakamura, as well as our research partners at Hitachi for their valued advice. Your contributions and your guidance was truly appreciated.

Finally, I would like to thank my girlfriend for her love and support! Thank you for all the help when I was stressed out, making dinner, doing laundry and everything else you did, so I could focus on my work. Thank you so much - I am not sure I would have survived without you!

*Dedicated to my grandparents:
Kitt, Thor, Synnøve & Arne*

Contents

Abstract	iii
Acknowledgements	v
1 Introduction	1
1.1 Goals	1
1.2 Motivation	1
1.3 Existing solutions	2
1.3.1 E-mail attachments	3
1.3.2 Other solutions	4
1.4 Contributions	5
1.5 Organization	7
2 Technology and Basic Principles	9
2.1 Cryptography	9
2.1.1 Key management	9
2.1.2 Certificates	10
2.1.3 Authentication	10
2.2 Web of trust	11
2.2.1 Overview	11
2.2.2 Certificates	11
2.2.3 Certificate sharing	11
2.2.4 Sybil attack	12
2.3 WebRTC	12
2.3.1 Connection setup	13
2.3.2 Signaling server	14
2.4 WebSockets	15
3 Concept and Design of SendIt	17
3.1 Concepts	17
3.1.1 First trust	17
3.1.2 Trust building	18
3.1.3 Authentication	20
3.2 System design	21
3.2.1 Cryptography	21
3.2.2 Protocol	22
3.2.3 File transfer functionality	24
3.2.4 Modes	25
3.3 ACS server design	25
3.3.1 Authentication	25
3.3.2 Communication	25
3.3.3 ACS Protocol	26
3.3.4 Forwarding	30

3.3.5	Semi-asynchronous transfer	31
4	Implementation of SendIt	33
4.1	Application	33
4.1.1	Cryptography	33
4.1.2	Connection setup	34
4.1.3	File transfer functionality	34
4.1.4	Programming languages, libraries and frameworks	35
4.1.5	Program flow	36
4.2	ACS server implementation	40
4.2.1	WebSockets	41
4.2.2	Protocol	41
4.3	Extendability and improvements	44
4.3.1	Key storage encryption	44
4.3.2	E-mail verification	44
4.3.3	Support for bigger files	45
4.3.4	Resume transfer	45
4.3.5	WebRTC IDP inclusion	45
4.3.6	SendIt as a platform	46
5	Operation modes	47
5.1	Serverless mode	47
5.1.1	Use case	47
5.1.2	Program flow	48
5.1.3	Implementation specifics	51
5.1.4	Possible improvements & extendability	51
5.2	Assisted Connection Setup mode (ACS)	51
5.2.1	Use case	52
5.2.2	Program flow	52
5.2.3	Implementation specifics	54
5.2.4	Possible improvements & extendability	55
6	Experiments	57
6.1	Motivation and contribution	57
6.2	Terminology	58
6.3	Implementation and results	58
6.3.1	Experiment 1	59
6.3.2	Experiment 2	60
6.4	Impact and findings	61
7	System evaluation	63
7.1	Evaluation of SendIt	63
7.1.1	First trust	63
7.1.2	DHT	63
7.1.3	ACS	64
7.1.4	Other	64
7.2	E-mail system comparison	64
7.2.1	Direct transfer	65
7.2.2	Authentication	65
7.2.3	Content control	65
7.2.4	Connection setup	66

7.2.5	Synchronous connection	66
7.2.6	First trust	66
7.3	SNS and Cloud systems comparison	66
7.3.1	False E2E encryption	66
7.3.2	Identity swapping	66
7.3.3	Other	67
8	Conclusion	69
8.1	Summary	69
8.2	Future work	72
8.2.1	SendIt	72
8.2.2	E-mail attachments	73
8.3	Final remarks	73
A	ACS Server	75
	Bibliography	87

List of Figures

1.1	SNS: False E2E encryption	4
2.1	ICE trickling	13
2.2	DTLS setup	15
3.1	Certificate transfer and evaluation	19
3.2	Authentication exchange: Encryption	22
3.3	Authentication exchange: Decryption	23
3.4	File sharing process	24
3.5	ACS server authentication scheme	26
3.6	ACS protocol: Communication	27
3.7	ACS protocol: Lookup	28
3.8	ACS protocol: Authentication setup	29
3.9	ACS protocol: Authentication	30
4.1	Application stack	35
4.2	SendIt: Install animation	36
4.3	SendIt: First launch pop-up	36
4.4	SendIt ACS mode: Home screen	37
4.5	SendIt Serverless mode: Home screen	37
4.6	SendIt: Navigation bar	38
4.7	SendIt: Settings screen	38
4.8	SendIt: Detailed settings screen	39
4.9	SendIt: Waiting for connection screen	39
4.10	SendIt: Transfer screen	40
4.11	SendIt: Final screen (Sender)	40
4.12	SendIt: Final screen (Receiver)	40
5.1	SendIt mode comparison	47
5.2	Serverless mode: Register identity screen	48
5.3	Serverless mode: Register recipient screen	48
5.4	Serverless mode: Display Offer screen	49
5.5	Serverless mode: Input Answer screen	49
5.6	Serverless mode: Register Sender screen	50
5.7	Serverless mode: Input Offer screen	50
5.8	Serverless mode: Display Answer screen	50
5.9	ACS mode: Identity selection screen	52
5.10	ACS mode: Sender screen	53
5.11	ACS mode: Recipient screen	53
5.12	ACS mode: Error screen	54
6.1	Experiment terminology	58
6.2	Results Experiment 1	59
6.3	Results Experiment 2	60

6.4 Experiment comparison	61
-------------------------------------	----

List of Tables

3.1	ACS protocol: Basic format	28
4.1	ACS protocol: Lookup packet	41
4.2	ACS protocol: Authentication Setup packet	41
4.3	ACS protocol: Authentication Setup Reply packet	42
4.4	ACS protocol: Authentication packet	42
4.5	ACS protocol: Authentication Result packet	42
4.6	ACS protocol: Initiate Connection packet	43
4.7	ACS protocol: Accept packet	43
4.8	ACS protocol: Answer packet	43
4.9	ACS protocol: ICE packet	44
4.10	ACS protocol: Error packet	44
4.11	Record of communication	45
6.1	Comparison of Sender's delay	61
6.2	Comparison of Receiver's delay	62
6.3	Comparison of Split delay	62

List of Abbreviations

ACS	Assisted Connection Setup
AES	Advanced Encryption Standard
API	Application Programming Interface
CA	Certificate Authority
DHT	Distributed Hash Table
DOS	Denial Of Service
DTLS	Datagram Transport Layer Security
E-mail	Electronic-mail
E2E	End T(w)o End
Etc.	Et Cetera
GCM	Galois / Counter Mode
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ICE	Interactive Connectivity Establishment
IDP	IDentity Provider
IP	Internet Protocol
IPv6	Internet Protocol version 6
IV	Initialization Vector
JS	JavaScript
JSON	JavaScript Object Notation
JWK	JSON Web Key
MIME	Multipurpose Internet Mail Extensions
MITM	Man In The Middle
NAT	Network Address Translation
PGP	Pretty Good Privacy
PKI	Public Key Infrastructure
P2P	Peer T(w)o Peer
RFC	Request For Comments
SCTP	Stream Control Transmission Protocol
SDP	Session Description Protocol
S/MIME	Secure/ Multipurpose Internet Mail Extensions
SMTP	Simple Mail Transfer Protocol
SNS	Social Networking Service
SRTP	Secure Real-time Transport Protocol
STUN	Session Traversal Utilities for NAT
TLS	Transport Layer Security
TURN	Traversal Using Relay NAT
VOIP	Voice Over IP
WebRTC	Web Real-Time Communication
WS	WebSockets
XSS	Cross Site Scripting

Chapter 1

Introduction

This thesis will introduce a system that allows for easy to use, improved security when transferring files. The feasibility of the system will be demonstrated by implementing it in an application named SendIt, which stands for *Secure, Serverless, Electron & Node.js-based, Direct Information Transfer*. It aims to reduce the risk of data theft while still being usable by people lacking technical insight. It also aims to improve the security compared to commonly used solutions.

1.1 Goals

The goal of this thesis, in its simplest form, is to improve the current situation regarding file transfers, with e-mail attachments as the main target.

The current way e-mail attachments work does not sufficiently protect user privacy. Nor does it try to minimize security risks. With all the technology available in today's technical society, there is no reason as to why this should be the case since there is an abundance of available remedies. As such this thesis volunteers an alternative, in an effort to lead by example towards better security solutions.

This thesis will explore opportunities for, and suggest, a system that increases usability, privacy and security, especially targeted towards people who are less technically capable. Security is currently something that is usually limited to those with in-depth knowledge of the subject. Rarely are secure solutions made easy to use and available to the masses. Why is that? This thesis aims to clearly show how the concepts of usability and security does not have to be at odds, but can be combined into an elegant and user-friendly system.

Another goal of this thesis is to reduce the risk of leakage and minimize the exposure of personal data. This has become a more and more pressing issue in recent times, and as such it should be considered. The majority of people are becoming aware of how vulnerable they are to having their data leaked online. As such, a lot of existing solutions have come under scrutiny. This thesis will show one approach towards these goals, and illustrate clearly how they can be achieved. Hopefully, this leads to a higher awareness and better knowledge of how to reach these goals, and allows for better solutions in the future.

1.2 Motivation

When transferring files between two end-users there is no logical reason to involve any mediator and, as such, P2P communication emerges as the logical choice. In current e-mail systems the file will reside in at least three locations: respectively on the sender's file system, on the e-mail exchange server and on the receiver's file system. A recent solution that is growing more popular, is storing the file in the

cloud instead of on the e-mail exchange server. In contrast, when using SendIt the file will not reside anywhere except on the senders and receivers computers, as is the intention when transferring a file. This reduces the attack surface, which reduces the risk of data leakage.

There is also a clear global trend towards higher requirements regarding data protection and handling regulations, as demonstrated by the EU's new General Data Protection Regulation [1, 2] and Special Publication 800-171 in the US [3]. In other words, there is a strong demand to minimize exposure of personal data. With this as motivation, this thesis seeks to find solutions that is in compliance with these trends while also keeping in mind the original intended model behind the internet. The Internet began as a decentralized network, but has gradually converged into a more centralized network of servers [4]. The goal is to break with this development and move towards a decentralized system where central services are only used when strictly necessary.

SendIt is an application that is simple to use and improves the current conditions regarding e-mail attachments. The intention behind making SendIt is to demonstrate how this new technology can be used to meet the regulations, and be in compliance with the model previously mentioned. It also demonstrates that even people unfamiliar with the technology can utilize it to protect their privacy and improve their overall security while transferring files. The benefit of making such an application is to demonstrate how new technology can be used, and to show that even people unfamiliar with the technology can utilize it to protect their privacy and improve their overall security while transferring files.

Another motivator is that there seems to be a lack of good solutions currently available. This is illustrated by the fact that it is not uncommon for the decryption-key and the cipher to be sent over the same channel when sharing encrypted files. This holds true even for security specialists! This signals a need for better implementations of the current technology as the possibility to create such programs definitely exists. One can safely assume that the currently available programs either lack in usability, functionality or a combination of the two. This thesis will clearly demonstrate that this does not have to be the case.

1.3 Existing solutions

The most important work related to the solution that will be suggested are PGP [5] and WebRTC [6] as well as DataChannels [7, 8]. The research and implementations stemming from these papers are vital for the choices and implementations suggested in this thesis. PGP and WebRTC are the groundwork upon which the system is built. The DataChannel functionality, as part of WebRTC, is also important to make the transfer of files easy and reliable.

There also exist solutions available and under development that resembles SendIt. FireFox Send is a quite recent release (Jan. 8th 2017), that also runs on Node.js. It aims to make sending files easier, in an encrypted fashion. They have chosen to use a cloud service to store the file. The link expires after 24 hours or the indicated number of downloads [9]. This is in contrast to SendIt's direct solution.

Tox is a solution that is fairly similar to SendIt. Both Tox and SendIt use a direct, serverless solution, which is a rare approach. Tox uses DHT to create a network layer for finding connections and another DHT network layer for connection setup between nodes. This results in a decentralized P2P network with end-to-end encryption. Unfortunately using DHT means it is harder to maintain and deploy, and that

one needs to enter the DHT network through certain nodes. This is called bootstrapping. SendIt can be deployed easier and has no need for users to enter the network through specific nodes. The trust system suggested in SendIt also differs from Tox's implementation. None of the problems mentioned are present in SendIt [10, 11].

There is also I2P-Bote, which is a solution for sending e-mails in an encrypted and secure fashion. Unfortunately, this solution has a few issues. Namely, it does not work together with already existing systems. Only people using this program can communicate with each other, however, there exists an add-on for Thunderbird that allows the two systems to work together. I2P-Bote is also hard to use, and even the instructions are difficult to understand for people without technical backgrounds [12, 13].

There is also an existing implementation that was used as a reference for SendIt. The Serverless-WebRTC solution [14] was what sparked the idea of making a system no longer depending on servers for sharing files. This solution is quite basic, but it is a good proof of concept.

Some general solutions should also be discussed. Regular e-mail attachments will be the first category called 'E-mail attachments'. The second category will be cloud-based solutions, SNS-based solutions, and similar solutions, which will be called 'Other solutions'. The reason for making this specific division into categories is because these solutions share almost all characteristics, with only minor differences in implementation.

1.3.1 E-mail attachments

The e-mail system is built on the SMTP protocol. It was built to facilitate transfer of cleartext messages. By also including MIME, support for media was also made possible. Neither of these protocols offer any encryption or security features, and lack support and consideration for basic security principles. The reason for this is that they are old protocols and it was part of the design of the e-mail system at the time [15].

There's also S/MIME which is the secure version of MIME. While this protocol does implement a lot of the security features lacking in MIME, it has a lot of practical issues. The main two being:

Certificates: The user has to obtain a certificate on their own. It needs to be signed by a CA that is recognized by the receiver. This usually means having to pay to get one issued. This is complex and should not be necessary to achieve basic privacy. It also lacks support in webmail clients, since they usually do not support S/MIME certificates.

Usability: The user is often required to manually check and verify digital signatures in S/MIME. There's also no way to know if the recipient can even verify such signatures, which in turn leads to confusion and not being able to retrieve the data.

There's also another reason for not using these protocols for transferring files. All files are required to be Base64-encoded. This usually adds approximately 37% to the original file size [16]. Most common mail providers and services limit file size to 25 megabytes, **after encoding**, which in reality means the limit is closer to 20 megabytes.

This limitation is also in effect in regards to incoming mail, which means that your mail may be dropped if it exceeds this size. One possible reason for doing this is to avoid DOS attacks, since sending large files may take up all the network capacity of the server. As such many e-mail clients have began storing files on cloud

services and instead attach a link to these files. This opens up a new issue, which is discussed in the next section [17].

1.3.2 Other solutions

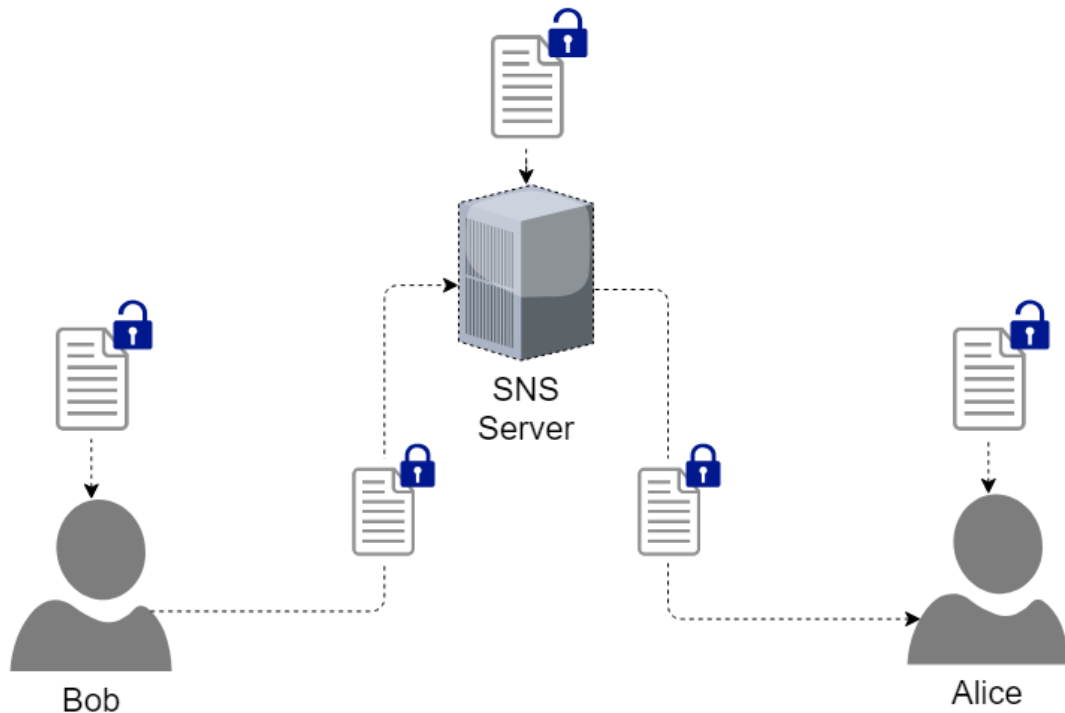


FIGURE 1.1: False end-to-end encryption used in some SNS. The service has access to the keys used to encrypt the data. This allows the service to decrypt the data you send.

The use of cloud services has exploded in recent times. The situation today is that if the file attachment is too large, it will often be uploaded to a cloud-service instead and a link to said file attached. While this sort of functionality is easy to use, it makes users lose control over their data. The user is giving a company access to the file that was intended only for the receiver. In reality, the user is giving away ownership of the data and longer has control over how long it is stored, how it is stored or who can access it. The user is at the mercy of the service provider. While it is doubtful that these service providers ignore attackers and disregard the privacy of their users, one would be very naive to assume that it cannot be stolen. There are countless examples of such services being successfully attacked or information being leaked on accident. A few examples are:

- **Celebgate/The Fappening:**
Attackers hacked celebrities' iCloud accounts and gained access to large amounts of private pictures, videos and other data [18].
- **Verizon leak:**
A misconfiguration in the cloud storage of telecommunications company Verizon, exposed the personal information of up to 14 million American customers [19].

- INSCOM leak:

A large amount of critical data belonging to the United States Army Intelligence and Security Command was leaked onto the public Internet [20].

There are also a plethora of SNS that support file transfers. Many of these services boast of being E2E-secure and protecting users privacy. Unfortunately, since there is no way for the users to know how each service handles their information, there is no way to know how secure their solutions really are. There is a precedent for being wary of these services though, as some have been revealed to only encrypt data between end nodes and the server, as shown in [Figure 1.1](#). One example of this is Skype [21, 22, 23]. While other services may not work the same way, the fact of the matter is that anything going through or being stored on their servers, is out of reach for you as an end user. Would it not be better if you could transfer the file directly to the intended receiver and be in full control over where the file is?

1.4 Contributions

The intended use case for SendIt as a system is when transferring sensitive files. For whatever reason one might have to keep data private and confidential, when transferring such data online, SendIt will be able to provide the necessary service.

One example may be for companies transferring personnel-files between their offices. A lot of companies outsource data storage and management. As such, sending personnel-files, which then get uploaded to a cloud storage, may in fact be illegal, since the company is no longer in full control of the data. This is an unnecessary risk as direct transfer is quicker, easier, and limits liability in these cases. Not only that, but it also removes the need for storing the file somewhere in transit, which removes the risk of it being stolen if an attacker gains access at a later date. We can conclude that in cases such as these, SendIt is an overall improvement compared to the current alternative. The arguments for this example can be used for any transfer of sensitive or private data, and as such is applicable to a variety of scenarios. Thus, the use cases of SendIt are many, but mainly focused around transferring confidential or private data.

Many solutions to share files already exists, with a few mentioned previously ([Section 1.3](#)). The exchange and management of identities combined with peer-to-peer transfer and trust evaluation is original to SendIt. The combination of these technologies into a singular system constitutes my original research activity.

This thesis' contributions can be summarized as follows:

- Serverless implementation of WebRTC in larger systems:
This thesis goes to show that serverless implementations of WebRTC can be used in larger systems. The thesis evaluates the advantages and disadvantages of such a solution in contrast to more commonly used solutions, both for WebRTC and other technologies. The advantages of not having a server broker all connections are many, and if IPv6 becomes more popular, an increase in such solutions are likely to occur due to the fact that NAT traversal will no longer be necessary. This in turn makes it easier to connect directly to endpoints. It would also allow for companies to provide services while lowering the cost of providing such services since they no longer need to provide servers to broker connections, greatly increasing profit margins. This would be very beneficial, especially for VOIP solutions, which is currently one of WebRTC's main usages.

- **Client-only development:**

A benefit following the serverless approach is that only a client application is necessary to use the service. No centralized program or dedicated servers are required for the service to stay operational. This reduces development time, resources needed, lowers attack surface and results in simpler implementations. Many programs exist only locally, but being able to communicate over the internet using such programs, is rare. Usually at least one connection to a centralized server or node is required for such programs to work. This is not the case for SendIt, and as such, opens up a new venue for application development.
- **Propose a new system that can be expanded as needed:**

A system offering endpoint authentication and direct P2P connections, while still being easy to use, is non-existent in today's technical environment. By introducing such a system, better security becomes available to the masses. It also functions as a platform which one can modify and build on to improve current functionality, or add new ones. It is easily expandable and can be used as a platform for any kind of application over the internet. It lays the groundwork so others can focus on the specifics of their solution, instead of worrying about connection setup or authentication of users.
- **Direct connections between users:**

As mentioned in the [Section 1.2](#), one of the goals of this thesis is to go back to the idea of a decentralized internet. A direct connection, when possible, is a big step in the right direction. By utilizing P2P this is easily achievable. The problem comes down to identification, authentication, and protecting the integrity of the data sent. This is rarely needed, however, as P2P is not commonly used in systems that require any kind of authentication. In such systems, tokens issued by a server can be used as authentication. By creating a system using P2P and avoiding the use of a central server, problems arise. These problems can be solved by combining P2P and public-key cryptography, but this again raises the issue of creating trust and trust management, since this is usually taken care of by the server. By using the web of trust model, this problem can be mitigated. This type of complete system for direct connections is new and original.
- **New perspective on e-mail attachments:**

It is about time this functionality gets a rework and improved implementation. It has been largely unchanged since it was implemented and is in desperate need of being replaced. This thesis suggests a new system that can easily be implemented in both online, as well as local e-mail clients. It can be used as either a supplementary system, or a replacement. To properly replace the current system, some updates and extended functionality would have to be added to the proposed system, but most of the work is already done. To clarify: this thesis is only one of many ways to improve the current system. A combination of solutions will probably yield the best results. What this thesis aims to do is be one of the better, if not the best, currently available solution for those who value usability and security.
- **Security to the people:**

From a layman's perspective, computer security as a discipline is difficult. The computer only became mainstream in the last ~25 years, and as such, society

has not yet completely adjusted to this new technology, especially the security aspect of it. While most of the workforce has adjusted and can now perform their tasks on a computer, they do not have a deep understanding of how computers work. Most users are not qualified to make judgments about how secure a solution is, or how it should be used. They rely on the IT departments to serve them easy to use applications that are customized for the intended purposes. These people occasionally need access to equally secure solutions as someone with a security background, but do not have the skills to use the same tools as the security professionals. As such, applications abstracting away all the technology and allowing for easy to use, secure systems are needed. This seems largely ignored by the security community today, as most applications are command-line utilities, or have extremely underdeveloped GUIs. SendIt tries to remedy this by being a system that allows users to focus on the task they want to accomplish, file transfer(s), while taking care of the security and technical challenges on behalf of the user.

- All of the previous points combined:
All items on this list may not be original, but combined they result in a unique approach to a problem that is largely overlooked. No other system has the combined properties outlined, which makes SendIt the first of its kind. Hopefully, this thesis will contribute to, and put focus on, the issues previously stated and lead the way for improvements and better solutions in the future.

1.5 Organization

The thesis is organized as follows:

- [Chapter 2](#) explains the technology and principles used in SendIt and gives a thorough review of the background for choosing them.
- [Chapter 3](#) goes through the concepts and design of the base system. It reviews the basis for the functionality of SendIt. It also discusses the design of the ACS server.
- [Chapter 4](#) examines the implementation of SendIt, based on the concepts and design discussed in the previous chapter. It shows how the application looks and functions, as well as how the ACS server is implemented. It will also give an overview of the extendability of the system.
- [Chapter 5](#) will discuss the design and implementation chosen for the two modes. It will show the use cases for each mode, discuss advantages and drawbacks and indicate how to best take advantage of the given functionality of each mode.
- [Chapter 6](#) will discuss the experiments done as part of the research. It will give an overview of the motivation for doing the experiments, the methods used and the impact and findings of the experiments.

- [Chapter 7](#) examines and evaluates SendIt as a system and compares it to existing systems. The analysis is focused on the security and usability of these systems.
- [Chapter 8](#) concludes the thesis by summarizing the contributions and work done. It also discusses future work and possible improvements.

Chapter 2

Technology and Basic Principles

This chapter will go over the technologies proposed and used in SendIt and give an introduction to their functionality, as well as the reasons for choosing said technology. It will give a basic understanding of the technology, which the rest of the thesis is built upon.

2.1 Cryptography

The encryption and authentication scheme used in SendIt is based on public-key cryptography and symmetric cryptography. Public-key cryptography is a technology that creates a public and a private key pair, where the public key is used for encryption. The encrypted data can *only be decrypted by the corresponding private key*. The inverse is also true. Anything encrypted with the private key, can be decrypted by the public key. The common usage is to spread the public key around, hence the name public. The private key (also called secret key) is kept safe, and only the identity that is the owner of the key pair should have access to it [24].

In contrast, symmetric cryptography utilizes the same key for encryption and decryption. The advantage of symmetric cryptography is that it can encrypt larger amounts of data. The disadvantage is that anyone with the key, can read any message. Symmetric keys usually also needs a value called IV (Initialization Vector), that is used to randomize the encrypted values. It allows for two identical messages to have different ciphers. There are two forms of IV schemes. One is using completely random number(s) each time, called a randomized scheme. The other scheme requires each IV to be unique, and is called a stateful scheme [25].

SendIt uses public-key cryptography to exchange encrypted messages that only the person having the correct key can decrypt. This is the same principles as used in Public Key Infrastructure. The difference lies in that PKI relies on a central authority for signing and verifying the keys used, in the form of certificates. SendIt tries to avoid any central form of governance, and as such, implements the web of trust model instead, in order to avoid this central management. This is discussed more in [Section 2.2](#).

2.1.1 Key management

This means that the keys have to be the same every time, which requires each user to store their keys for later. This should be done with some consideration, since a stolen private key means anyone can now take your identity and successfully authenticate as you. As such the need for secure storage arises. This can be solved by encrypting the keys with a password, then storing them on the local machine. Only when a key is in use, will it be read and decrypted. This decreases the chance of the key being

stolen, since it minimizes the attack surface by only having the key available when it is in use.

When storing keys it is necessary to keep more information than just the key. For example, it is also necessary to know who it belongs to. For the proposed system, information relevant to the trust-level of each key also needs to be stored. The easiest way to do this is to store the certificate of each key with the key. That means the trust in a key will have to be re-calculated every time it is loaded. This is costly when it comes to computational resources, but allows the system to re-calculate the trust in a key before each use. This gives a flexible and up-to-date environment where trust is evaluated, similar to in the real world.

2.1.2 Certificates

SendIt also proposes to use digital certificates like the ones used in the web of trust model [5]. Certificates traditionally contain various information. The certificate is then signed by any number of “introducers”, by encrypting it using their private key. An introducer is an identity who trusts another identity and vouches for the legitimacy of that identity [5].

Certificates used in PKI are required to contain enough information to provide a third party with the subject’s public key. Usually certificates used in PKI contain more data than required. The minimum required data can be summed up in four points [26]:

- CA identification information
- Subject identification information
- Subject public key
- Validity (time)

More information on introducers and certificates can be found in [Section 2.2](#).

2.1.3 Authentication

The last part of the puzzle is *how can a user be authenticated based on this information?* There are many different schemes, but SendIt focuses on the one suggested by PGP for securely exchanging messages [27]. The system is based on public key encryption. It also uses a session key, which is a random, one time generated symmetric key. The system guarantees the authenticity, confidentiality, and integrity of the data. Encryption and decryption using the PGP system is shown in [Figure 3.2](#) and [Figure 3.3](#).

The way PGP works is by first generating a symmetric key and encrypting the data to be transferred with this symmetric key. This results in a cipher. Let us call this *Cipher 1*. Then the symmetric key is encrypted with the recipients public key. This also results in a cipher (*Cipher 2*). The data transferred to the recipient is *Cipher 1* and *Cipher 2*.

The recipient then retrieves the symmetric key, by decrypting *Cipher 2*, using their private key. Then, using the retrieved symmetric key, *Cipher 1* is decrypted and the original data is available.

This solution is made with a one-directional exchange of data in mind. This system will be used to authenticate both endpoints, which means it will have bi-directional communication. The way to solve this issue, is by taking advantage of

the way symmetric keys encrypt data. The initial exchange from Sender to Receiver gives proof to the Sender that the recipient is as intended. This is not the case if the same IV is re-used for the reply. As such, it should be changed.

Usually the IV is shared in cleartext, since knowledge of the key is also required to decrypt it, therefore it is not sensitive data. In SendIt, the IV used in the reply is encrypted with the Sender's public key. That means that in order to be able to access the data, the private key is needed, and as such, the Sender is also authenticated. This solves the issues of mutual, bi-directional authentication.

2.2 Web of trust

This section is a suggestion and evaluation of the system only. This has not been implemented in SendIt due to time constraints, but is considered a part of the design.

2.2.1 Overview

The model suggested is based on the web of trust. The web of trust model says that trust should be split into two groups: **Trustworthiness of public-key certificate** - *How willing are you to trust this key pair when it is being used to communicate?* and **Trustworthiness of an introducer** - *How willing are you to trust this key pair when it is being used to introduce another key?*

It also suggests that the trust level given to a key should be based on points given to a certificate, based on the sum of introducers and their respective trust level. There is no concrete policy or algorithm suggested for evaluating which level of trust is assigned. The model does define levels of trust, but they are different for the two groups previously mentioned.

There is also no recommendation of which trust level should be considered safe or unsafe [5]. This is left up to each individual user, since there exist many ways to rate and evaluate trust. For this system, a simple point-based system is likely to be the best fit. The web of trust model has also been recommended by other researchers, albeit for a more traditional WebRTC authentication system [28]. For a look into more options, Jøsang et al. (2007) [29] gives a good overview over possible alternatives.

2.2.2 Certificates

The way one infers, evaluates, and determines trust is by using certificates. The trust is evaluated based on interactions with others and the interactions between trusted introducers and others. This means that information from identities one does not trust will be disregarded (their introductions mean nothing), while still valuing and receiving updated information from identities one does trust. In other words, many untrusted identities introducing someone does not hold any value, while introductions from a few trusted identities has a significant impact on the trust level assigned to a certificate.

2.2.3 Certificate sharing

The exchange of keys and the trust each identity assigns these keys are central to the web of trust model, as described previously (Section 2.2). Since the connection used for communication is already secure, and each party authenticated, this data can be transferred without additional encryption. This section will describe how this is

intended to be implemented in SendIt. This is included as part of this chapter since it has not been implemented in the prototype.

Request certificates:

SendIt will have an automated, bi-directional sharing of certificates, that happens on every exchange. This sharing will be part of the file transfer protocol and will happen after the file transfer has completed (before terminating the connection). This automatic sharing will abstract the system away from the user, to make the system easy to use and understand, while still updating the trust-level for different identities according to recent changes in the system. The choice to make this part mandatory also allows for the system to avoid being biased by only receiving updates from certain identities.

Evaluate certificates:

While exchanging certificates with all communication-partners is important to avoid bias, it is also important to let the user influence and differentiate the level of trust given to each partner. This is where settings and evaluation comes into play. Each user can change the level of trust in each partner manually, based on his own preferences.

As an extension of the trust given to each partner, the information they share is also evaluated based on how trusted the partner is. The trust evaluation of each key, and by extension each identity, will vary depending on user settings and preferences. It will however always follow the scheme as described above. The effect of this evaluation will result in a corresponding trust-level associated with the identity. Identities that have too low a trust level to be used can either raise an *untrusted user warning* before communication occurs, or a *connection terminated message*, depending on the user's settings. These keys will still be stored in the user's key-chain and stay available for future evaluation, even if their current level of trust is low.

2.2.4 Sybil attack

A common attack in decentralized reputation systems is the Sybil attack. It is an attack that takes advantage of the fact that the system can be manipulated by having a large number of identities share the same, false information, to innocent identities. In SendIt's case, that a certain key pair belongs to a certain identity, while in fact it does not. Through sheer numbers, the attack manipulates identities in the system to believe that the attacker should be the trusted identity, not the original identity. This misleads users to connect to the wrong identity, which makes them vulnerable to attacks. It can also lead to being unable to connect to the legitimate identity, since all traffic for that identity is instead routed to the attacker [30].

2.3 WebRTC

WebRTC is a fairly new technology developed for real-time communication between web entities. It focuses on allowing video, audio, and data transfers over P2P, through an easy to use API. It does this without the need to install plugins or having to download native applications, since it is already implemented in most of today's popular browsers.

2.3.1 Connection setup

To set up a connection using WebRTC, there are different variables one has to take into account. First, *what are the network conditions for each endpoint?* To be able to create a connection, a way to address each endpoint directly is needed. To do this WebRTC utilizes STUN [31], and/or TURN [32], and ICE [33].

ICE, STUN, and TURN

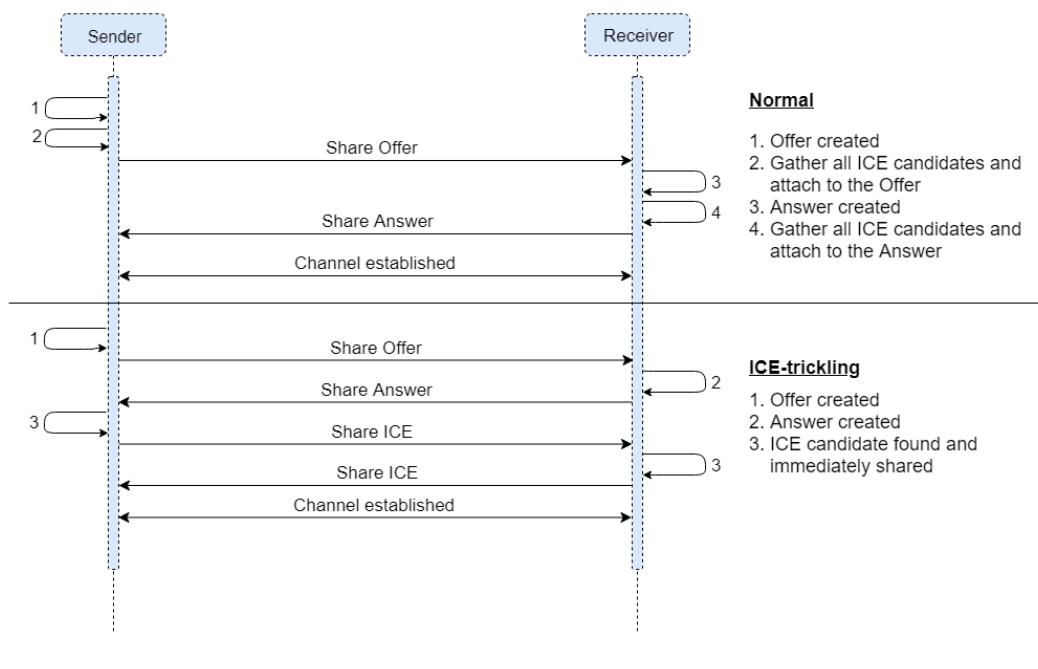


FIGURE 2.1: This illustration displays how ICE trickling works. Do note that for ICE trickling, process number 3 in the illustration can happen at any time, and any number of times.

The ICE protocol works by trying to connect directly to the endpoint. If an endpoint is behind symmetrical NAT, however, a TURN server is required. A TURN server acts as a relay-server that both endpoints connect to, and there is no longer a direct P2P connection. Instead each endpoint is connected to a server by a P2P connection, and the server forwards information to the other end. It is the ICE protocol that detects and manages this information, and tries to find the best possible connection [34].

ICE is a protocol for NAT traversal used in Offer/Answer protocols. In regular WebRTC-applications, something called ICE trickling is used. This is because generating the Offer and the Answer is quick, but gathering ICE candidates takes time. When using ICE trickling, the Offer/Answer is generated and shared independently of the ICE candidates. Once the Offer/Answer is generated, ICE candidate gathering begins. Once an ICE candidate is found, it is immediately shared. This allows the connection setup to happen first, then ICE candidates to be shared afterwards [35]. This is illustrated in Figure 2.1. In SendIt, this functionality is utilized in the ACS mode.

In Serverless mode, all the information is gathered, then attached to the Offer/Answer, and then shared. ICE is used in order to be able to address and reach computers behind NAT [33]. The easiest way to describe the whole process is to

imagine ICE as the decision maker, utilizing STUN and TURN to find information about the endpoint. Once all the information from STUN and TURN is gathered, ICE prioritizes the information, and shares it as separate possible ways to connect to the endpoint. These connection alternatives are called ICE candidates.

SDP and DTLS-SCTP

Session Description Protocol (SDP) is a format for session descriptions. This is a protocol for formatting data, for example discovered through the ICE protocol, to be shared with the other endpoint [36]. For the Serverless mode, it is up to the users how they share the Offer and Answer, and as such it does not use a transfer-protocol. In the ACS mode, secure WebSockets are used for sharing the information.

The WebRTC DataChannel used for direct communication uses DTLS-SCTP as it's transport-protocol [7].

“DTLS itself is modelled upon the stream-orientated TLS, a protocol which offers full encryption with asymmetric cryptography methods, data authentication, and message authentication [37].”

In other words, the communication is securely transported between the endpoints, once the connection setup is done.

Encapsulation of SCTP over DTLS, as well as the ICE and SDP protocols are complex. As such, an in-depth explanation is available in Holmberg, H.-C. (2015)[8], for those unfamiliar with this technology. In summary: each endpoint gathers the necessary connection data and then formats it according to the SDP protocol, for session-negotiation and initiation. Then the DataChannel is created and the connection is made.

Offer and Answer

Following, is how the relevant RFC describes how endpoint guarantee is done:

“A certificate fingerprint is a secure one-way hash of the DER (distinguished encoding rules) form of the certificate. If the X.509 certificate presented for the TLS connection matches the fingerprint presented in the SDP, the endpoint can be confident that the author of the SDP is indeed the initiator of the connection [38].”

This explanation can be a little hard to understand, so let us break it down. Both the Offer and the Answer generated by the WebRTC API has a fingerprint of a certificate attached. The endpoint generates a X.509 certificate and a corresponding private key. By matching the fingerprint received during signaling with the certificate used during the setup of the DTLS-SCTP connection, one can be certain that the endpoint is the same. This is illustrated in [Figure 2.2](#).

2.3.2 Signaling server

In the usual use case of WebRTC, a signaling server is used to share SDP Offers and Answers, as well as continuously sharing ICE candidates. This separate channel allows for re-negotiation of communication, if the connection breaks down. An Offer is the exchange of SDP information from the sender, to the receiver. An Answer is the exchange of SDP information from the receiver to the sender.

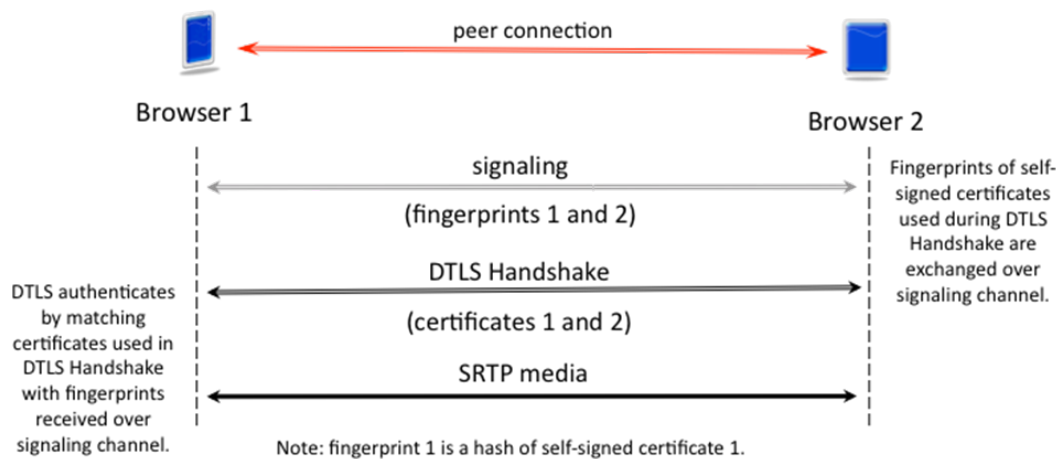


FIGURE 2.2: This illustration shows and describes how DTLS is setup between two endpoints. The final channel in the illustration is of the type SRTP, but the same procedure is true for SCTP [39]

There are several reasons for the usage of a signaling server. One is the ability to reconnect in the event of a failure in the established connection. Another is that it delays the Offer and Answer from being created until both endpoints are online, after which it is immediately created and shared. This is the case when using ACS. When using Serverless mode, however, this is not the case. This means it is affected by the fact that network conditions can change rapidly and, as such, ICE candidates may no longer be viable, leaving endpoints with no way to connect or renegotiate the connection. For more information on the lifetime of the exchange, see [Chapter 5](#).

Another use of the signaling server is that it allows for encrypted communication between the parties, via the server. As long as you trust the signaling server, your communication is confidential. This is important in regards to the exchange of the Offer and Answer, as they contain information that allows a secure connection to be set up, as described in the previous section.

Serverless mode does not have a secure channel over which the Offer and Answer is exchanged. That means it would be vulnerable to a MITM-attack. To combat this, the Offer and Answer will be encrypted by each endpoints public key for all connections after the first (See [Figure 3.2](#) for illustration). With this protection in place, an attacker can not get a hold of the Offer and/or Answer without stealing the private key that can decrypt the cipher.

2.4 WebSockets

To understand why WebSockets are necessary, let us first examine how normal communication between clients and a server is done. The client sends a request and the server sends a response. If the data contained in the response is time critical, it will, in many cases, already be outdated by the time it is rendered by the client. A manual way to combat this is to refresh the page. A more elegant way is the use of polling. Polling is a regularly timed, synchronous call the client makes to the server, to look for new information. This works well if you can predict when new data will be available. The problem is that this is often not the case. There are other alternatives as well, for example, long polling or streaming. However, they all come with

certain issues regarding real-time, two-way communication. Especially latency and overhead are problematic.

This is where WebSockets come into play. WebSockets offers real-time, duplex, bi-directional connections. It makes communication between a client and a server easier and faster. It also supports real-time communication, which is an added bonus. WebSockets are commonly used in a lot of real-time applications in today's online environment. From transferring game data to simple chat applications. It is also commonly used for signaling in WebRTC applications. This is also how it is used in SendIt, specifically in the ACS mode.

Another great thing about WebSockets, is that they support the traditional use of TLS. This means that the WebSocket protocol can use the same security mechanisms as traditional HTTPS traffic. This makes it easy to protect the confidentiality, integrity, and availability of network communications. Finally, all modern browsers have native support for the WebSocket protocol and as such it is incredibly easy to deploy [40]. For these reasons, the WebSocket protocol was chosen to take care of the communication between endpoints and the server, required by the ACS mode.

Chapter 3

Concept and Design of SendIt

This chapter will discuss the overall concepts and design of SendIt. It will illustrate how SendIt utilizes the solutions discussed in [Chapter 2](#) and how this is beneficial to the system as a whole. In addition, general information relevant to the application and system will be discussed here. It will also discuss the design of the ACS server.

3.1 Concepts

The following concepts and assumptions build the base of how the technology is applied, while also being a product of the technological limitations and how this technology is implemented. The concepts introduced are a mix of new concepts and ideas and concepts borrowed from previous research. The assumptions are a result, and consequence, of the initial idea of the system, as well as the technological limitations imposed by the chosen technologies, combined with the lack of better options. These assumptions were necessary to narrow down the work and to be able to keep the system simple for the end users. While these assumptions may limit or reduce some of the security measures available, the limitations they would impose on the usability were considered more important.

3.1.1 First trust

The first trust, or initial trust, is the basis of every interaction in every computer system, as well as in the real world. In the real world, you can see the other person, and as such, easily identify who it is. This is not the case in computer systems, therefore, a way to identify entities is necessary. In this system, the choice was made to use e-mail addresses as a basis for uniquely identifying each entity. Since one of the main goals is to improve e-mail attachments, it felt like a natural choice, as well as being easy to implement. Since there is now a way to address each entity, the necessity for confirming that identity arises. This process is called authentication, and how it is done will be discussed more in-depth later.

To be able to authenticate an identity, the fact that the identity has to possess something unique has to be addressed. If not, someone else can impersonate them. In the real world, everyone has a unique (or close to unique) appearance, which is almost impossible to fake. This is not the case for computers. Usually the PKI model (explained in [Section 2.1](#)), is used to vouch for the validity of an identity in computer systems. As explained previously, this requires a lot of setup and cannot be done easily. Because of this, it is not fit to be used in combination with SendIt.

SendIt is built on the idea of trusting the first interaction, based on non-absolute authentication methods. This means the first Answer and Offer exchange (connection setup) is done assuming the other endpoint is not malicious. It is done in an

unencrypted manner, without any guarantee of the integrity of the message or using any authentication mechanisms. The reason for choosing this solution is that it allows for an easy and convenient way to start communicating with new partners. It is important that the system is kept simple, to keep it user friendly.

It also allows for an intuitive approach to the first interaction. Instead of having someone's pre-shared secret or other means of authentication, the timing and files shared can be a means of authentication. It feels unnatural to have someone randomly send files to you, unless it is somewhat expected. It is also strange to receive files where the content seems unknown or strange, without any previous communication or knowledge. As such, the argument can be made that this system is more intuitive than the alternatives.

While the timing and files shared do continue to count towards the trustworthiness of an identity, it is important to note that *only the first interaction is inherently trusted*. All subsequent communication is end-to-end encrypted and the endpoint is *always authenticated* before communication starts. As such, if the user takes care to be certain of the endpoints identity for the first interaction, all subsequent interactions are guaranteed to be with the same identity, barring that user having their credentials stolen.

The take away from this is, while the timing and files shared should still be considered, all subsequent interactions are, by and large, guaranteed to be with the same identity. As such users are encouraged to take extra care during the initial setup, but can relax and trust the endpoint during all future interactions and rely on the system to warn them if something is amiss.

To sum it up by using a comparative example: If someone were to place a random package outside your door at a random time, with no information on it, one would naturally be wary of the contents and motives of that person. In contrast, if a friend made an appointment to deliver a package that day, one would be much less suspicious, even if it was identical to the package described in the previous scenario. There is no guarantee that the package is from your friend, but it is likely.

This is the basis of the first trust in SendIt, and allows for an unauthenticated interaction to be the basis of the trust building. The added benefit is that unlike in the real world, SendIt can guarantee that the package is unaltered and from your friend for *every interaction after the first one*, no matter when it is delivered and what the package may look like.

There is no denying that this decision leaves the system and end user open to be attacked during this first interaction. The advantage is that it keeps the system easy to use, and allows for a relatively secure system, which relies on human and intuitive factors for establishing the first trust. It relies on machine made, absolute authentication from that point on.

3.1.2 Trust building

Trust building in the suggested system, is based on several factors. Before getting into the specifics, it is necessary to point out that the trust system is only a theoretical approach and has not been implemented in the prototype. The proposed system uses trust transitivity [41] (*Trusting in Alice and Alice trusting Bob, means that we trust Bob more, than if Alice was not involved*).

Other means of building trust includes frequency of communication and bi-directional communication (Acting as both sender and receiver with the same identities involved). Trust built by such activities will gradually accumulate, in contrast to transitive trust. Transitive trust is just a way of assigning an identity a trust level

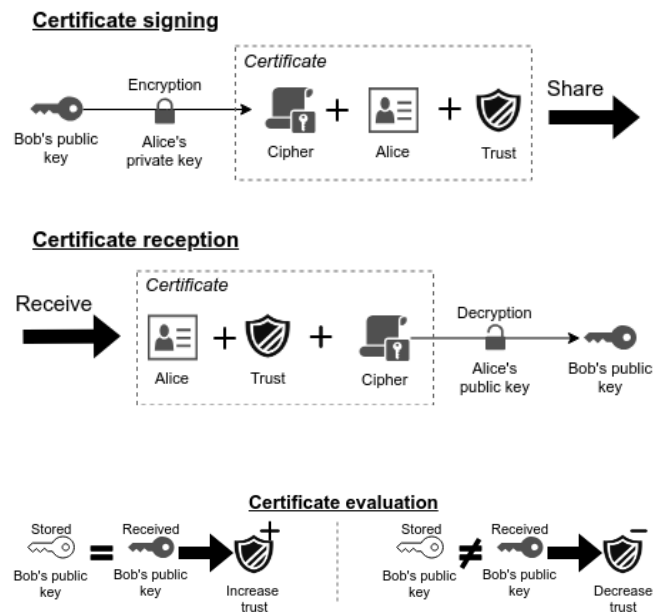


FIGURE 3.1: Certificate operations example where Alice signs Bob's certificate.

based on the introducers of that identity, and their respective level of trust in said identity. (*How many trusted identities signed, how much do we trust those identities and how much do they trust the certificate?*)

Trust can also be reduced. One event that can cause this is if a user tries to communicate with a previous partner, but that partner is not authenticated according to the key associated with that partner. SendIt's trust system works in a way that gives more weight to reduction in trust. This means it will be harder to gain trust, than to lose it. This is because it is generally a safer approach to be wary, than to assume good faith. In other words, a negative incident will reduce the trust significantly more than a positive incident will increase it. This will help maintain a balanced system that detects untrusted behavior rapidly and acts accordingly.

Trust is shared by using certificates. Unlike regular certificates (see [Section 2.1.2](#)), SendIt proposes a minimal and simple solution. It proposes that a certificate is just a list of identities and the corresponding cipher. The cipher is made by encrypting the public key of the identity being vouched for, with the users own private key. This way all that is needed to verify the authenticity of a key, is to find the introducer's public key and decrypt the cipher. As such, certificates will contain these fields: Introducer's name, Cipher, and Trust. The cipher is as just described. The trust field is a value of either 1 or 2, where 1 indicates full trust and 2 indicates partial trust. The trust value is included so it is possible to assess how much trust the introducer has in the introduced key.

When it comes to transitive trust, the specific values and model for assigning trust is not specified, but the idea is to assign different levels of trust, based on the amount of introducers from different trust levels. The level of trust a user has in the introducer of a certificate, directly affects the trust level given to the certificate.

Once an endpoint has received the certificates, they will be evaluated based on the signatures attached and potentially existing trust values. When the value has been calculated and the new trust level assigned, the key is available like any other

and will be used in communications with that identity. See [Figure 3.1](#) for an example. Do note that one certificate can hold many signatures.

The practical evaluation scheme for the system is not implemented in SendIt, but the initial idea was to assign these trust-levels based on a point system, where an identity accumulates points by getting verified by introducers that the user trusts. An option is to have different levels of trust, where an identity is assigned a trust level according to the amount of points it has accumulated.

The amount of points granted for each introducer depends on the trust level the user assigns to the introducer. It also depends on how much this introducer trusts in the key. This is so the nuance of transitive trust is clearer, since the system is based on these nuances of trust. SendIt should have an easy and intuitive trust evaluation, while also having a simple way to fairly assert the trust level of an identity. Unlike the web of trust model, different levels of trust for communication and for introducing keys should not be applied. There should only be one trust level assigned to each identity and it should apply to both uses equally.

The practical impact of the different trust levels should be visible to the user as a warning, if the receiver does not have a trust level that is sufficient according to the users settings. The user should be able to choose if he wants to ignore the warning or cancel the transfer. If the receiver meets the requirements, no notifications or warnings should appear.

SendIt should come with pre-configured values so novice users can rely on the standard settings. If they want to change these settings, they should have the option to choose to do so. The users should have an overview of all identities and should also have an overview of the respective trust-levels available in the settings menu, and the option to manually change these levels. This allows the users to override the system in cases where they deem themselves better equipped to evaluate the trust-level of the identity.

As noted in [Section 2.2.4](#), the Sybil attack is a problem in decentralized, reputation based networks. The proposed system will not be completely immune to such attacks, but instead look towards making it hard to propagate and become regarded as trustworthy by the majority of identities. This will be achieved by requiring interactions with other identities, as well as a signed certificate from trusted introducers, to gain a high level of trust. By requiring these, in combination, an attacker can not simply create a large sub-network of controlled identities that all trust each other, and then spread it to the main network with the same level of trust.

This allows for high resistance against this attack, since the attack exploits the fact that a large number of untrusted identities can manipulate how the main network regards the attacker. Since none of these identities will have a high amount of trust in the main network, they will not be regarded as safe. This is true even if they have a large amount of introducers, because the introducers are untrusted identities as well.

3.1.3 Authentication

The authentication scheme in SendIt is based on public-key cryptography. Each identity, or e-mail address if you will, is associated with a unique public and private key. These key pairs are used for authentication. In practice, this is done by encrypting the Offer and the Answer used to establish the WebRTC connection with the other endpoint's public key. This way the ability to connect to each other also acts as authentication, since only the identity with the correct private key can access

the information. This explanation is a slight simplification and will be discussed in detail in [Section 3.2.1](#).

This is in contrast to usual solutions, where authentication is done before initiating a connection. SendIt's solution removes the extra step, and combines these two processes into one. The reason this is made possible is because of how WebRTC's connection setup is done. Because of the authentication built into the Offer and Answer exchange, the system guarantees authentication of both endpoints. For more details on WebRTC's built in authentication of endpoints, see [Section 2.3.1](#).

As recently noted, the authentication in SendIt relies on key pairs. The key associated with an identity is shared during the first interaction. While there is no authentication of the respective identities during this interaction, the keys are shared over the P2P channel. This channel is end-to-end encrypted, as to not be vulnerable to attacks that manipulate the data in transit. In summary, as long as you connect to the correct endpoint in your first interaction, there is no reasonable way an attacker can impersonate that endpoint at a later time.

It is also important to note that while the proposed system suggests using e-mail addresses as identifiers, there is no direct connection between the identities in the system and the actual e-mail system. By using e-mail addresses, it makes it easier to add extra trust, by implementing an e-mail verification step if desired. The consequence of this initial separation does mean that the e-mail system has no direct influence or inherent effect on SendIt, but their close relation makes it easy to link the two. This gives all the benefit of the e-mail system, while not having to consider the risks or threats that are inherent to that system. It is left up to the developers and users of SendIt to decide if they believe such a link is a positive or negative addition, and as such their choice to make.

3.2 System design

The following section will address the design of the general solutions used in SendIt. It will explore the functionality that is used for both modes, and explain how they work and why they are necessary. It will also build the basis for understanding how the two modes operate, and which functions they build upon in order to work.

3.2.1 Cryptography

When a new key pair is created it is not stored on the disk until the public key has been successfully exchanged with another identity, in order to avoid storing unnecessary keys. The file where the keys are stored should be password protected, or otherwise encrypted, to ensure that if it is stolen, it takes considerable effort to steal the keys.

SendIt also utilizes symmetric keys for encrypting data. These symmetric keys are generated for every session, and used for one session only. The reason symmetric keys are necessary, is because the amount of data that can be encrypted with public-key cryptography is fairly small, and as such, a solution which can encrypt more data is needed.

The biggest difference between the symmetric keys and the asymmetric key pairs in SendIt, is that the symmetric keys are not stored across multiple sessions. The asymmetric keys however, are stored and stay the same. This is because they are used for authenticating the identities. The way this is done is in a similar fashion

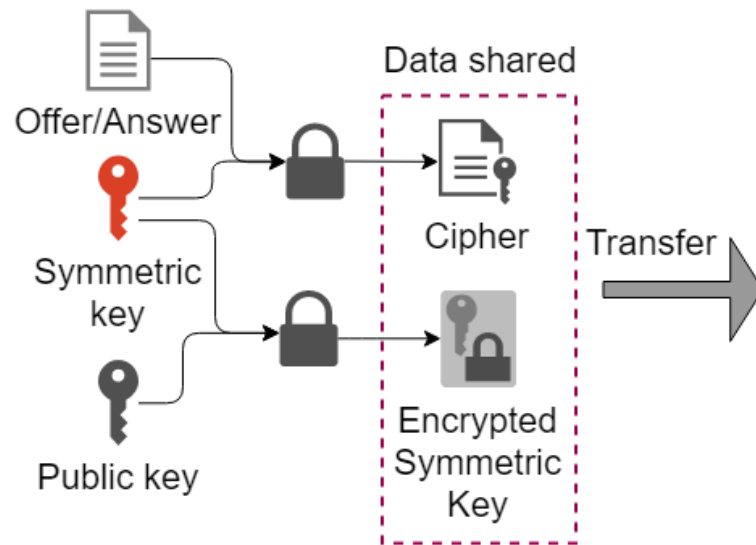


FIGURE 3.2: In this figure it is shown how the Offer/Answer is encrypted with a symmetric key, creating a cipher. This symmetric key is then encrypted with the recipients public key. The cipher and the encrypted key are then shared.

to standard PGP encryption and decryption, as explained in [Section 2.1.3](#). See [Figure 3.2](#) and [Figure 3.3](#) for illustrations on, respectively, encryption and decryption.

The plaintext in this case is the Offer or the Answer. If it is encrypted, it also contains the encrypted symmetric key, as well as the initialization vector for that key. The initialization vector consists of random integers. The IV is needed to correctly decrypt the data.

The difference between SendIt and PGP is that in order to authenticate the Sender, the Recipient needs to encrypt the IV with the Sender's public key and transfer this. If a connection is made, it means both endpoints had to have the correct keys. This is because of the way the connection setup works, as described in [Section 2.3.1](#).

3.2.2 Protocol

The base protocol used in SendIt is based on the one used in PubShare, an online WebRTC P2P file transfer solution [42]. This protocol has been extended and altered to fit with SendIt's needs. This protocol takes care of communicating the status of the file transfer, chunking the data, sharing file metadata, and verification of transfer completion. The protocol is also in charge of setting up authentication of the endpoints, since the data to authenticate each other should be exchanged directly. This protocol is very limited, and as such, unlikely to have a large overhead and impact on performance, however, this is only an assumption. Following will be a description of the different functionalities of the protocol.

Authentication setup

The authentication setup functionality consists of an Authentication setup packet and an Authentication setup reply packet. Both of these packets contain the identity and their associated public key. If one of these packets are received, the system checks that the information supplied does not conflict with existing data. If it does

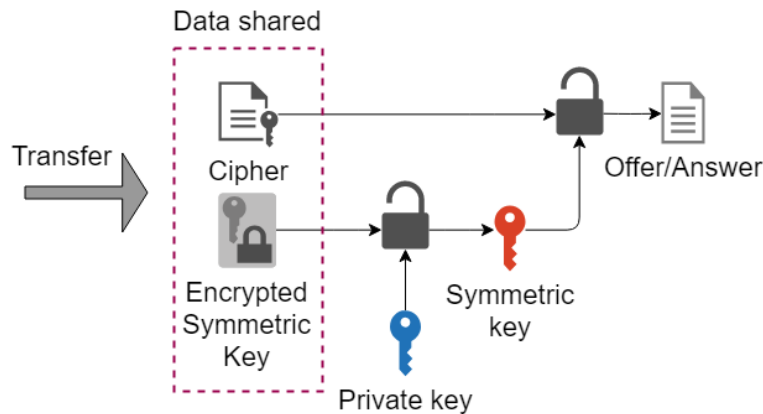


FIGURE 3.3: Here one can see the received data and how it is decrypted back into the original Offer/Answer. Only the identity with the correct private key can access the original data.

not, and the packet was of the type Authentication setup, the endpoint creates an Authentication setup reply packet and sends it back. If it was of the type Authentication setup reply, the transfer is initiated.

Offer and Answer

The Offer packet contains metadata about the files to be sent, so the recipient knows how many files will be transferred, what type of files they are, and how big they are. The Answer packet is just a confirmation that this data was received and that the recipient is getting ready for the transfer.

Request and Data

Once the recipient is ready to receive the first data, a Request packet is sent indicating which chunks it wants to receive. It is also used to confirm which packets it has previously received, and to request more data once all the previous chunks have been received. The Data packet contains information about which chunk is being sent, and the file-data for that chunk - the part of the file currently being transferred.

Following is an example to illustrate how it works: The recipient sends a Request packet requesting chunks 10 through 20. This also confirms all chunks up until 10 is complete. The Sender then transfers these chunks to the Recipient. Once all chunks are received, it requests more chunks until all chunks have been received. If a chunk is not received, the chunk before is treated as the last chunk received, and the chunks afterwards are re-transmitted.

Done

The Done packet contains no data field, and is a confirmation that the Recipient has received the file currently being transferred. If there are no more files to transfer, this indicates the end of the connection. If more files are waiting, transfer of the next file will begin once this packet is received.

Cancel and Error

The Cancel packet is sent if the user for some reason decides to cancel the current transfer. This interrupts the transfer and closes the connection. If the Error packet is sent, it means an error occurred and the transfer is stopped and the connection closed.

Request metadata

The Request metadata packet is only used in the ACS mode. It is necessary so the Recipient can request data about which files are being offered by the Sender, before accepting the connection.

3.2.3 File transfer functionality

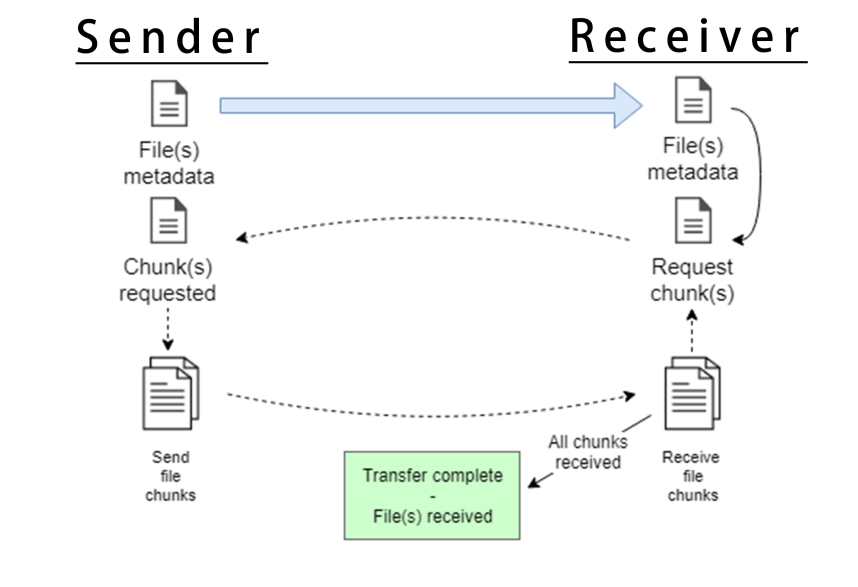


FIGURE 3.4: This illustrates the process of transferring file(s).

The file is always transferred over a secure P2P channel and with end-to-end encryption. It is transferred using the protocol mentioned in the previous section, in order to manage the transfer.

Implementing support for bigger files, which means additional chunking, should be relatively easy. As a result, transport of data larger than the current limitations, was not implemented. If this becomes necessary in the future, it can be added as an extra feature. For more on this subject, see [Section 4.3.3](#)

The chunking, as well as managing which files are to be sent and which files have been received, is done by the File Manager. The File Manager reads a file into memory, then separates the data into chunks. These chunks are then sent through the P2P channel. Once the recipient has received all the chunks, indicated by the metadata previously received, the transfer of that file is regarded as complete. This triggers the chunks to be combined into the original file, and then stored on the recipients computer. It also notifies the sender to start transfer of the next file, or that the transfer is complete.

3.2.4 Modes

SendIt operates with two unique modes. One is the ACS mode, which acts as a helper in establishing the P2P connection. The other mode is the Serverless mode, which leaves it in the hands of the end user. What is important to note is that these modes can be selected as the user sees fit. It will also retain information used and gathered in one mode, and utilize it for future connections in the other mode. What it cannot do is transfer data from one endpoint in one mode, to another endpoint in a different mode. Both endpoints have to be in the same mode in order to be able to establish a connection. However, they can switch modes and still connect at a later time, as long as they both utilize the same mode. These modes will be further discussed in [Chapter 5](#).

3.3 ACS server design

The ACS server is in charge of forwarding information from one endpoint to another. It also has some other functionality, like authenticating identities, for the sake of forwarding information to the correct endpoint. The functionality of this server is kept minimal and, as such, should be easy to utilize. It is also easily extendable and deployable, which makes it easy for users to host their own, for their own needs, if they for any reason do not trust any available service.

3.3.1 Authentication

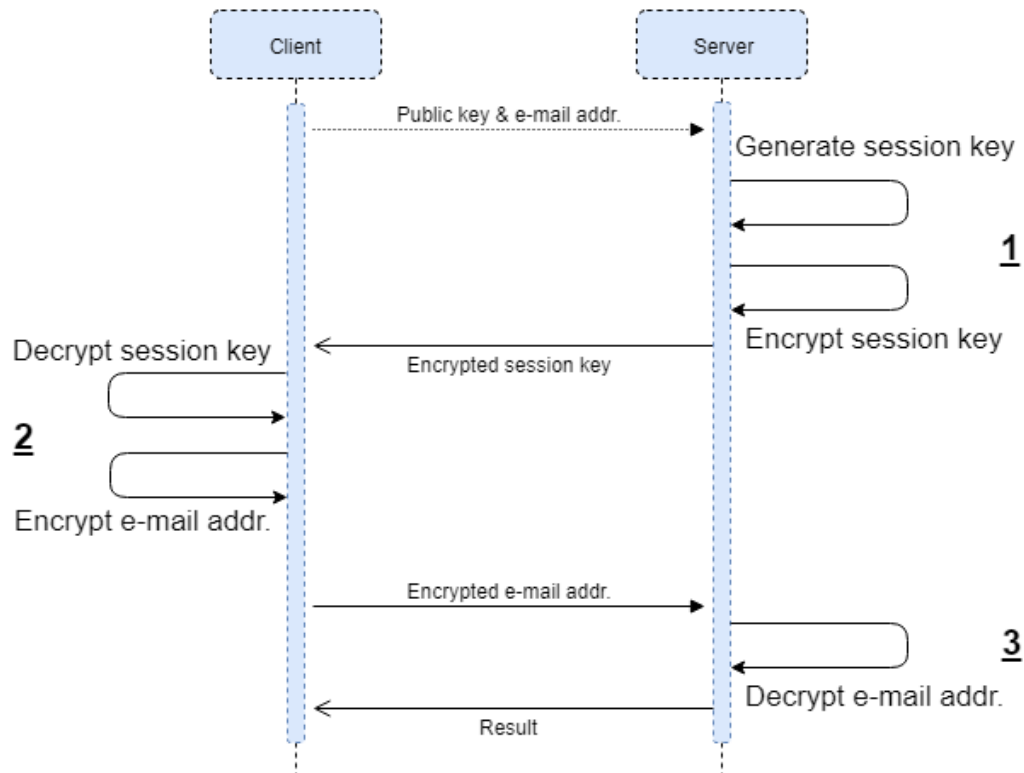
The authentication scheme for the ACS server is extremely simple and just meant as an example for more sophisticated solutions. It works by having the ACS server create a symmetric key and encrypt it using the endpoint's public key. To prove to the server that the endpoint could decrypt the session key, it encrypts its own identity (e-mail) using the symmetric key and sends it back. If the identity matches the one stored, then the endpoint is authenticated.

If it is the first time an endpoint connects to the server, the endpoint shares their public key and identity with the server. The server then generates a symmetric key, encrypts it with the public key supplied, and shares it with the endpoint. The authentication scheme is illustrated in [Figure 3.5](#).

The reason for incorporating this into the server is that forwarding data to the wrong endpoint could potentially reveal information not meant for that identity. As such, some basic form of authentication should be added to ensure that the information ends up at the right identity. This authentication does not replace or interfere with the regular authentication between endpoints in any way. It is simply a feature added to make sure the data arrives at the intended endpoint.

3.3.2 Communication

No code needs to be sent from the ACS server to the endpoint in order to communicate, since all the code is pre-programmed into the client software. As such, the server cannot manipulate nor control the clients behavior in any way, except that intended by the protocol. In this section, when WebRTC is specified, it can be substituted with any Offer/Answer based model or protocol.



1. Generate session key and encrypt with client's public key
2. Decrypt session key using own private key, then use session key to encrypt e-mail address.
3. Decrypt e-mail address and compare it against the one stored for this connection.

FIGURE 3.5: This figure illustrates how the ACS server authenticates clients. The initial transfer from client to server only happens if they are communicating for the first time.

3.3.3 ACS Protocol

The protocol used to negotiate and broker a connection in the ACS mode is made specifically for use with Offer/Answer based protocols. It is kept as minimal as possible, with some of the functionality not yet implemented. This functionality is not necessary, but would improve the system. The basic format of every interaction is shown in [Table 3.1](#).

For an overview of the general communication flow, once authenticated and properly connected, see [Figure 3.6](#). This is where the main functionality of the ACS server is shown. For information about the specific implementation of this protocol see [Section 4.2.2](#).

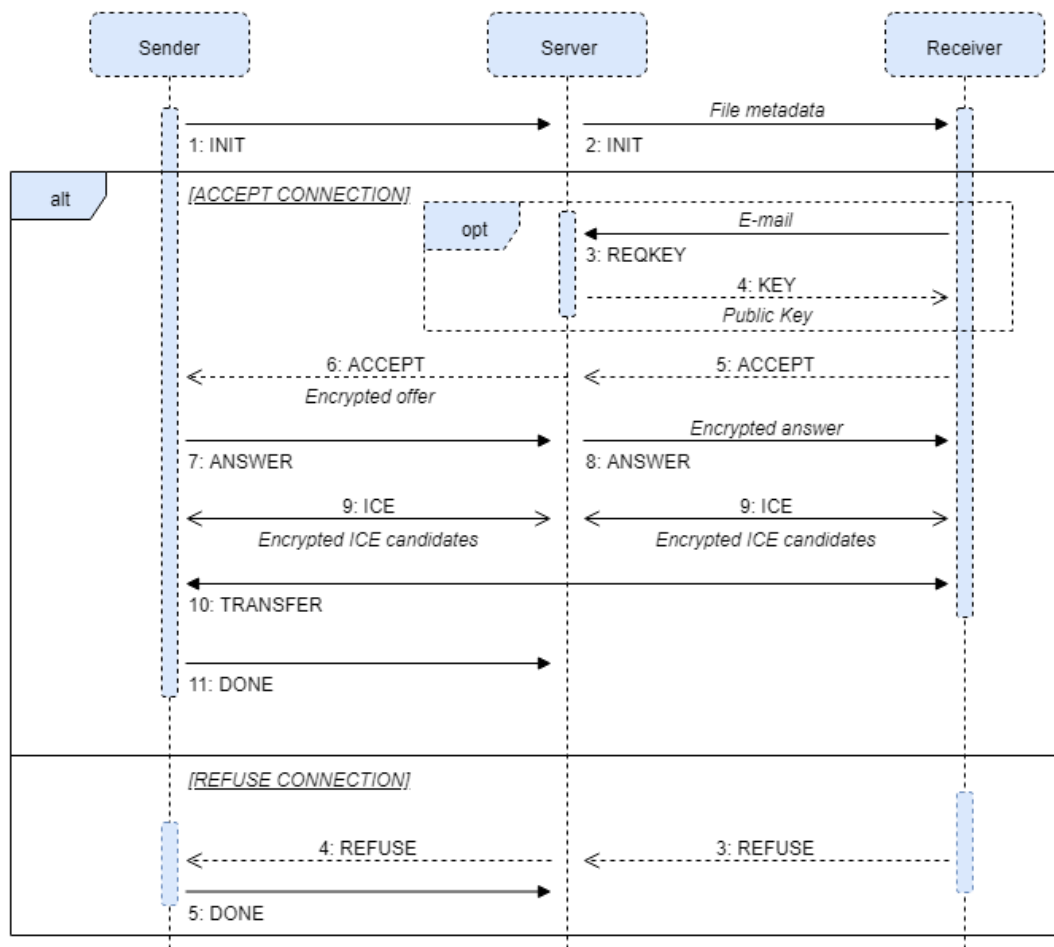


FIGURE 3.6: This is the main functionality and flow when using ACS mode to transfer files. Do note that the area marked OPT is not implemented.

Lookup

The first interaction between the client and the server is the *lookup* packet. The client connects to the server, and sends a *lookup* packet containing the identity's e-mail address. If the server has previously communicated with the identity, it will create and send a session key (symmetric key) encrypted using the key associated with the identity. If it has not previously communicated with the identity, the server's public key is sent.

The flow of this procedure is shown in Figure 3.7. This functionality is necessary so the endpoint knows if it should initiate an authentication setup (if there is no associated key in the server) or an authentication process (if there is an associated key in the server).

Authentication setup

The authentication setup between a client and the ACS server is illustrated in Figure 3.8. This happens the first time an identity communicates with an ACS server and allows the ACS server to be sure that it is always communicating with the same endpoint for any given identity. The endpoint sends its public key, attached to an

TABLE 3.1: Basic protocol format

Field Name	Type	Description	Required
Function	String	Name of function	Yes
Origin	String	E-mail address	Yes
Destination	String	E-mail address	No
Data	JS Object	Relevant data	No

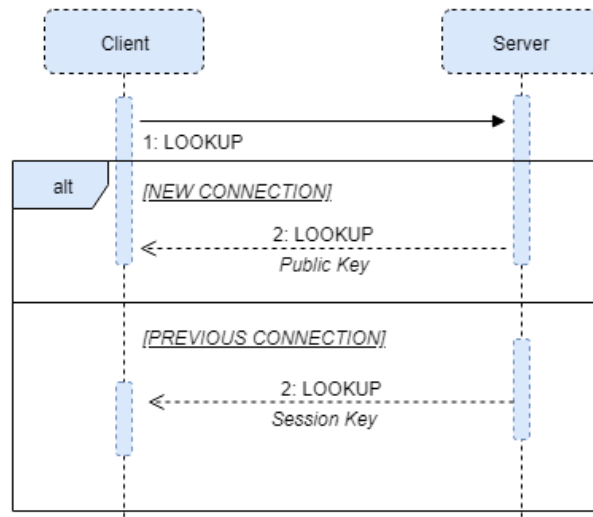


FIGURE 3.7: The lookup function exists so that the endpoint knows whether to start an authentication setup or an authentication process with the server.

Authentication setup packet, to the server. The server then checks if the e-mail is already registered. If it is, it notifies the endpoint by sending an *Authentication setup reply* packet, indicating that the setup failed. If the setup is successful, the server sends a symmetric key attached to an *Authentication setup reply* packet. This symmetric key is encrypted with the endpoint identity's public key, as part of the authentication.

Authentication

The authentication is done as described in [Section 3.3.1](#). The endpoint encrypts its e-mail address using the symmetric key, and sends it to the server in the data field of an *Authentication init* packet. The server then decrypts the cipher and compares the e-mail stored to the one received. If there is a match, the authentication is successful. If it does not match the connection is dropped. The result of the authentication is sent as part of the *Authentication reply* packet. See [Figure 3.9](#) for a diagram of this process.

Init

Once the endpoint has been authenticated by the server, it is available to receive offers and send offers. This is where the *Init* functionality comes in. This is an offer to send files to an identity. It contains information about who the sender is, who the intended recipient is, and metadata about the file to be shared. See [Figure 5.11](#) for an

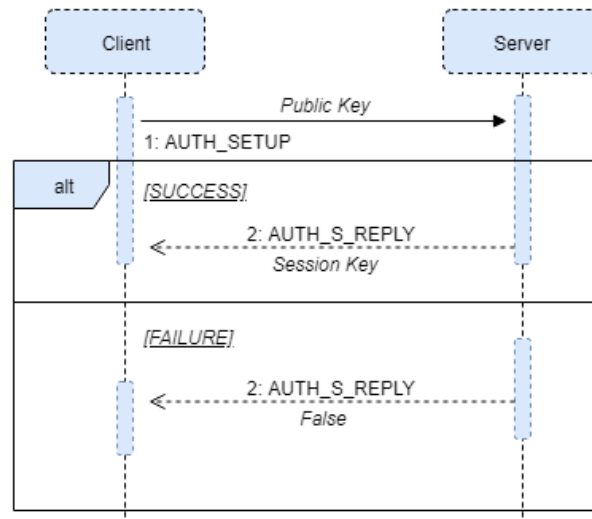


FIGURE 3.8: This shows how the authentication setup happens, and the data which is exchanged between the server and client in the two scenarios. The setup fails if the ACS server already has information stored for the given key or e-mail address.

example. If the server receives this type of request, it is forwarded to the indicated endpoint. The file metadata can be end-to-end encrypted without impacting the functionality.

Accept

Once an endpoint receives an offer to receive files (in the form of an *Init* packet), the user has the choice to accept or refuse the connection. If the user accepts, an *Accept* packet containing a WebRTC Offer is sent back through the ACS server.

Refuse

In the same way as explained for the *Accept* functionality, the *Refuse* functionality declines the offer to connect to the other endpoint. It terminates the connection setup between the two identities.

Answer

If the Sender receives an *Accept* packet, it means the other endpoint accepted the connection. As such, the Sender processes the WebRTC Offer received, and generates a WebRTC Answer. This is then shared back to the Recipient through the ACS server in the form of an *Answer* packet.

ICE

Since this solution utilizes ICE trickling (Described in [Section 2.3.1](#)), it needs a protocol to continuously share ICE candidates between the two endpoints. This function exists to allow the sharing of such candidates. In other words, the ICE candidate is attached as the data portion of the *ICE* packet.

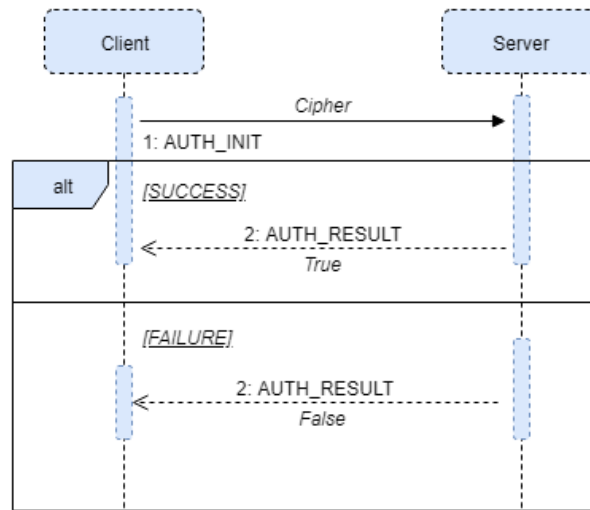


FIGURE 3.9: This shows how the server authenticates the user. If the decrypted cipher matches the expected value, the authentication is successful.

Done

This functionality indicates that the previous connection, for whatever reason, has ended. If one endpoint sends such a packet to the ACS server, the server should consider both endpoints available for new connections.

Error

If for any reason an error occurs on either side, this functionality allows for sharing what went wrong with the other endpoint. As such, the reason for the error is added to the data portion of the packet. There is no requirement to share error messages with the connection partner, but it can be useful in order to discern between an intended pause/stop in the connection, or if something went wrong.

Wait

This functionality is primarily aimed at allowing semi-asynchronous communication. It is used as a message to the endpoint from the ACS server, if they try to reach an identity currently connected to another endpoint. This indicates that the intended recipient is busy, but online. If one implements queuing, it can be used to tell the sender to wait until the endpoint is ready and then automatically connect.

3.3.4 Forwarding

The ACS is a service that exclusively forwards information, regardless of whether it is encrypted or not. This is done by having the Sender, Recipient, and type of communication unencrypted. The server needs this information in order to handle the requests, and as such, these fields are left in cleartext. Using this information, the ACS picks the correct functionality and executes it. For most functionality, that means just forwarding the information to the correct endpoint. It does, however, need to authenticate the user and take care of that authentication setup, therefore it supports functionality beyond just forwarding information. While some of the data

is unencrypted, it is important to note that the communications channel is encrypted, so an attacker cannot gain access to any of the data.

3.3.5 Semi-asynchronous transfer

The ACS solution can allow for semi-asynchronous messaging. This means that one can issue the connection request at any time, but wait for it to be handled until the other endpoint comes online. It also allows for a queuing system that stores requests and handles them at the first available opportunity. This still requires the sender to be online, connected to the ACS, and available for transfer. As such, while the actual transfer is synchronous, the request and user interactions are not, thus 'semi-asynchronous'.

This brought forward the need for the *wait* and *busy* functionality in the protocol, as well as a queue of jobs for each connection, temporarily kept in the ACS server. This type of transfer is not implemented in SendIt, but some of the frameworks and systems needed for such functionality is implemented and ready to be used.

Chapter 4

Implementation of SendIt

In this chapter the implementation specifics of SendIt will be discussed. It will go through how the application looks and functions, how the ACS server is implemented, and finally, how the application can be extended and improved.

4.1 Application

In this section, everything regarding the implementation of the application will be discussed, except the usage modes. It will go through the implementation and usage of SendIt and clearly explain how the different concepts and technology works.

4.1.1 Cryptography

In SendIt's implementation, the Web Crypto API (SubtleCrypto module) is used for generating and handling keys [43]. This allows for an easy and reliable way to standardize the handling of keys, encryption and decryption, in every system. The Web Crypto API is available in most internet browsers. It has been available through the Chrome browser, since the release of version 37 [44]. Using a pre-developed, and tested API for SendIt's cryptographic functions allows for a more reliable system. Since the implementation of the cryptographic functions and debugging has already been done, it allows more time to be spent on developing other functionality. This is the reasoning for using the Web Crypto API.

The system begins with creating a unique key pair, if no such key pair already exists. This check is done at the time of choosing to send or receive a file so that, if desired, the user can change to the correct identity for this connection. The key pairs used consists of two RSA-OAEP 2048 bit keys with SHA-1 hashing, but this is easy to change if the need arises. The choice of key-type was based on the recommendation of the WebCrypto API specification [43].

The key-exchange is done over the secure DataChannel created by WebRTC's PeerConnection [6]. Once a key exchange (successful connection) has taken place, the created key pair and the other identity's public key will be written to the disk.

SendIt imports the file into memory once a key is needed. After which it extracts the keys stored. Once the keys have been used and the communication has finished, SendIt will overwrite the existing file with the new, updated information. The file containing the keys has information exceeding just the known keys. Both the identity associated with a key, and the key itself, has to be stored.

As discussed in [Section 3.2.1](#), SendIt uses symmetric and asymmetric keys, since asymmetric keys can only encrypt small amounts of data. The maximum amount of data the asymmetric key pair mentioned previously (*RSA-OAEP 2048 bit with SHA-1 hashing*) can encrypt is 214 bytes [45]. SendIt uses an AES-GCM symmetric key, with

a length of 256 bits, which allows for encryption of up to 2^{39} -256 bytes of data [43]. This is well within the limits of what is necessary in SendIt.

4.1.2 Connection setup

This section will explain the functionality of generating the connection information and how each endpoint processes that information in order to create the connection. SendIt implements two different ways to share the connection information, in order to create a P2P connection between two endpoints. These two modes will be explained in [Chapter 5](#).

The Offer and Answer generation and processing is done as described in [Section 2.3](#). The difference from the normal usage is that they can also be encrypted before being transferred, which means they have to be decrypted before being used. The original form of the data is a JavaScript object.

To encrypt the data, it is first turned into a string, then to an ArrayBuffer, after which it is encrypted. To decrypt the data, it is converted from a string to an array of integers. It is then converted to a Uint8Array before being decrypted. The decrypted data is also a Uint8Array, which is turned into a string, and then back into a JavaScript object. All the conversions just mentioned, stem from the different data formats required by the different libraries and frameworks. For more information on how the data is exchanged, see [Chapter 5](#).

The actual setup of the connection and the creation of the DataChannel is done according to the examples and descriptions in [Section 2.3](#). The only difference in connection setup between the two modes, is if ICE trickling is used or not.

When the Offer and Answer is successfully exchanged, a direct connection is created. There are scenarios, when this is not case. Known causes of issues with completing the connection are:

- Setup not completed within a certain time frame (see [Chapter 6](#))
- One or both endpoints are behind symmetrical NAT
- One or both endpoints change network location (For example, connect to a different network.)

When implementing SendIt, the choice was made to not support symmetrical NAT, as it requires a TURN server. This means the connection would no longer be a direct end-to-end connection. Widespread use of IPv6 would solve this issue, as it would eliminate the need for NAT traversal and TURN servers. As for changes in network conditions, there is nothing to be done on the application side, except including a signaling server. As such, the system assumes that users will stay in the same network conditions for the duration the connection is active. For the ACS mode, automatic reconnection is an option which can be added as an extra feature.

4.1.3 File transfer functionality

The communication will consist mostly of file data, and as such, it is interesting to know how much data can be handled. In theory, splitting files into chunks during reading, and then transferring these chunks, allows for infinitely large transfers. The default max size for Node.js is approximately 1 GB for 32-bit machines, and 2 GB for 64-bit machines. This indicates the maximum amount of data that can be kept in memory. The reason for this limit is because this is the max amount of data the V8 JavaScript engine used by Node.js can have in memory at the time [46, 47].

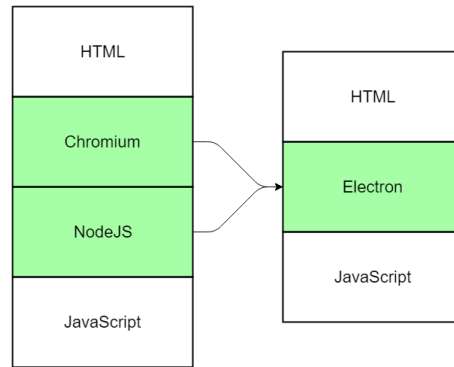


FIGURE 4.1: The stack for the prototype application. It is built using HTML, Electron, and JavaScript.

The current functionality separates each file into chunks of 1200 bytes, as is the limit imposed by the Chromium implementation of WebRTC [48]. The max file size is set to 160 megabytes during development, since it was the max size used by Pub-Share [42]. It is likely that this can be increased without any issues, since Node.js has support for keeping larger files in memory, but it would require some testing before being ready to deploy. See Figure 3.4 for an illustration of the previous explanation.

4.1.4 Programming languages, libraries and frameworks

SendIt is developed in JavaScript utilizing the Electron framework. These technologies were chosen because they allow for easy implementation, while supporting multiple operating systems. Easy connection setup and direct communication via P2P is also available, through pre-developed and tested frameworks. In addition, it also allows for the use of existing libraries and standardizations developed for web browsers, in desktop applications.

To clarify, this means that the user does not need to open their web browser to utilize SendIt, but can install it and run it like they would run any desktop program with a GUI. It also allows for easy creation of an installer file, which means the end user only needs to download and run the installer, for the program to be usable.

The electron framework uses Node.js for the back end and Chromium for the front end [49]. In practice, this means that the system is built on: HTML, Electron and JavaScript, where Electron utilizes Chromium and Node.js. (see Figure 4.1) This allows us to use modules and frameworks from any of the previously mentioned entities independently. Our implementation uses these libraries and frameworks:

- Node.js API - Used for reading and writing to the disk and finding correct files and folders [46].
- Chrome Web Cryptography API - Used to handle the creation and exportation of keys, encryption, and decryption [43, 44].
- Node.js Clipboardy - Utilized to automatically copy the generated Offer/Answer to the clipboard [50].
- Node.js Electron-prompt - Used to create pop-ups for requesting user input [51].
- Chrome Native WebRTC - Used for creating and managing WebRTC connections [52].

- jQuery v3.2.1 - Utilized to manage front end actions and dynamic updates [53].
- Bootstrap v3.3.7 - Used to manage front end modules and dynamic updates [54].

4.1.5 Program flow

The general appearance of the program will be explained in the following section. The functionality which is unique to each mode will be discussed in [Chapter 5](#). That means that most of the actual functionality is not explained here, but rather the installation, setup, and settings available. In addition, it will also explain the screens not related to the connection setup. These include the waiting screen, the transfer status screen, and the transfer complete screen.

Installation and launching application

The application comes in the form of installer files for Linux, Mac, and Windows. These files are in the format of *.deb* for Linux, *.dmg* for Mac, and *.exe* for Windows. The installers are very basic and require no interaction except executing them. After that is done, the image shown in [Figure 4.2](#) will appear and display a small animation. Afterwards, SendIt is installed and will be available. In most cases, the icon will be available on the desktop. Once the application is launched for the first time, a pop-up window will appear, as indicated in [Figure 4.3](#). Afterwards, the user is taken to the Home screen. This pop-up window will only be displayed the first time the program is opened.

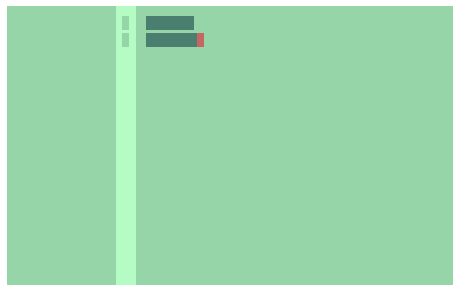


FIGURE 4.2: While the installation of SendIt is ongoing, this small box, with basic animations, will appear. Once it disappears, the program is installed.

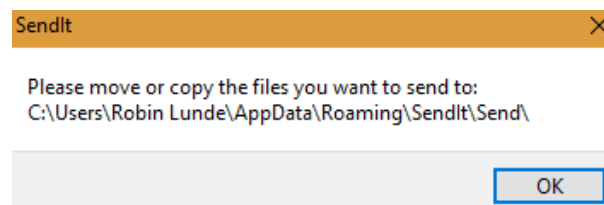


FIGURE 4.3: This pop-up appears the first time the program is started. It informs the user of the default location of the upload folder. The files the user wants to send should be placed in this folder.

Home screen

The Home screen is the first screen the user sees. On this screen, there is not much detail or information. The logo and acronym for SendIt is displayed, as well as the navigation bar. From here on, it is all about choosing the desired functionality or tweaking the settings to fit the users desire. The only difference between the Home screen for the two modes, is the formatting of the word 'Serverless' at the bottom of the screen, as well as the ACS mode not having a receive button on the navigation bar.

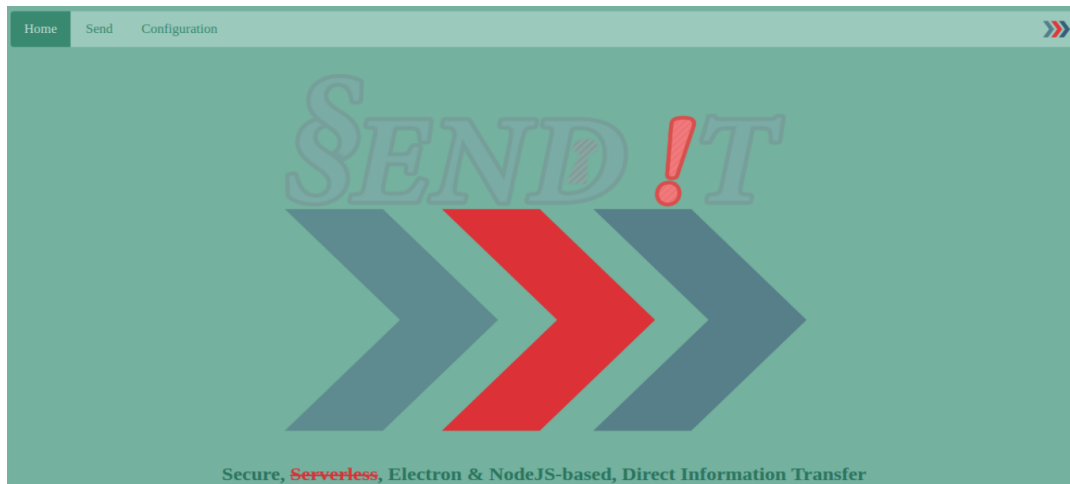


FIGURE 4.4: The Home screen displayed when the application is in ACS mode. The navigation bar has no button for receiving files. The 'Serverless'-part of SendIt's acronym is also crossed out, at the bottom of the page.



FIGURE 4.5: The Home screen displayed when the application is in Serverless mode. The navigation bar has a button for both sending and receiving files.



FIGURE 4.6: The navigation bar displayed in Serverless mode. The receive-button is not present in the ACS mode, since a separate page will be displayed if someone offers to send file(s) to you. This bar is always displayed at the top of the window.

Settings

The base view of the settings page is represented in Figure 4.7. At the top one can input which identity (or e-mail address if you will) to use. Further down, one can remove the configuration file, the file which stores all the data about which identity and which settings to use. Following are radio buttons, where one can choose which mode to use and if one wants to use custom locations, for example, where to store downloaded files. At the bottom, information about the current settings are displayed. Finally, there is a 'save changes' button, to store the changes made.

In Figure 4.8, more detailed options are displayed. These appear when clicking the non-default option of the radio buttons. If the ACS option is selected for *Mode selection*, the field for indicating the address of the server is displayed, as well as a 'save' button and a 'reset' button. Afterwards, there is an option for manually selecting a file from which to load keys. One can also remove the file currently used, by pressing the 'remove ALL current keys!' button or remove individual keys by clicking on the corresponding e-mail address. Finally, one can customize the download and upload folder location.

FIGURE 4.7: The screen used for indicating user preferences, upload and download locations, identity management, and mode selection.

FIGURE 4.8: The expanded version of the settings screen with the selections and menus displayed.

After successful connection setup

All the following examples are taken from the Serverless mode, but they look identical in the ACS mode, with the exception of the navigation bar. These are the different screens shown once the endpoint has completed their part of the connection setup exchange.

Waiting screen

The waiting screen is displayed while the endpoints are waiting for WebRTC to establish the P2P connection.

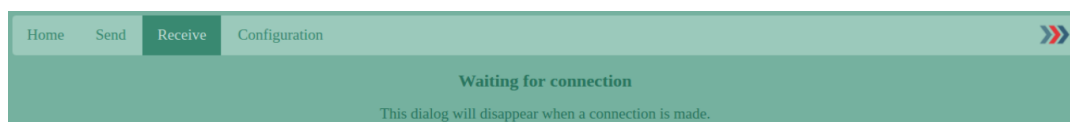


FIGURE 4.9: This screen is displayed while waiting for the endpoints to connect via P2P (WebRTC).

Transfer screen

It displays details about the current file being transferred; it's name and type, as well as the total number of files to transfer. It also shows the percentage of data

transferred for the current file. Finally, there is a cancel button in case one end wants to stop the transfer.

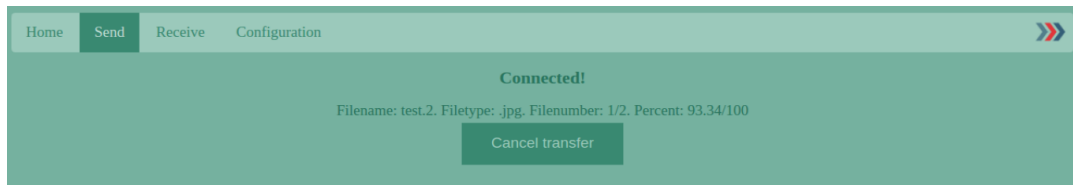


FIGURE 4.10: This screen displays details about the status of the current transfer.

Connection completed screen (*Sender*)

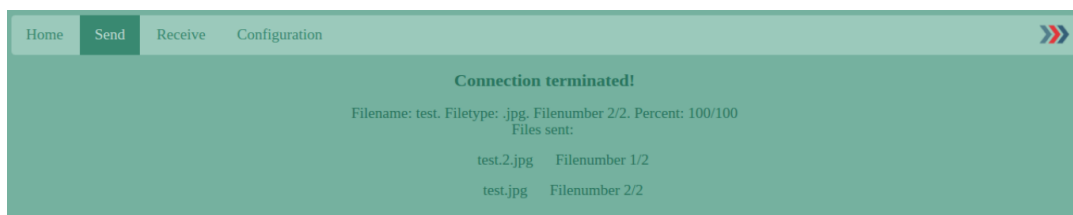


FIGURE 4.11: This screen displays details about which files were transferred.

Connection completed (*Receiver*)

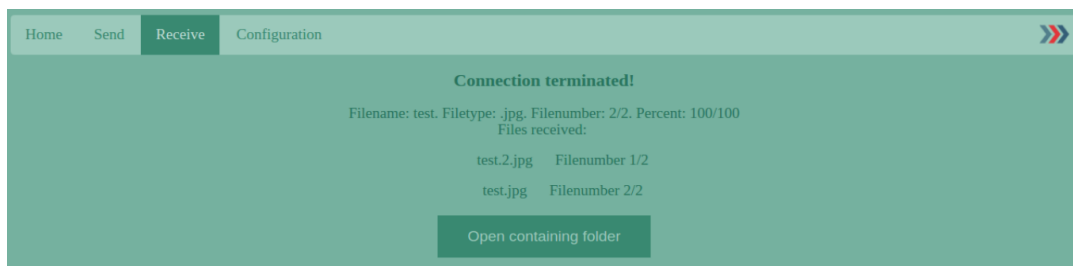


FIGURE 4.12: This screen displays details about which files were received. It also has an 'open containing folder'-button for easy access to the received files.

4.2 ACS server implementation

The ACS server is implemented in JavaScript, using the Node.js environment. It uses a library implementing the Web Crypto API for Node.js [55], which allows for the use of the same keys, the same encryption scheme and generally the same cryptographic solutions as in the application. This makes it easy to use and means there is no need for developing support for, or using other cryptographic methods, to authenticate with the ACS server.

TABLE 4.1: Lookup packet

Name	Type	Argument details	Required
res	Boolean	Indicates if authentication setup is needed or not	Yes
key	JWK	Server's public key in JWK format	Yes
wrap	Array	Encrypted symmetric key	No

TABLE 4.2: Authentication Setup packet

Name	Type	Argument details	Required
key	JWK	Client's public key in JWK format	Yes

4.2.1 WebSockets

All communication between endpoints and the ACS server is done over WebSockets using HTTPS. This enables bi-directional communication at any time and creates an easy interface to use for communicating with different endpoints. The communication is exclusively done using the protocol described in the next section.

The WebSocket interface and API also allows for easy handling and management of clients. Clients can connect and disconnect randomly without affecting the service as a whole. All clients are treated equally and it is easy to address each client individually. Because of this, it is very easy to receive information from one client and immediately forward it to the intended recipient.

4.2.2 Protocol

This section reviews the implementation of the protocol discussed in [Section 3.3.3](#). For an overview of the packet format, see [Table 3.1](#). The format indicated for each of these packets, goes in the data field of the general packet format. Following is the practical implementation of the protocol.

Lookup

The *Lookup* packet from client to server does not contain any data. The packet sent from the server to the client can contain the data indicated in [Table 4.1](#). If *res* is *true*, the data will consist of the fields *res*, *wrap* and *key*. The *wrap* field consist of the symmetric key, encrypted with the other endpoint's public key. If *false*, it will consist of *res* and *key* only.

Authentication setup

The possible arguments used by the Authentication setup and Authentication Setup Reply functionality are shown in respectively [Table 4.2](#) and [Table 4.3](#). The public key of the client is sent to the server. The server tries to set up authentication for subsequent connections. The authentication setup is considered a success if no previous data is stored for this e-mail or public key. If this is the case, the *Authentication Setup Reply* packet will contain *res* and *wrap*, where *res* is set to *true*. The *wrap* field consists of the symmetric key, encrypted with the other endpoint's public key. If the setup fails, the *wrap* field is not included.

TABLE 4.3: Authentication Setup Reply packet

Name	Type	Argument details	Required
res	Boolean	Authentication result	Yes
wrap	Array	Encrypted symmetric key	No

TABLE 4.4: Authentication packet

Name	Type	Argument details	Required
ciph	Array	Client's encrypted e-mail address	Yes

Authentication

Table 4.4 and Table 4.5 shows the possible arguments for the authentication packets. The *ciph* field contains the client's email address, encrypted with the symmetric key. The *Authentication Result* packet returns a boolean value directly in the data field, indicating the result of the authentication process. Successful authentication returns *true*.

Init

The *Init* packet can consist of the data shown in Table 4.6. It contains information about the files being offered. Each object in the *files* array has information about the file name, file type and the size of the file.

Accept

The *Accept* packet contains the WebRTC Offer generated and indicates that the endpoint wants to receive the data previously offered. The WebRTC Offer can either be encrypted or in cleartext. See Table 4.7 for more information about the data transferred. If the Offer is sent in cleartext, the data transferred is just the WebRTC Offer object. If not, the data consists of the other three fields (*wrap*, *iv*, and *ciph*). The *wrap* field consists of the symmetric key, encrypted with the other endpoint's public key. The *ciph* field consists of the WebRTC Offer, encrypted with the symmetric key.

Refuse

The refuse packet contains no data. If this packet is received the connection setup is stopped.

TABLE 4.5: Authentication Result packet

Name	Type	Argument details	Required
-	Boolean	Authentication result	Yes

TABLE 4.6: Initiate Connection packet

Name	Type	Argument details	Required
files	Array	Array of objects with file data	Yes

TABLE 4.7: Accept packet

Name	Type	Argument details	Required
-	Object	The WebRTC Offer generated by the endpoint	No
wrap	Array	Encrypted symmetric key	No
iv	Array	Initialization vector for the symmetric key	No
ciph	Array	Encrypted WebRTC Offer	No

Answer

The *Answer* packet contains the WebRTC Answer generated by the endpoint. The WebRTC Answer can either be encrypted or in cleartext. See [Table 4.8](#) for more information about the data transferred. If the Answer is sent in cleartext, the data transferred is just the WebRTC Answer object. If not, the data consists of the other two fields (*iv* and *ciph*). The *iv* field consists of the initialization vector, encrypted with the other endpoint's public key. The *ciph* field consists of the WebRTC Answer, encrypted with the symmetric key.

ICE

This packet contains the data showed in [Table 4.9](#). The data can either be encrypted or sent in cleartext. If it is not encrypted, it will be sent as an ICE candidate object. If it is encrypted, it will consist of the fields *ciph* and *iv*. The *ciph* field consists of the ICE candidate, encrypted with the symmetric key.

Done

The *done* packet contains no data, and indicates that the connection is terminated and that the endpoints are ready for a new connection.

Error

The *error* packet can contain data about which error occurred. If the server receives an *error* packet with an indicated destination, it will forward it to the correct endpoint. The data will have the format indicated in [Table 4.10](#).

TABLE 4.8: Answer packet

Name	Type	Argument details	Required
-	Object	The WebRTC Answer generated by the endpoint	No
iv	Array	Encrypted initialization vector for the symmetric key	No
ciph	Array	Encrypted WebRTC Answer	No

TABLE 4.9: ICE packet

Name	Type	Argument details	Required
-	Object	The ICE candidate generated by the endpoint	No
ciph	Array	Encrypted ICE candidate	No
iv	Array	Initialization vector for the symmetric key	No

TABLE 4.10: Error packet

Name	Type	Argument details	Required
-	String	Error details	No

Wait

The *wait* packet does not contain data. It indicates that an endpoint is busy, and, as such, cannot partake in a connection at this time. If a destination is specified, the server forwards the packet to the correct endpoint.

4.3 Extendability and improvements

In this section the extendability of SendIt, in general, will be assessed. SendIt can be used as a platform to build extended functionality, and as such, it should be noted in what way this can be done, and how one stands to benefit from doing it. It will also discuss possibilities for improving the current implementation.

4.3.1 Key storage encryption

The file where keys are stored should be encrypted and password protected, or otherwise access restricted. The key file should only be usable if the correct password is provided. If the wrong password is provided, the keys should not be accessible. The key file should be updated every time it is used by the system. This will allow users to change the password used to access the keys between each use and also make it easy to update information regarding each key. In addition, it gives no guarantee that the same encryption is used each time, which makes attacks over time harder to execute, since there is no reliable way to analyze changes or patterns in the way the file is stored. This functionality should be implemented and would improve the solution. It is not currently implemented in SendIt, due to time constraints.

4.3.2 E-mail verification

One way to extend the current functionality is to add e-mail verification to the process of registering an identity. This would increase the trustworthiness of each identity since proving ownership of the registered e-mail address would be a necessity. It would however, also include all the issues stemming from how the e-mail system is implemented. It would also make the registration process harder and require more from the users before being able to use the system. Because of these issues, it is not included in the system by default, but can easily be added. It is left up to the end users to develop and extend the proposed system, if such functionality is desired.

TABLE 4.11: Fields included in the record of communication.

Sender	File(s)	Date	Completed
test@email.com	picture.jpg	2018-03-20	0
another@email.com	document.doc	2018-05-13	4

4.3.3 Support for bigger files

Supporting bigger files can be achieved by reading in chunks of the file. Then, once a chunk is completely transferred, the next chunk is read into memory. This will allow both endpoints to handle smaller amounts of data at a time, while still having transmitted the whole file after the transfer of all chunks have been completed. It is not recommended to implement this until after the 'resume transfer' feature is implemented, as transferring large amounts of data, without any way of resuming it in case of failure, is less than optimal.

4.3.4 Resume transfer

This functionality can easily be implemented based on a communication record. Since every identity will have a list of previously transferred files, it will have the file name included. If the transfer is not completed, this record can store information about which chunk of the file was the last to be received, and the endpoint can request the transfer to be continued from there.

If the sender is not willing to resume the previous transfer, it can either start over, or the sender can offer to transfer another file. This decision is up to the sender's settings and/or preference. If the sender chooses to not resume the transfer, the data previously stored on the receiver's local system should be removed, and the record updated as a failed transfer. The system should only allow for the requested file to be shared on the subsequent connection.

To clarify: Alice tries to send Bob *File A*, but the connection is broken. If Alice tries to send *File A* again, it will resume from the last chunk received. If it fails again, it will also allow for the transmission of *File A* to be resumed. However, if Alice contacts Bob again, but tries to send *File B* this time, the previously transmitted information (*File A*) stored by Bob should be removed.

The record of communications can be implemented by creating a log that contains the fields indicated in [Table 4.11](#). The last field can either be -1 (meaning failed), 0 (meaning success) or the number of the last received chunk. This is useful for being in compliance with the GDPR, allowing the user to keep track of their interactions for reviewing their activity, and implementing a 'resume transfer' functionality, as mentioned.

4.3.5 WebRTC IDP inclusion

WebRTC comes with a suggested standard for implementing Identity Provider services. An Identity Provider is a trusted third party that corroborates an identity. An example would be connecting one's Facebook account to an identity, as a means for other endpoints to verify the authenticity of that identity. This is possible for many different services and can be a means of increasing trust in identities. Currently, there are some arguments and disagreements on how this should be implemented in WebRTC, and as such, there are very few existing frameworks that can be used.

This is expected to change, and at that point, using these services will allow for an easier way to increase trust in endpoints.

The trust is of course reliant on the end user already trusting the service that is used as the Identity Provider, and that the identity is as expected. For example: If the Identity Provider used is a known service (such as Facebook), one can reasonably trust the data received. If it is from a service unknown to the user, then the identity provision does not increase the trust at all, since the data may be created for malicious purposes. In the same way, if the endpoint is expecting to be communicating with Alice, but Bob's identity is asserted by the provider, the end user should be sceptical.

In summary, this functionality would allow one to link accounts from other, independent services with their WebRTC connection, in order to corroborate the endpoint's identity and increase trust.

4.3.6 SendIt as a platform

This is an interesting idea since the proposed system allows for connection setup and identity assertion. One can use SendIt for this functionality and build any kind of additional functionality on top, if so desired. Especially combining with VOIP, which WebRTC is often used for, can be useful. It allows the developers to focus on their services and additional functionality, while allowing the easy to use and secure setup offered by SendIt to take care of identity management and authentication. The design and implementation of SendIt is modular, which means one can easily pick and choose which functionality one wants to utilize, and discard the rest. This makes it easy to take advantage of the wanted functionality, while not complicating the solution by including the functionality that is not useful to the specific scenario at hand.

Chapter 5

Operation modes

In this chapter the in-depth design and functionality of SendIt will be discussed. The program flow and use of the application will be illustrated and the two different modes will be contrasted and compared.

SendIt consists of two modes: Serverless and Assisted Connection Setup (ACS). The difference between these modes is how the setup of the connection is completed, as illustrated in [Figure 5.1](#). This affects the flow of the program and also impacts how easy it is to use. In the following sections, the differences will be clearly shown and suggestions on how to best utilize the system will be given. It is important to note that neither mode changes the required technical knowledge to use the program - only the interactions required. This is to make sure all users can utilize both modes.

5.1 Serverless mode

Serverless mode is a mode completely independent of servers (as the name indicates) when establishing the connection. This mode allows the users to be unpredictable in how the Offer and Answer exchange is done and maximizes the improved security of SendIt. It does, however, slightly lower the ease of use and requires more from the end user.

5.1.1 Use case

As indicated previously, the use case for this mode is by and large for those who value security over usability. This mode should be used if you are already having a conversation with the other endpoint and can quickly take care of the connection

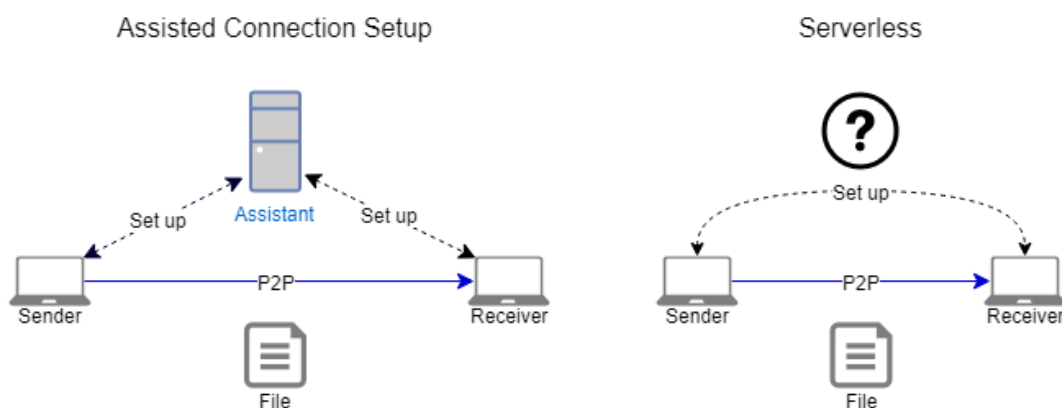


FIGURE 5.1: Left: Assisted Connection Setup mode, Right: Serverless mode

setup. Say Alice is talking to Bob and wants to transfer a file. Alice and Bob then take care of the connection setup using an instant messaging service. This allows for a quick and easy file transfer, while getting the added bonus of having the file transferred in a very secure manner. Another example is if Bob wants to transfer an extremely sensitive file to Alice. They can then make a plan on what time they are to do the transfer, in order to be able to take care of the connection setup quickly. This way the file transfer is done with the highest level of security achievable in the system.

As illustrated above, at the cost of some of the usability this mode gives full control to the end user as to how the sharing of the Offer and Answer is done. It does require more manual interactions as well as more planning. This is because the lifetime of the Offer and Answer exchange is short, and as such, puts a time limit on the exchange. It is also because the user has to manually transfer the Offer-/Answer generated by their own application and manually input the Offer/Answer received from the other endpoint. There is no change in the technical knowledge requirements to use this mode, just the time constraint and manual interactions as mentioned previously.

5.1.2 Program flow

The program flow is explained from after the functionality is chosen. For information about functionality other than Send and Receive, see the explanation in [Section 4.1.5](#). The program flow of the Serverless mode varies depending on if the user is taking the role of Sender or Receiver. As such, the next section will take you through the two different roles, the choices and screens showed, as well as what is going on. If the user has not yet chosen an identity, this screen will always be the first to appear after choosing the intended functionality (*Send/Receive*):

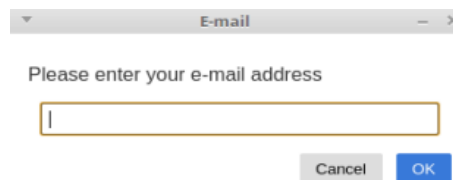


FIGURE 5.2: This screen displays a pop-up forcing the user to choose which identity (e-mail) to use.

Sender:

First screen

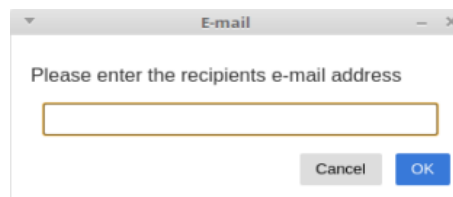


FIGURE 5.3: This screen displays a pop-up forcing the user to input the Recipient's identity (e-mail).

Second screen

On this screen, an empty white box appears. Once the Offer is ready, it will appear in the box along with a notification informing the user that it has been copied to the clipboard automatically. The Offer can either be in cleartext or encrypted. Underneath, there is data about which file(s) are to be transferred. It also indicates the file in red if it is not included in the transfer. Finally, there is a continue button that is only clickable after the Offer has been shown.

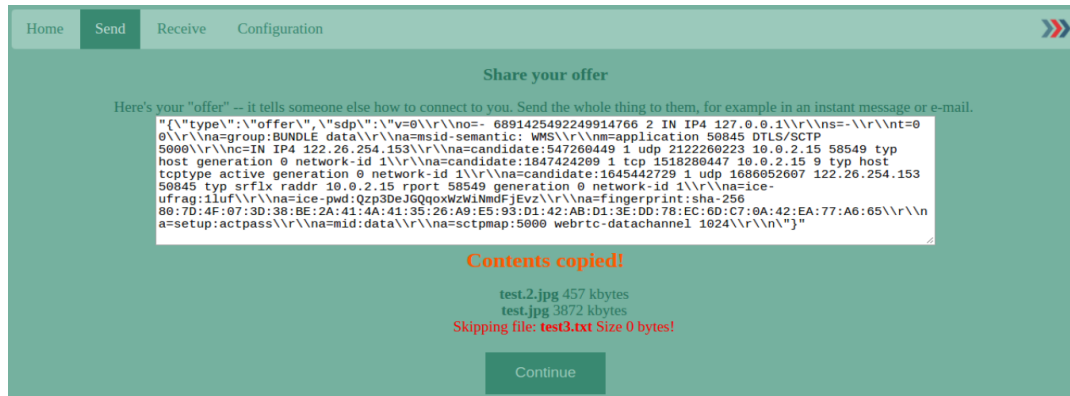


FIGURE 5.4: This screen displays the Offer and the file(s) to be shared. The user should share this Offer with the intended Recipient.

Third screen

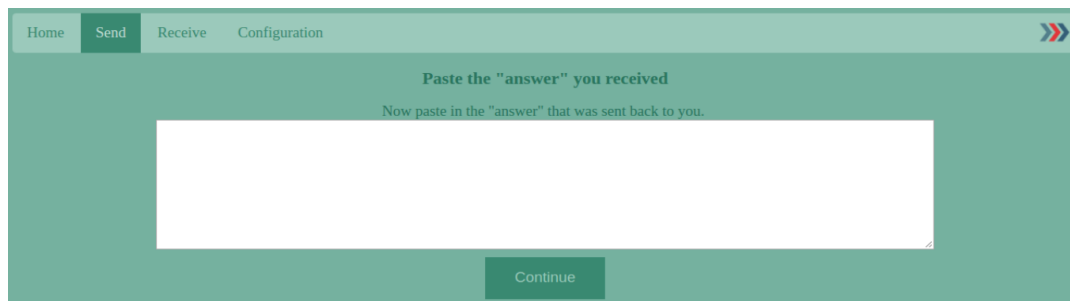


FIGURE 5.5: This screen displays another white box where the user should paste the Answer received from the Recipient. The continue button is only clickable after some information has been entered into the box.

Receiver:First screen

FIGURE 5.6: This screen displays a pop-up forcing the user to input the Sender's identity (e-mail).

Second screen

FIGURE 5.7: This screen displays a white box where the user should paste the Offer received from the Sender. The continue button is only clickable after some information has been entered into the box.

Third screen

FIGURE 5.8: This screen displays an empty white box until the Answer is generated. Once it is generated, it appears in the box along with a notification that it has been copied to the clipboard. The user should share this Answer with the Sender.

Shared:

After the steps previously explained, the functionality and screens are the same for both the Sender and the Recipient. When these steps are completed, if no error or problems occur, it will follow the description given in [Section 4.1.5](#).

5.1.3 Implementation specifics

The biggest difference in implementation is of course the connection setup. Since the serverless solution has no set way of doing this, the connection setup will be discussed as part of the ACS mode, in [Section 5.2.3](#). In Serverless mode, the Offer and Answer is displayed in the form of a string. The string can either be a cipher or a cleartext representation of the Offer/Answer.

Since this part of SendIt has user interactions and relies on converting user input into JavaScript objects, it was at risk for XSS attacks and other code execution/injection attacks. As such, input sanitation was added to make sure that such attacks can not be executed. All data in this phase, both exported and imported, has to conform to the JSON format, which is then parsed and turned into JavaScript objects and used for establishing the connection via WebRTC.

5.1.4 Possible improvements & extendability

This mode allows for no server-involvement, which in itself is a great way for companies to reduce expenses. For SendIt, it is only applied to file transfers via DataChannels. If this is applied to VOIP, which is currently one of the the most used applications of WebRTC, it could allow companies to increase profit margins. This does, however, set requirements for improving the lifetime of the connection setup. Specifically, the Answer will probably need to have it's lifetime doubled or tripled, for it to be usable in this scenario.

If one can somehow make changes to the current WebRTC system and make endpoints permanently addressable by a specific Offer and Answer combination, or allow the Offer and Answer to be re-usable, it would also benefit this solution a lot. Of course these permanent addressing or reusable Offers and Answers would have to be unique for each pairing, which might also raise some new issues. Working on extending the lifetime of the connection setup is objectively the best and most effective way to improve this solution.

5.2 Assisted Connection Setup mode (ACS)

Assisted Connection Setup mode alleviates some of the difficulties and problems from the Serverless mode, by automatically taking care of the connection setup. This is done through a server communicating by WS over HTTPS. It uses a custom protocol on top of WS, designed for sharing connection information for WebRTC, to control and understand the communication between the endpoints.

The server can be explicitly chosen by the users. They can host their own or choose to use an existing one offering this service. The only requirement is adhering to the pre-configured protocol. For details about the protocol, see [Section 3.3.3](#). Because of this, the system is still unpredictable and hard to attack.

Another security feature is that there is no need for the server to send code to be executed or loaded by the client, as this functionality is pre-programmed in the client application. As such, the server has *no influence* over the functionality of the client. The server only authenticates the user and forwards the connection information to the other endpoint. In addition, all communication with the ACS happens over a secure channel, so an attacker would have to break the server itself to gain access to the information relayed. For more information about how the ACS server and the client communicates, see [Section 4.2.1](#).

5.2.1 Use case

The use case for this mode is when planning or co-ordinating with the other endpoint is not done in advance or hard to do. It is for when one is willing to sacrifice a small degree of security to improve the usability. This mode allows the connection request to be sent first, and only when the other endpoint has agreed to take part in the exchange, is the connection setup done. As such, the lifetime of the Offer and Answer is not an issue, since the exchange does not happen until both endpoints are online and ready to exchange information. It happens through an intermediary, so the exchange is almost instant. Since this intermediary is a point of attack and monitoring, it adds a small weakness to the system compared to the Serverless mode.

5.2.2 Program flow

The program flow is explained from after the functionality is chosen. For information about functionality other than Send and Receive, see the explanation in [Section 4.1.5](#). The ACS mode program flow differs depending on if the user has the role of Sender or Receiver. As such, the program flow of both of the roles will be shown and explained below.

Both roles begin with this screen, when entering ACS mode (This can be when starting the application, or when switching modes):

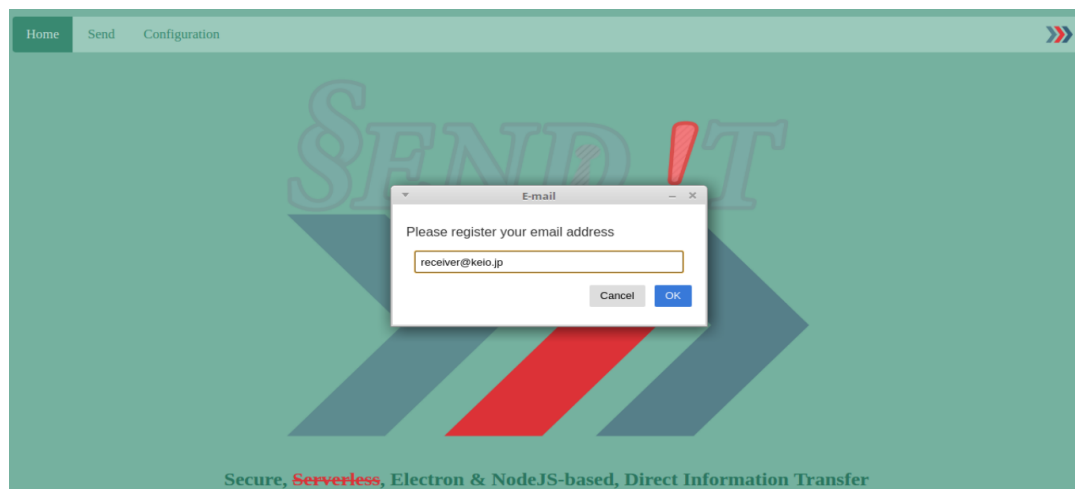


FIGURE 5.9: The ACS mode requires an identity to be given, in order to be able to authenticate with the ACS server. This is also done so the ACS server knows where to forward data for each identity.

Sender:First screen

FIGURE 5.10: This screen displays the e-mail address for the current identity, a field to indicate the Receivers e-mail address, and information about the file(s) to be transferred. If a file is marked in red, it means it will not be included.

Receiver:First screen

It displays the e-mail for the current identity, the e-mail address of the intended Receiver (as indicated by the Sender), the Senders e-mail address, and data about the file(s). The user has two buttons: Accept and Decline. If the decline button is pressed, the user returns to the home screen.

FIGURE 5.11: This screen notifies the user that a connection request has been received and displays details about the Sender and the file(s) offered.

Shared:

After the previously explained steps are completed, if no error or problems occur, it will follow the description given in [Section 4.1.5](#). If an error occurs, a message will appear in the position shown in the next illustration, but on the screen currently being displayed.

Error screen

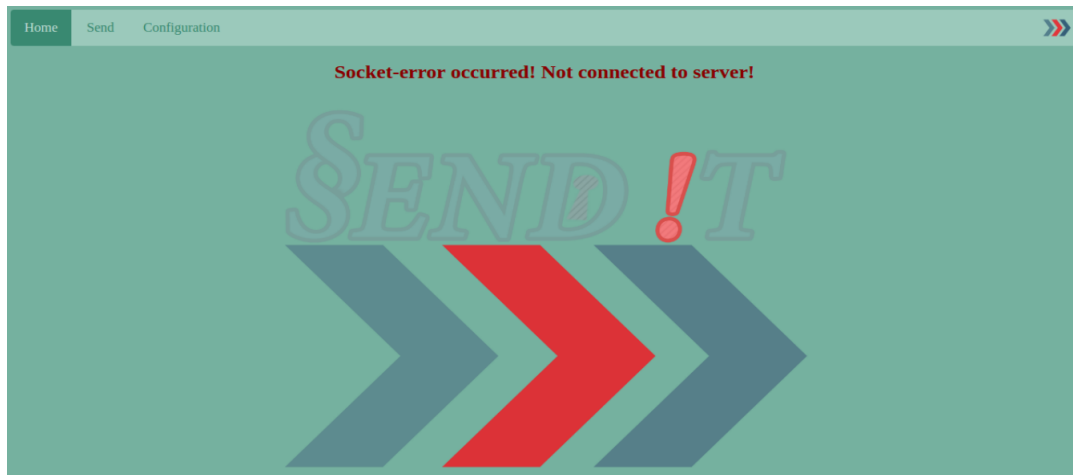


FIGURE 5.12: This screen shows where the error will appear, if there is one. The message varies depending on which error occurs.

5.2.3 Implementation specifics

The biggest difference in implementation comes in the form of how the connection setup is exchanged and processed. The way of communicating with the ACS server and how the WebRTC connection setup is done, will be explained in this section. The functionality for actually sending the file(s) stays the same, but since the connection setup happens through a server, it needs a secure way to interact with the server. In this section the client-side implementation will be described. For more information about the server and the protocol used, please see [Section 3.3](#).

Communication:

The ACS mode utilizes WebSockets to communicate with the server. It also forces the use of HTTPS to make sure that the communication is secure. All communication has to be in the format of the protocol described in [Section 3.3.3](#). This means that all information shared has to be processed and then correctly represented. Based on the information exchanged with the endpoint via the ACS, a P2P connection is established, just like in the Serverless mode. At this point the ACS server is no longer utilized. Once the connection has finished, for whatever reason, the ACS server is notified that the connection is over and that the endpoint is ready for a new connection.

WebRTC Connection setup:

There are two major differences in how the connection setup for the P2P channel changes in this mode. First of all, the exchange does not just happen in the form of one Offer and one Answer. The method called ICE trickling, as described in [Section 2.3.1](#), is used. This separates the Offer and Answer from the ICE candidates. It allows for continuous exchange of ICE candidates, as well as a faster exchange of the Offer and Answer, since they can be shared without waiting for all the ICE candidates to be gathered before being shared. As such the lifetime of the Offer and Answer is irrelevant in this mode.

Secondly, the connection setup does not happen until both endpoints have agreed to connect to each other. This means that the endpoints are ready to receive and process the Offer and Answer, which allows the exchange to happen almost instantaneously. This minimizes the risk of any issues arising when doing the exchange and allows for an efficient connection setup. It also means that the endpoints do not redundantly create an Offer, thereby wasting time, in the case that the other endpoint declines the connection.

Finally, the roles of Sender and Receiver in SendIt, compared to in WebRTC, is reversed. This is to avoid the redundant creation of an Offer, as mentioned in the previous section. The Receiver in SendIt will receive a connection offer in the form of the screen showed in [Figure 5.11](#). If the endpoint accepts this connection offer by pressing the accept-button, a WebRTC Offer will be generated and exchanged via the ACS server. Afterwards, functionality will be the same as in Serverless mode.

5.2.4 Possible improvements & extendability

Most of the improvements and extendability for this mode will rely on making changes to the current functionality of the ACS server. It would also require changes to the ACS client code, and as such, the discussion will be done in this section.

Key request:

Allowing the endpoints to request the key for an identity via the ACS server, would allow for encrypted end-to-end communication at all times. This requires some guarantee from the owner of the key to be attached, so the ACS server can not easily set up a MITM-attack. This would remove some of the issues with trusting the first exchange, and also allow for better trust evaluation and improved key exchange. The difficulty arises in the fact that the information needs to have a guarantee that it belongs to the correct identity, without having any previous or additional communication. One way to achieve this functionality is to go through the additional channel of using e-mail as a means of exchanging some shared secret, but this is left up to future work.

Multicast:

Extending SendIt to allow *false multicast* messages would mimic the way current e-mail attachments work and make sharing a file with multiple recipients at once easier. The reason for calling it *false multicast* is because the transfer would still be one to one, but for the sender, it would seem identical to sending data to a multiple recipients. While this may allow for rapid spreading of malicious files, it would also improve usability in legitimate use cases. A way to achieve this is to have each client connect to a 'room' in the server, where the sender has to authenticate endpoints before/upon entering. After being authenticated, a P2P connection between the sender and the endpoint can be established and the file transferred.

Automatic reconnect:

Automatically reconnecting after an endpoint has been disconnected or gone offline, would be a huge improvement to usability and user friendliness. It would allow for a very smooth user experience and increase the stability and reliability of the system. A way to achieve this would be to classify a connection as broken in the ACS server. This can be done by having endpoints report to the server if their connection

is broken. The server then classifies the other endpoint as unavailable. Once the ACS server detects that the endpoint is once again available, it would automatically start the connection setup for both endpoints. The endpoints should have functionality that automatically accepts and connects, if this is the case.

Pausing connections:

Allowing for pausing connections would increase user friendliness and help alleviate some issues that present when using synchronous communication. It would require some way to resume transfers from their previous point, but with such functionality in place, it should not be too hard to implement pausing connections. This would primarily be useful when being in the middle of a connection, when the need for transferring another file urgently occurs. Say the user is transferring a movie to a friend, when all of a sudden an important document from work needs to be received. Being able to temporarily pause the movie-transfer, receive the document, then resume the movie-transfer would be very useful. Having it automatically done through the ACS server, would be preferred.

Chapter 6

Experiments

In this chapter the experiments conducted will be discussed. The reasoning behind doing them, the way they were conducted, as well as the results, will be explained. The impact of the results and what the results mean for the usage of SendIt will also be clearly stated and evaluated.

6.1 Motivation and contribution

The original goal of doing these experiments was to evaluate the time limit for successfully establishing a connection, in regards to the WebRTC Offer and Answer exchange (from now on referred to as the connection setup). The Serverless mode requires manual interactions from the users, so knowing the time constraint is important for using the system optimally. In addition, if improvements were to be implemented, it would be necessary to have a benchmark to test against. It would also give a good indication on how to best proceed with improving the solution. As an added bonus, the setup for the experiments can also function as a framework for testing possible improvements and enhancements to the system.

The motivation for doing these experiments was to test the lifetime of the connection setup. Based on the results, the usability and limitations of the Serverless mode would be clear. This indicates the advantages and limitations of the Serverless mode, which allows for better decision-making when choosing modes. It also allows for a clear definition of the potential problems with the serverless solution and gives a tangible, concrete frame of reference for improving the solution. In more detail, the goal of the experiments can be divided into four parts:

- Find the approximate lifetime of the Offer
- Find the approximate lifetime of the Answer
- Find the approximate lifetime of the connection setup
- Find the most impactful part of the connection setup

Since the experiments are meant as a reference and benchmark for SendIt's functionality, the setup mimics SendIt's connection setup and exchange as much as possible. This means the results of the experiments should be directly transferable to SendIt and the results should reflect what to expect from SendIt's current functionality. The results are also similar to the conditions and results experienced when developing and testing SendIt.

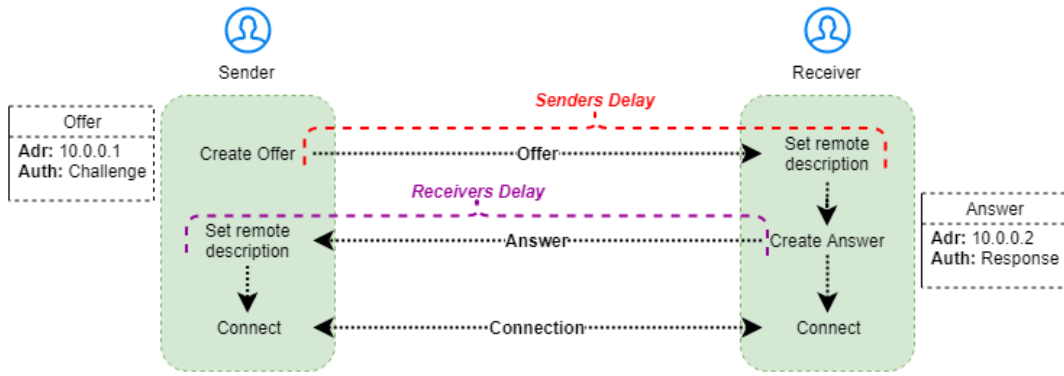


FIGURE 6.1: Simplified illustration of experiment setup

6.2 Terminology

The terminology is as follows:

Offer: WebRTC connection offer generated by the Sender in order to initiate a connection with the receiver.

Answer: WebRTC connection answer generated by the Receiver, from the Offer received, to complete the connection setup.

Sender's delay (S): is the delay from creation of the Offer until it is received by the other endpoint.

Receiver's delay (R): is the delay from creation of the Answer until it is received by the other endpoint.

Split delay: is the delay split equally between S and R . If the delay is 10 seconds, it means 5 seconds Sender's delay and 5 seconds Receiver's delay, for a total of 10 seconds delay. For a better understanding of these terms, look at the simplified representation in [Figure 6.1](#).

6.3 Implementation and results

In these experiments, the success-rate of connections between a server and a client was measured. The connection was created using WebRTC with a varying delay before the Offer and/or Answer was shared. The experiments were conducted by having users connect to a web server over HTTP and naturally having the server deliver content via HTTP. After the necessary HTML and JavaScript had been delivered to the client, they would try to establish another connection to the server using WebSockets. This allows for bi-directional and efficient communication. Over this WebSocket connection, the connection setup for the following tests was done. The P2P connection and setup was done utilizing the WebRTC PeerConnection functionality. The whole experiment consisted of three test sets:

- Test set 1 - Sender's delay (S)
- Test set 2 - Receiver's delay (R)
- Test set 3 - Split delay

Each of these test sets consisted of five tests. Once all test sets were finished (or when the client disconnected), the client shared their log with the server, and the

server stored both logs for analysis and comparison. The pattern described below indicates how one test was done. Once this had been completed for all five tests in all three test sets (for a total of 15 tests), the experiment was considered finished, and the connection torn down. If a client disconnected before finishing all tests, the results for the tests completed up until that point was shared with the server, and stored.

The tests were done as follows: For each test, the respective WebRTC Offer and Answer was exchanged. Depending on the test case, the sharing of the Offer or the Answer, or both, was delayed by varying length. Once the exchange had completed, the server and the client tried to establish a WebRTC P2P connection. Once this was done, a status check was immediately issued in order to check the result of the connection establishment. The results of this status check was used to indicate whether the connection was successful or not.

It is important to note that in initial tests, the results of the status check was noted before trying to utilize the P2P channel to do actual data transfers. In these tests, the result of the status check always correctly indicated whether the connection was successful or not. As such this status check was considered sufficient to indicate the result of the connection establishment. The result of the check was then logged for both server- and client-side. The P2P connection was then torn down and a new connection setup began.

For the illustrations in the following sections, the percentage of successful P2P connections is represented on the vertical axis, and the total delay before trying to establish the P2P connection on the horizontal axis. Each test set has a different type of line.

6.3.1 Experiment 1

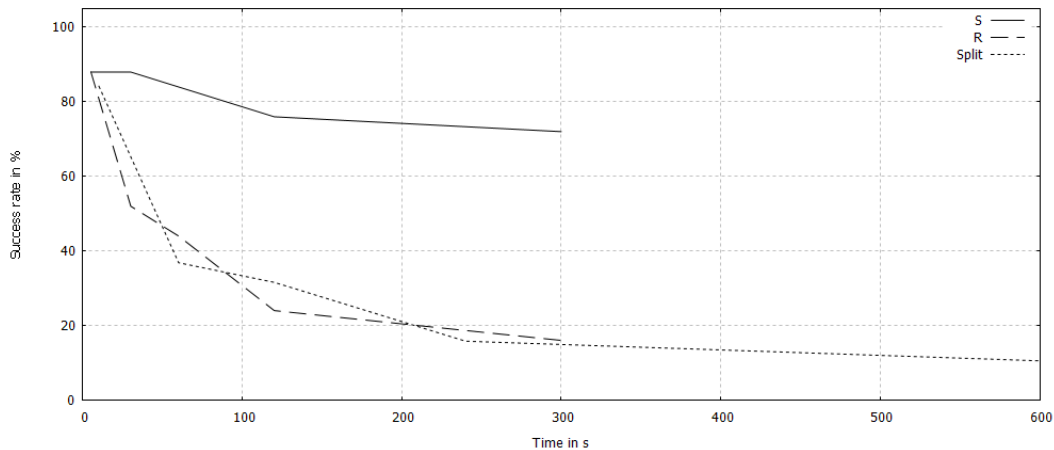


FIGURE 6.2: Delayed WebRTC connection setup success rate with Server as Sender (S)

The results of the first experiment are shown in [Figure 6.2](#). The server took the role of Sender in all connections in this experiment.

The size of the dataset is as follows: test sets 1 and 2 (S & R) consists of 25 connections. For test set 3 (Split delay), only 19 connections completed all tests. This is after cleaning the dataset. Connections that failed all tests were removed because the probability of failing all tests and still being a supported connection is very low.

These connections are assumed not to be in the target group of computers eligible to use SendIt, and as such would skew the results.

Now let's discuss the findings and implications of this experiment (See [Figure 6.2](#)). As you can see, *S* is fairly irrelevant. Even at 300 seconds (5 minutes) the success rate is 72%. *R* is much more important. At a 5 seconds delay, *S* and *R* have the same success rate. However, at 30 seconds, there is a gap of 36% ($S=88\%$, $R=52\%$). This indicates that the problem with keeping a connection open largely resides on the Receiver's end. Splitting the delay equally between *S* and *R* (as indicated by 'Split Delay'), seem to improve connectivity compared to *S*, which corroborates the theory that *S* has more influence on the success rate of creating a connection.

One factor that might influence this result is that, in this experiment, the test-server always acted as *S*. The server may have a more stable internet-connection than most endpoints. As such, another experiment was conducted where the server acts as *R*, so the results could be compared and the conclusion re-evaluated.

6.3.2 Experiment 2

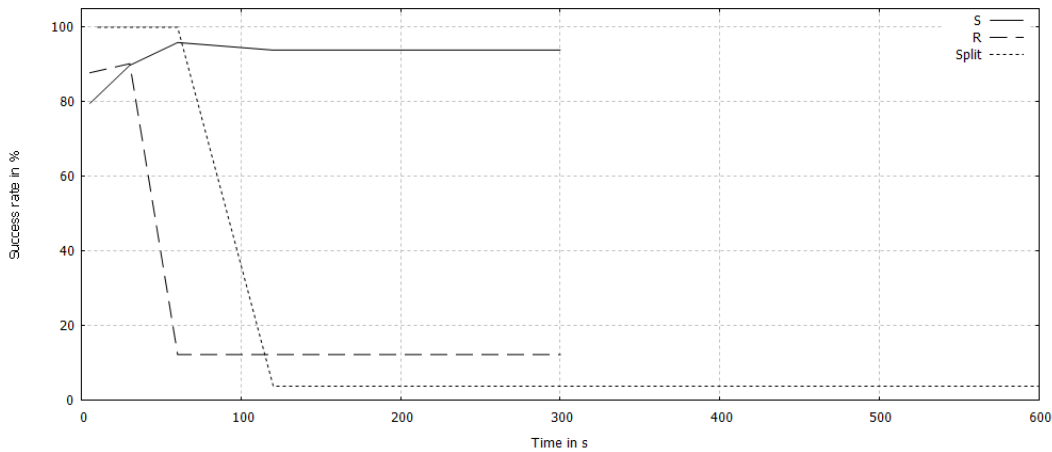


FIGURE 6.3: Delayed WebRTC connection setup success rate with Server as Receiver (*R*)

The results of the second experiment are shown in [Figure 6.3](#). The server took the role of Receiver in all connections in this experiment.

The size of the dataset is as follows: test set 1 (*S*) consists of 49 connections. Test set 2 (*R*) has 41 connections. Finally, for test set 3 (Split delay), there are 27 connections. This is after cleaning the dataset. Connections that failed all tests were removed because the probability of failing all tests and still being a supported connection is very low. These connections are assumed not to be in the target group of computers eligible to use SendIt, and as such would skew the results.

Because of some technical issues during the second experiment, a lot of connections timed out after only completed a few tests. As such, connections who did not complete the first test set (*S*) were also excluded. During analysis of the results, it also became clear that there were a number of repeating connections from the same client, with identical results, in a very short time span. As such, results from the same IP with identical results, with less than one hour between tests, were removed. This resulted in a more correct representation of the data gathered.

TABLE 6.1: Comparison of the Sender's delay in Experiments 1 and 2. The values represent the rate of success in establishing a connection.

S	Time in seconds				
	5	30	60	120	300
Experiment 1	88%	88%	84%	76%	72%
Experiment 2	80%	90%	96%	94%	94%

Now let's discuss the findings and implications of this experiment (See [Figure 6.3](#)). In a similar fashion as in Experiment 1, S is much less impactful than R . In this experiment, R has an even more drastic and rapid decrease in success-rate. Already after 60 seconds, the rate has dropped to only 12%. S on the other hand, experiences very little variation in success-rate over time. It is safe to say that this experiment indicates that S is insignificant in comparison to R . It is also interesting to note that the Split delay has an even more severe drop-off than R . Between 60 and 120 seconds, it drops from 100% success-rate to only 4%. Why this is the case is hard to evaluate, as there is no clear indication why this behavior occurs.

6.4 Impact and findings

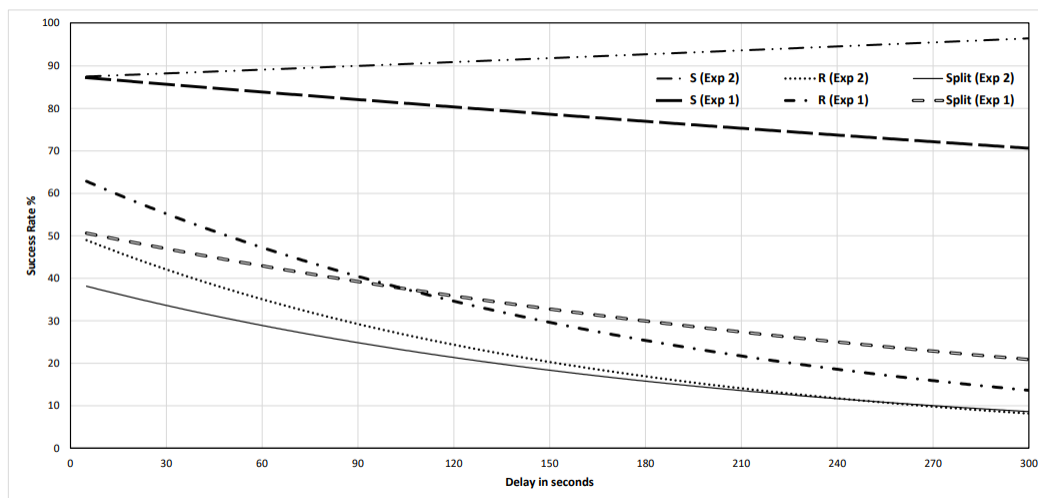


FIGURE 6.4: Shows the exponential approximation of the results in the two experiments for comparison. These indicate how the average results will look over time.

The results from these experiments can now be compared (Shown in [Table 6.1](#), [Table 6.2](#), [Table 6.3](#) and [Figure 6.4](#)), by examining the collected data.

As you can see, the results are fairly similar. This indicates that delaying the Answer has far more impact than delaying the Offer. One can assume this by comparing the success-rate of connections for R and S in [Table 6.1](#) and [Table 6.2](#). The percentages drop drastically, as the delay increases for R . On the contrary, S has only a slight decrease in success-rate (and only for Experiment 1). From this one can conclude that the original assumption was correct: The Answer is by far the most impactful part of the connection setup.

TABLE 6.2: Comparison of the Receiver's delay in Experiments 1 and 2. The values represent the rate of success in establishing a connection.

R	Time in seconds				
	5	30	60	120	300
<i>Experiment 1</i>	88%	52%	44%	24%	16%
<i>Experiment 2</i>	90%	93%	12%	12%	12%

TABLE 6.3: Comparison of the Split delay in Experiments 1 and 2. The values represent the rate of success in establishing a connection.

Split delay	Time in seconds				
	10	60	120	300	600
<i>Experiment 1</i>	~88%	~37%	~32%	~16%	~11%
<i>Experiment 2</i>	100%	100%	~4%	~4%	~4%

By looking at and comparing the results from Experiments 1 and 2, one can clearly identify that *R* experiences a significant drop when the delay is between 30 and 60 seconds. With a success-rate of respectively 52% and 93% for 30 second delay, compared to 44% and 12% for a 60 second delay. With this knowledge, one can conclude that the average lifetime of *R* is somewhere between these two values.

One can also draw the conclusion that the Offer can last at least 5 minutes, based on the experiments. The lowest success-rate for the Offer was 72% at 300 seconds (5 minutes) delay, in Experiment 1. For Experiment 2, this value was comparatively 94%. From this it can be extrapolated that it is likely that a connection will be successful, even if *S* approaches, or even extends past 5 minutes.

By combining the results of two previous conclusions: *S* having a lifetime of at least 5 minutes, and *R* having a lifetime between 30 and 60 seconds, the average lifetime of the connection setup can be found. Even though *R* experiences a rapid decrease between 30-60 seconds, *S* seems to have the ability to extend past 5 minutes. For this reason, the lifetime of the connection setup will be around 6 minutes. This should be kept in mind if one is utilizing the Serverless mode.

Finally, an interesting observation, however expected, is that when the Server acted as the Receiver (Experiment 2), the results were far more volatile, then when the roles were reversed (Experiment 1). This is probably because it is beneficial to have the most stable endpoint initiate the connection, as it allows for more predictable results.

The more stable connection is likely to have more routes to be reached through, and as such, is more likely to result in a more stable setup environment. If the route changes, with the more stable endpoint being the initiator, one of the other routes can be used instead. This is not the case for unstable, constantly changing endpoints, as they may only have one route, or the other routes also become unusable. It should be emphasized that there is no proof to back up such an assumption in these experiments, but it seems to be the likely scenario.

Chapter 7

System evaluation

To understand the contribution and usability of SendIt, an analysis of existing solutions is needed. By comparing SendIt with existing solutions it is possible to get an idea of what has been improved, what stayed the same, and what still needs to be improved. In the following section the e-mail system, a generic SNS system and SendIt will all be evaluated and analyzed.

It will also discuss and compare SendIt to other solutions, and evaluate how SendIt matches up against these solutions.

7.1 Evaluation of SendIt

SendIt has fewer weaknesses than the other two systems. The main advantage is the direct transfer, which minimizes the attack surface compared to the other two solutions. This evaluation will not include all attacks made possible by the technology chosen, but rather focus on the weaknesses that come as a consequence of the proposed system. By this it is meant that attacks that are present in all JavaScript solutions will not be discussed. If the attack is possible because of the way JavaScript is used in this solution, it will be included.

7.1.1 First trust

The main weakness, as mentioned previously, is the first trust. This can allow an attacker to appear as the legitimate owner of an identity, without this being the case. This is a risk that any user should be aware of, as it is a critical part of the system. Even though it is a part of the design choice, it has to be acknowledged that this is less than optimal.

The mitigation for such attacks is split in two parts. The first is relying on the user to evaluate whether the user is legitimate. By using social cues like timing, each user should evaluate if it is likely that the other identity would initiate a transfer. One can also contact the other endpoint and confirm that it is actually them. This is, of course, only necessary for the first connection.

The second is the trust model, which is an automatic part of the system. It combats this issue by sharing information and allowing users to have as much data as possible, for all identities. This allows for detection of false identities and will eventually make these identities untrusted entities in the whole system.

7.1.2 DHT

While P2P transfer is direct communication between two endpoints, it needs some way to set up the connection. Traditionally this has been done using servers or DHT networks. Since the whole point is to avoid using servers, the only option left is

DHT. DHT solutions rely on connecting to specific endpoints, called bootstrapping, to enter the network. This is predictable behavior that can be exploited by attackers, and it also increases the difficulty of using the system.

As such, the system suggested in this thesis leaves it up to the end users to decide how to share the information, as it is unpredictable and allows for easier usage of the system. It removes the need for any bootstrapping or server, and does not specify any set way to share the connection information.

7.1.3 ACS

Another attack vector is the ACS server. If it is not trusted, it can pose a security risk, as the endpoint has to rely on the fact that the ACS server relays the data to the correct recipient. If the data sent is encrypted, it will not pose any immediate risk, but it will stop the endpoint from connecting to anyone, as the other endpoint will not get authenticated properly. If it is the first time these endpoints are connecting, however, it puts the victim at risk of being put in touch with the wrong identity.

The ACS server can also act as a logger that keeps track of which identities are communicating with each other. While not directly an attack, it can raise privacy issues, since it allows for monitoring of traffic. It can also be argued that it can be used for reconnaissance, which allows for a targeted attack on one of the endpoints. Because of this, endpoints should only use ACS servers they trust.

The ACS server can also deploy MITM-attacks during the connection setup, which leads to the attacker gaining full access to the data transferred. The biggest problem with this type of attack is that, in most cases, it cannot be detected until after the data has already been shared and the attacker has full access to it. It will eventually be detected by the trust system, but at that point it might already be too late. While there is also a possibility for this to happen during the connection setup of the Serverless mode, it is much harder to do, since there is no pre-determined channel for sharing the Offer/Answer, and as such, the attacker has no set point to attack.

7.1.4 Other

Finally, there is a theoretical possibility of having XSS attacks executed in the client, since the Serverless mode relies on user input. While there are countermeasures deployed, one can never guarantee that there are no such security holes.

7.2 E-mail system comparison

Comparing the e-mail system to SendIt is a little like comparing your mailbox to a safe. While the mailbox can be locked and it can be made more secure, it is still outside your house, with no access control, not made to withstand attacks, and ultimately, it can just be stolen as a whole. In contrast, the safe is made to withstand attacks, and store your data privately.

In a similar fashion, e-mail was made to send and receive information, like a mailbox. SendIt is made to make this process secure against outside tampering and allow for minimal risk while exchanging information. To continue the analogy, it is like meeting the Sender with your safe, giving the Sender a small slot to put the mail in, that only fits the letter you expect to receive, and immediately returning back to the safety of your house. SendIt would be the safe in the aforementioned scenario. To summarize, the email system offers some secure solutions, but guarantees none.

In practice, that means one should assume that none is applied. Therefore, anything is an improvement.

7.2.1 Direct transfer

The first major advantage SendIt has over traditional e-mails, is that it transfers the file directly. None of the issues stemming from files being stored in transit applies to SendIt. These issues range from attackers hacking the server and getting access to the files, to allowing attackers to monitor data going through the server, since the data will always pass through this point. As a general rule, a resource is more secure, the fewer copies exist, since it limits the attack surface.

Another point to take into consideration is that even if only encrypted data is stored on a server, if an attacker gets access to the data, it is only a question of how long it takes to break the encryption. As such, minimizing the risk of leaking any data should be considered critical. This is why direct transfer of files offer such an advantage.

The attack surface which is exposed is also important to evaluate. Having a server store the files, means the attack surface is quite large, since it is always online. With SendIt's direct transfer function, the file is only available during transfer. The time difference for how long the file is available to be attacked is huge. As such, this is another advantage of direct transfer, since the attack surface is significantly reduced.

7.2.2 Authentication

Sender and Receiver guarantee is another advantage of SendIt. In the e-mail system, the Sender's identity can be forged easily, which is not the case for SendIt. It also has some issues where the data sent is not kept confidential and can easily be stolen by a passive listening attack. These two combined make phishing attacks, as well as targeted exploits, easier to create, since you know what data or information the endpoint is seeking or expecting. By sending this data from a forged address, the chances of successful attacks increase. The e-mail system has S/MIME which allows for authentication, but as discussed in [Section 1.3.1](#), it is rarely used and is not user friendly.

7.2.3 Content control

In the ordinary e-mail system, one is also not able to stop unwanted content. If someone sends an e-mail attachment, it will be delivered to the receiver's PC, or at least their e-mail server. This is unfortunate, since receiving malicious files may cause them to be executed at a later time, or by an unintended action. It is also possible that they may execute without the user knowingly doing so. As such, it is optimal for the user to be able to stop the transfer of unwanted files, something which SendIt supports. Automatic spreading of malicious files is a big problem today. It is spread to all the victim's contacts and keeps spreading in a similar fashion. While SendIt has no direct solution to mitigate this, the fact that it does not support multicast, and that the files cannot be automatically received, will naturally take care of resolving this issue.

7.2.4 Connection setup

The disadvantage of SendIt is that it is required to complete a connection setup phase in order to communicate with the other endpoint. The e-mail system has no such need and one can easily transfer information at any time without the user having to worry about the connection setup. This is because all communication occurs between the client and a server that is publicly available.

7.2.5 Synchronous connection

Another case where SendIt exhibits different behavior is during the actual transfer. While the advantages of direct transfers have been clearly indicated, SendIt does lack support for asynchronous transfers. This is a point where the regular e-mail solution has the edge over SendIt. It can be argued, however, that it is a small obstacle to overcome, for such an increase in security.

7.2.6 First trust

Finally, the e-mail system does not have to rely on the trusted first setup, that SendIt does. In the case that the e-mail system uses the security features available, it relies on PKI for the initial trust. This choice of a central service makes sense in the e-mail system which relies on servers, but not for SendIt.

7.3 SNS and Cloud systems comparison

While there is no one SNS or cloud based system to compare against, they all share certain functionality and design choices. As such, this generalization will be the model used for comparison. As discussed in [Section 7.2.1](#), SendIt directly transfers the file, removing the need for central storage, thereby lowering the attack surface. While the original point is made in comparison to the e-mail system, this applies equally to SNS and Cloud based systems. This is why SendIt provides a clear use case and improvement to current systems.

7.3.1 False E2E encryption

As mentioned in [Section 1.3.2](#) and shown in [Figure 1.1](#), false E2E encryption is sometimes utilized in these systems. This is a scary thing, as a consumer of these services, because both you and your communication partner are unable to verify if the communication has true E2E encryption or not. Effectively, this is a MITM-attack by design. As such, when using these services, one should assume that the data can be decrypted and read by the service and whoever they disclose the data to. For SendIt, the public-key cryptography guarantees end-to-end encryption, since the keys are exchanged over P2P, which means no central service can tamper with the keys sent.

7.3.2 Identity swapping

Another thing that is hard to do in these systems is swap identities, if the user desires. One may not want to associate an identity with both work and private life, which means the need for several identities arises. This requires logging in and out, managing two sets of credentials, and keeping track of which identity is currently in use. For SendIt, all the user has to do is swap to the file containing the correct keys and the identity has been changed.

7.3.3 Other

The points where SNS and Cloud based systems have clear advantages also resembles that of the e-mail system. The points made in [Section 7.2.4](#) and [Section 7.2.6](#) apply to these systems as well. For the first trust in these systems, they also rely on the PKI model, which is understandable, since they are also based around the server-client model.

These solutions do have the advantage of storing a backup of your files, which can be desired in certain situations. Though this can easily be done after the transfer has completed, which gives the user more control over how it is stored. As such, these solutions are more suited for regular files, not files with high requirements in regard to privacy and security.

Chapter 8

Conclusion

This thesis has given an introduction to the current solutions available for file transfers, and reviewed the advantages and pitfalls of these solutions. Afterwards, the basic technology and principles of SendIt were explained. Then the general design and implementations were explained. After which it went through the specifics of each mode. Following, was a review of the experiments done and their contributions to the system, and an evaluation of SendIt and comparable systems. Finally, this chapter will summarize the work done and explore directions for future research.

8.1 Summary

This thesis has introduced a system and a prototype for improving the current situation of file transfers, focusing on e-mail attachments specifically. It did so by finding technology that fit well with the requirements, and by developing an alternative solution. In an effort to return to the original idea behind the Internet as a decentralized system, this thesis suggests a serverless system and shows a prototype of an implementation of such a system.

A serverless implementation also follows the logical idea of file transfers as a transaction only between two end-nodes. There is no reason to involve a third party, unless absolutely necessary. In addition, it also reduces the cost of running services, since central servers do not need to be hosted or managed. The disadvantages of a serverless implementation is that there is no central management where one can access information easily, nor a central entity who can co-ordinate communication.

Since P2P technology allows us to avoid including a third party, this becomes the natural choice to use for the actual transfer of the files. While the usual way of creating P2P connections is by getting information from a server, or through DHT, SendIt suggests leaving it up to the users, since it will make the system more unpredictable, and as such harder to attack, while also allowing for easier use.

Since this creates additional burden on the user, the Assisted Connection Setup mode was also developed. Endpoints send connection setup information to a server over secure WebSockets, which then relays it to the correct endpoint. While this does use a server, the server acts solely as a forwarding agent. Users can either use the default server, or host their own. An example of the default server will be available with the code and installation files for SendIt. This is mainly to cater to the inexperienced users who care more about usability than maximizing security. The opposite is true for the completely serverless version.

An added benefit, and another reason for choosing P2P communication, was that it minimizes the attack surface of the application. Restricting the time of which

the file, or metadata about the file, is available online greatly reduces the risk of an attacker gaining access to it.

The contrast from the proposed system compared to regular solutions is enormous considering the fact that files are usually left on servers even after the transfer has completed, which leaves the information accessible for infinitely longer than with a direct solution. As such, the suggested system is a clear improvement, and has distinct advantages when it comes to reducing the attack surface of file transfers. The drawback of P2P is also the inherent advantage: It is synchronous communication. This means that it is up to developers to find ways to facilitate asynchronous transfers by other means, if such functionality is wanted.

When choosing how to implement these features, WebRTC came forward as the logical option, because of its inherent support for many of the desired features. It offers serverless connection setup, creates P2P connections, and also incorporates security functionality. It offers authentication of peers, based on the Offer and Answer, and automatically encrypts the communications channel. It is also easy to deploy because it is a web technology, which means it can be used in an internet browser, making it easy to deploy and install.

There are two drawbacks of using WebRTC. The first is that it offers no consistent way to reach or address endpoints, which means the connection setup has to be repeated every time a connection is to be made. The second is that there is no way to consistently identify or authenticate users. For the first issue, there are no easy remedies, the second can be solved by adding an additional identification and authentication scheme.

For such schemes, there are many options available. The most used scheme is public-key cryptography, and for good reason. As outlined in [Section 2.1](#), it is a system for encryption and decryption that ensures confidentiality and integrity, while also being easy to manage. By associating each identity with a public key, the system guarantees that only the entity with the corresponding private key can access the information. This protects user privacy, as well as, making sure that it is not possible for attackers to see or manipulate the contents of the data.

The downside of public-key cryptography is that it is hard to understand and manage correctly by people who are less technically capable. They have a hard time understanding the concept behind how it works, the difference between the two keys, and how to utilize it in practice. In the proposed system, this is taken care of programmatically and the users do not have to worry about such issues.

It is important to use the correct key for the corresponding endpoint, in order for the encryption and decryption scheme to be useful. This is why identity management and trust is necessary. To be able to use public-key cryptography, users need to exchange keys and, only then, can data be encrypted.

The exchange of keys is usually done through a trusted third party, in the form of a server. Since SendIt does not have a central server to rely on, the first connection, and with it the exchange of keys, is regarded as a trusted interaction. What this means is that the system assumes the information is correct and not being supplied by a malicious user. Afterwards, the system can guarantee all the benefits of public-key cryptography, but it relies on this first setup being correct. In order to reduce the risk of an attacker exploiting this, a trust system is suggested.

The web of trust model allows users to re-evaluate and nuance their trust in endpoints. This gives a collective view of who is trustworthy and who is not, and

can also help detect attackers trying to abuse the first trust. While this initial trust is detrimental to the security of the system, changing it would affect usability severely. As such, the system is built upon this first trust, with the web of trust model as a continuous evaluation to detect attackers, and to minimize the effect of potential attackers. It is possible to extend or change this functionality, if one so pleases, when using SendIt as a platform for further development.

These features create the basis for a reasonably secure system. In addition, the system is made to be as user friendly as possible. Installation is extremely easy, all that is required is downloading the installer and executing it. There are no options or dialogs during the installation. There is also no requirement to sign up, to do any key management, or any other aspects that may take focus away from the main goal: transferring files.

The user only has to care about placing the files in the correct folder and initiating the transfer to the correct destination. In a similar fashion, when receiving files, all one has to do is decide whether to accept or decline such an offer. The functionality is clear and easy to use. The flow of the program is easy to follow, and requires little interaction from the user.

Once the transfer is complete, the connection is automatically torn down and the user can start a new transfer. All of these features are there to make sure that people with low technical understanding can still get the benefit of using the system. The difficult, technical parts are automatically set up and taken care of for the user. Advanced users are also kept in mind, as they are allowed access to details about the system and can make changes in the settings window.

Comparing SendIt to existing solutions clearly illustrates the advantage of direct communication and the increase this yields in regards to protecting users security and privacy. It gives the user better control over where their data is, and also reduces the chance of an attacker gaining access to the data.

Additionally, it requires no setup on the user side, and can be used directly after installation has finished. While other solutions have encryption and authentication, the systems used cannot be inspected by the user, and they can easily be tricked by false E2E encryption. This is not the case for SendIt as it can be easily inspected and managed. One can also easily switch identities, if desired, without having to create a new account or go through additional steps. This is in stark contrast to the comparative solutions.

The contributions of this thesis can be summed up as a user-friendly system that allows for direct, secure file transfers. SendIt can also serve as a platform to expand for a wide range of functionality and services. SendIt, as a platform, would likely be used for it's identity management and authentication functionality, not the file transfer functionality. While this is one potential use case, another can be adding more functionality to the file transfer aspect. This would allow it to become more of a complete solution that could function as a stand alone system, instead of a complementary one.

It also contributes to exploring new implementations of e-mail attachments. The serverless functionality also opens for a new way of software developing, and contributes and encourages a break with the current server-based development. Finally, it shows that secure solutions do not necessarily have to be hard to use and incentivizes development of more user-friendly security applications.

In conclusion: This thesis introduced a system that improves the current implementation of e-mail attachments. It does so by directly transferring files, lowering the attack surface significantly. It also guarantees end-to-end encryption and end-point authentication (*based on the key exchange done as part of the first connection*), which is not commonly used for e-mail attachments, making it resistant to both active and passive attacks. It achieves this while keeping the system easy to use, by taking care of key management and authentication on behalf of the user, without the need for any setup or central corroboration. SendIt outperforms e-mail attachments in regards to security, as well as usability, and cost efficiency. The drawbacks of SendIt are the limitation of only supporting synchronous transfers, as well as, it's inherent trust in the first connection.

8.2 Future work

There are still many venues and ways of improvement for systems such as SendIt, and e-mail attachments in general. By separating future work into two sections, it is easier to get a clear view of which improvements are necessary for SendIt, and what changes are needed in regards to e-mail attachments in general.

SendIt has the core features and design developed, but since it has been a one person job, the depth and complexity has been limited. With more time or more manpower, many of the issues currently existing can be fixed, and a complete alternative to the current e-mail attachment functionality can be finalized. As such, the future research for SendIt focuses more on implementations and testing of the system.

For e-mail attachments in general, a more holistic approach is taken in regards to future work. Things like large scale evaluation and system design become more important and need to be prioritized more than when designing a simpler system.

8.2.1 SendIt

The following topics can be future research for the development for SendIt:

- **Developing and testing extendability of the connection setup for the Serverless mode.** Adding to the lifetime of the connection setup would greatly increase the usability and reliability of this mode. Adding some kind of connection-manager or middleware was considered as part of this research, but could not be done in time.
- **Improving reliability of connections and connection setup.** In conjunction with the point above, increasing the success rate of the connection setup would make the system more reliable and easier to use. Making sure connections are stable and do not disconnect will help the usability and stability of the system.
- **Exploring more ways to evaluate trust.** Testing, implementing, and comparing different systems in practice, as well as finding an algorithm for evaluation. This work would add great value to the trust management of SendIt.
- **Improve the initial trust.** Finding a way to increase the trust in the first connection, or some way of authentication that does not rely on a central service, would be a big improvement. In addition, allowing for sharing keys across applications may also be a way to bridge this initial trust issue. Such problems are left up to future research.

- **Adding support for pausing transfers.** Having a way to pause transfers should be easy to implement, and will benefit the system greatly. Especially for big transfers, this will allow users to take breaks temporarily, if needed.
- **Checking overhead of the file transfer and optimizing the protocol.** Optimizing the speed of the file transfer, as well as, minimizing the overhead of the protocol, will ensure that the communication is as effective as possible. This would allow for shorter transfer-times, less data transferred, and higher efficiency.

8.2.2 E-mail attachments

The following topics can be future research for e-mail attachments and how to improve the current system:

- **Finding a better way to asynchronously transfer files.** Any improvement to the way e-mail attachments currently work would be a welcome contribution, but particularly an improvement to user privacy is necessary. Avoiding server-storage would be optimal, but finding a way to ensure that the file does not reside on the server after the message has been transferred would be an acceptable compromise.
- **Exploring other systems and alternatives to SendIt.** Evaluating their usability and functionality. By doing this, one can clearly indicate which solution is optimal, and recommend a new standard which can be adopted by e-mail clients.
- **Creating alternative solutions to complement SendIt.** As mentioned in the introduction ([Chapter 1](#)), replacing the current email attachment system with SendIt alone is not recommended. Developing a complementary system to SendIt, which in combination can be implemented to completely replace the current system, would go a long way towards improving the current situation.

8.3 Final remarks

As a final note, I would like to add that the source code for both SendIt and the ACS server will be made available [on github](#)¹ once it has been cleared for release. I hope it will be developed further, and that it can succeed as an open source application that provides improved security and privacy for everyone.

¹<https://github.com/Robiq>

Appendix A

ACS Server

LISTING A.1: ACS Server

```

1  const HTTPS_PORT = 7443;
2
3  const fs = require('fs');
4  const https = require('https');
5  const WebSocket = require('ws');
6  const WebSocketServer = WebSocket.Server;
7  const q = require('./resources/queue.js');
8  const crypto = require('@trust/webcrypto');
9  var uuid=1;
10 var sockuuid=1;
11 var sUUID = 0;
12 var conn = {};
13 var key = new Object();
14
15 var wss_prot = {
16   //Authentication
17   AUTH_SETUP: "setup", //Register keys
18   AUTH_S_REPLY: "setup_reply", //Registration OK/NOT OK
19   AUTH_INIT: "auth_init", //Start authentication
20   AUTH_RESULT: "result", //Authentication result
21
22   //All directions
23   ERROR: "error", //error occurred - specified in data
24   WAIT: "wait", //Waiting for other end - specified in data
25
26   //Client-Server messages
27   INIT: "start", //Offer sending - forwarded
28   DONE: "done", //Connection finished
29   REQKEY: "request_key", //Request public key - Include own public
        key!
30
31   //Server-Client messages
32   KEY: "key", //Share keys for other end
33
34   //Client-Client messages
35   ACCEPT: "accept", //Accept offer - contains offer!
36   REFUSE: "refuse", //Refuse offer
37   ANSWER: "answer", //Contains answer
38   ICE: "ice", //Contains ICE-candidates
39   LOOKUP: "lookup" //For looking up email presence
40 };
41
42 //console.log("Argv1: " + process.argv[2]);
43
44 // Yes, SSL is required
45 const serverConfig = {
46   key: fs.readFileSync('server-key.pem'),

```

```

47     cert: fs.readFileSync('server-cert.pem'),
48   };
49
50   start();
51
52   // Create a server for the client html page
53   var handleRequest = function(request, response) {
54     // Render the single client html file for any request the HTTP
55     // server receives
56     console.log('request received(https): ' + request.url);
57     try{
58       //If testing, serve files
59       if (process.argv[2] === 'test'){
60         if(request.url === '/') {
61           response.writeHead(200, {'Content-Type': 'text/html'
62           });
63           response.end(fs.readFileSync('/home/robin/Project/
64           Server/test/wss_test.html'));
65         } else if(request.url === '/wss_test.js') {
66           response.writeHead(200, {'Content-Type': 'application/
67           javascript'});
68           response.end(fs.readFileSync('/home/robin/Project/
69           Server/test/wss_test.js'));
70         }else{
71           console.log('Invalid URL requested - no HTML support!')
72           ;
73           response.writeHead(404);
74           response.end();
75         }
76       }
77       //If production no files!
78       }else{
79         console.log('Invalid URL requested - no HTML support!');
80         response.writeHead(404);
81         response.end();
82       }
83     }catch(e){
84       console.log("Exception when serving file(https): ", e);
85     }
86   };
87
88   //Read in keypair or create one if first run.
89   function start(){
90     try{
91       var buf = fs.readFileSync("./keys.crp", "utf8");
92       var dec = buf.split(";\\n");
93
94       //console.log(buf);
95       key.privateKey = JSON.parse(dec[0]);
96       key.publicKey = JSON.parse(dec[1]);
97
98       //create list of conn!
99       for (var i = 2; i < dec.length-1; i++) {
100         console.log(dec[i]);
101         var tmp = dec[i].split(";");
102         conn[tmp[0]] = new Object();
103         conn[tmp[0]].key = JSON.parse(tmp[1]);
104       }
105
106       key.expkey = key.publicKey;
107
108       Promise.all([
109         (

```

```

104     crypto.subtle.importKey(
105         "jwk", //can be "jwk" (public or private), "spki" (public
            only), or "pkcs8" (private only)
106         key.privateKey,
107         { //these are the algorithm options
108             name: "RSA-OAEP",
109             hash: {name: "SHA-1"}, //can be "SHA-1", "SHA-256", "
                SHA-384", or "SHA-512"
110         },
111         true, //whether the key is extractable (i.e. can be used
            in exportKey)
112         ["decrypt", "unwrapKey"]//["wrapKey", "unwrapKey"]
113     )
114 },
115 (
116     crypto.subtle.importKey(
117         "jwk", //can be "jwk" (public or private), "spki" (public
            only), or "pkcs8" (private only)
118         key.publicKey,
119         { //these are the algorithm options
120             name: "RSA-OAEP",
121             hash: {name: "SHA-1"}, //can be "SHA-1", "SHA-256", "
                SHA-384", or "SHA-512"
122         },
123         true, //whether the key is extractable (i.e. can be used
            in exportKey)
124         ["encrypt", "wrapKey"]//["wrapKey", "unwrapKey"]
125     )
126 )
127 ])
128 .then(function (keys) {
129     console.log("Keys imported!");
130     key.privateKey = keys[0];
131     key.publicKey = keys[1];
132 });
133 }catch(err){
134     console.log("Error reading cryptofile: ", err);
135     //create keys
136     crypto.subtle.generateKey(
137         {
138             name: "RSA-OAEP",
139             modulusLength: 2048, //can be 1024, 2048, or 4096
140             publicExponent: new Uint8Array([0x01, 0x00, 0x01]),
141             hash: {name: "SHA-1"}, //can be "SHA-1", "SHA-256", "
                SHA-384", or "SHA-512"
142         },
143         true, //whether the key is extractable (i.e. can be used in
            exportKey)
144         ["encrypt", "decrypt", "wrapKey", "unwrapKey"]
145         //["encrypt", "decrypt"] //must be ["encrypt", "decrypt"] or
            ["wrapKey", "unwrapKey"]
146     )
147     .then(function(keys){
148         console.log("Keys created!");
149         key=keys;
150         return crypto.subtle.exportKey(
151             "jwk", //can be "jwk" (public or private), "spki" (public
                only), or "pkcs8" (private only)
152             key.publicKey //can be a publicKey or privateKey, as long
                as extractable was true
153         )
154     })
155     .then(function(expkey){

```

```

156     key.expkey=expkey;
157 }
158 .catch(function(err){
159     console.error(err);
160 });
161 }
162
163 //console.log("Keys", key);
164
165 }
166 //

```

```

167
168 var httpsServer = https.createServer(serverConfig, handleRequest);
169 httpsServer.listen(HTTPS_PORT, '0.0.0.0');
170
171 // Create a server for handling websocket calls
172 var wss = new WebSocketServer({server: httpsServer});
173
174 wss.on('connection', function(ws) {
175     ws.id=sockuuid++;
176     console.log("Client %d connected!", ws.id);
177
178     //Message received in server!
179     ws.onmessage = function(message) {
180         console.log('received message from id: ', ws.id);
181         //Handle message!
182         handleMessage(ws, message.data);
183     };
184
185     ws.onclose = function(){
186         console.log("Socket closed! Client id nr. %d disconnected!",
187             ws.id);
188     };
189
190     ws.onerror = function(err){
191         console.log("Error occurred: ", err);
192     };
193 });
194
195 function handleMessage(sock, msg) {
196     console.log(msg);
197     msg = JSON.parse(msg);
198     switch(msg.prot){
199         case wss_prot.LOOKUP:
200             console.log("Received mail: ", msg.origin);
201             //console.log(key.publicKey);
202
203
204             if(msg.origin in conn){
205                 console.log("Found mail ", msg.origin);
206                 var thiscon = conn[msg.origin];
207                 //Check queue for waiting connections!
208                 //checkQueue(thiscon);
209
210                 //create session key!
211                 createSymmkey(sock, wss_prot.LOOKUP, thiscon, key.expkey);
212                 //send(sock, wss_prot.LOOKUP, {res: true, symkey:
213                     key.symkey});
214             }else{
215                 console.log("Did not find mail ", msg.origin);

```



```
215         send(sock, wss_prot.LOOKUP, {res: false, key: key.expkey});
216     }
217     break;
218
219     case wss_prot.AUTH_SETUP:
220         console.log("Protocol received: Authentication setup");
221         auth_S_Reply(sock, msg);
222         break;
223
224     case wss_prot.AUTH_INIT:
225         console.log("Protocol received: Authentication Initiation");
226         console.log(msg);
227         auth_result(sock, msg);
228         break;
229
230     case wss_prot.ERROR:
231         console.log("Protocol received: Error! Details: ", msg.data);
232         if(msg.destination != null){
233             forward(sock, msg);
234         }
235         break;
236
237     case wss_prot.WAIT:
238         console.log("Protocol received: Wait");
239         if(msg.destination != null){
240             forward(sock, msg);
241         }
242         break;
243
244     case wss_prot.INIT:
245         console.log("Protocol received: Initialize connection");
246         forward(sock, msg);
247         break;
248
249     case wss_prot.DONE:
250         console.log("Protocol received: Done");
251         //Check the queue!
252         //var thiscon = conn[msg.origin];
253         //checkQueue();
254         break;
255
256     case wss_prot.REQKEY:
257         console.log("Protocol received: Request Key! Looking for key
258             for: ", msg.data);
259         //todo fix!
260         if(msg.data in conn){
261             console.log("Found key for ", msg.data);
262             var x = msg.data;
263             send(sock, wss_prot.KEY, {x: (conn[msg.data]).key });
264         }else{
265             console.log("No key found for ", msg.data);
266             send(sock, wss_prot.KEY);
267         }
268         break;
269
270     case wss_prot.ACCEPT:
271         console.log("Protocol received: Accept");
272         forward(sock, msg);
273         break;
274
275     case wss_prot.REFUSE:
276         console.log("Protocol received: Refuse");
277         forward(sock, msg);
```

```

277         break;
278
279     case wss_prot.ANSWER:
280         console.log("Protocol received: Answer");
281         forward(sock, msg);
282         break;
283
284     case wss_prot.ICE:
285         console.log("Protocol received: ICE");
286         forward(sock, msg);
287         break;
288
289     case wss_prot.AUTH_S_REPLY:
290         console.log("Protocol received: Authentication setup reply\
291             nERROR! Not supposed to be in server!");
292         break;
293     case wss_prot.AUTH_RESULT:
294         console.log("Protocol received: Authentication Result\nERROR!
295             Not supposed to be in server!");
296         break;
297     case wss_prot.KEY:
298         console.log("Protocol received: Key\nERROR! Not supposed to
299             be in server!");
300         break;
301
302     default:
303         console.log("Unknown message: ", msg);
304         break;
305 }
306
307 //Return setup result
308 function auth_S_Reply(sock, msg){
309     console.log("Org mail: ", msg.origin)
310     //reply true or false - evaluate!
311     var auth=true;
312     //Check if key already associated with email
313     for (var ent in conn){
314         //console.log(conn[key]);
315         if(conn[ent].key === msg.data){
316             auth=false;
317             console.log("Key already exists for another email: %s!",
318                 conn[key].id);
319         }
320     }
321
322     if(msg.origin in conn){
323         auth = false;
324         console.log("Authentication error! E-mail already registered!
325             ")
326     }
327
328     if(auth){
329         conn[msg.origin]= new Object();
330         conn[msg.origin].key = msg.data;
331         conn[msg.origin].sock = sock;
332         conn[msg.origin].id = uuid++;
333         var thiscon = conn[msg.origin];
334
335         console.log(conn);
336         //Share symkey!
337         createSymmkey(sock, wss_prot.AUTH_S_REPLY, thiscon);
338     }else{

```

```

335         //Return false
336         send(sock, wss_prot.AUTH_S_REPLY, {res: auth}, msg.origin);
337     }
338 }
339
340 //Return authentication result
341 async function auth_result(sock, data){
342     console.log("Conn: ", conn);
343     console.log("Data: ", data.origin)
344     //data=JSON.parse(data);
345     if(data.origin in conn){
346         if(await isAuth(data)){
347             console.log("User mail %s is authenticated!", data.origin);
348             conn[data.origin].sock=sock
349             send(sock, wss_prot.AUTH_RESULT, true, data.origin);
350         }else{
351             console.log("User mail %s is not authenticated!", data.origin
352             );
353             send(sock, wss_prot.AUTH_RESULT, false, data.origin);
354             sock.close();
355             conn[data.origin].sock=null;
356         }
357     }else{
358         console.log("Details not stored for this user mail (%s) -
359         please do an authentication setup!", data.origin);
360         send(sock, wss_prot.AUTH_RESULT, false, data.origin);
361         sock.close();
362     }
363 }
364
365 //Compare and authenticate
366 async function isAuth(data){
367     //console.log("isAuth data: ", data);
368     //decrypt data
369     var decrypted = await decrypt(data);
370     console.log("Email: " + data.origin + " Decrypted: " + decrypted)
371     ;
372     if(data.origin === decrypted){
373         return true;
374     }
375     return false;
376 }
377
378 //Decrypt data and return value or empty string
379 async function decrypt(data){
380     var thiscon = conn[data.origin];
381     var decryData;
382     data=data.data;
383     decryData = Object.values(data.ciph);
384     thiscon.iv=data.iv;
385
386     return crypto.subtle.decrypt(
387     {
388         name: "AES-GCM",
389         iv: thiscon.iv,
390         //label: Uint8Array([...]) //optional
391     },
392     thiscon.symmetric,
393     //this.key.privateKey, //from generateKey or importKey above
394     new Uint8Array(decryData)//ArrayBuffer of the data
395 )
396 .then(function(decrypted){

```

```

395     //returns an ArrayBuffer containing the decrypted data
396     console.warn("Data decrypted raw: ", new Uint8Array(decrypted))
397     ;
398     decryData = new Uint8Array(decrypted);
399     decryData = convertArrayBufferViewToString(decryData);
400     console.log("Data decrypted: ", decryData);
401     return decryData;
402 }
403 .catch(function(err){
404     console.error(err);
405     return '';
406 });
407 }
408 //Convert from array buffer to string
409 function convertArrayBufferViewToString(buffer){
410     var str = "";
411     for (var iii = 0; iii < buffer.byteLength; iii++)
412     {
413         str += String.fromCharCode(buffer[iii]);
414     }
415     return str;
416 }
417 }
418
419 //Send message through socket
420 function send(sock, sig, data=null, dst=null){
421     var msg = {
422         prot: sig,
423         origin: 'server',
424         destination: dst,
425         data: data
426     }
427     msg = JSON.stringify(msg);
428     if(sock.readyState === WebSocket.OPEN) {
429         console.log("Sending to id: ", sock.id)
430         console.log("Sending: ", msg);
431         sock.send(msg);
432     }else{
433         console.log("Error sending to id; %s Mail: %s ", sock.id, dst
434             );
435         console.log("Error sending: ", msg);
436         console.log("Socket state: ", sock.readyState);
437     }
438 }
439 //Try to forward message
440 function forward(sock, msg) {
441     if (msg.destination in conn){
442         console.log("Forwarding protocol %s to %s!", msg.prot,
443             msg.destination);
444         sendFw(conn[msg.destination].sock, msg);
445     } else{
446         console.log("Destination email %s not connected!",
447             msg.destination);
448     }
449     if(msg.prot==wss_prot.INIT){
450         /*todo - add checking functionality for queued messages on
451             connect!
452         //q.add2Q(msg);
453         console.log("Waiting for %s to connect!\n Sending waiting
454             signal to %s", msg.destination, msg.origin);
455         send(sock, wss_prot.WAIT);
456         */

```

```

452     send(sock, wss_prot.REFUSE);
453 }else{
454     console.log("Connection error! ID %s went offline", sock.id);
455     send(sock, wss_prot.ERROR, 'Other end disconnected from
        server!');
456 }
457 }
458 }
459 //Send to correct destination
460 function sendFw(sock, msg){
461     msg = JSON.stringify(msg);
462     if(sock.readyState === WebSocket.OPEN) {
463         console.log("Sending to ID: %s Mail: %s", sock.id,
            msg.destination)
464         console.log("Sending: ", msg);
465         sock.send(msg);
466     }else{
467         console.log("Error sending to; ", sock.id);
468         console.log("Error sending: ", msg);
469         console.log("Socket state: ", sock.readyState);
470     }
471 }
472
473 //Create symmetric key for connection
474 function createSymmkey(sock, prot, thiscon, expkey=null){
475
476     //Create symmetric key
477     crypto.subtle.generateKey(
478         {
479             name: "AES-GCM",
480             length: 256,
481         },
482         true,
483         ["encrypt", "decrypt"]
484     )
485     .then(function(key){
486         //Encrypt with symmetric key
487         thiscon.symmetric=key;
488         return crypto.subtle.importKey(
489             "jwk", //can be "jwk" (public or private), "spki" (public
                only), or "pkcs8" (private only)
490             thiscon.key,
491             { //these are the algorithm options
492                 name: "RSA-OAEP",
493                 hash: {name: "SHA-1"}, //can be "SHA-1", "SHA-256", "
                    SHA-384", or "SHA-512"
494             },
495             true, //whether the key is extractable (i.e. can be used in
                exportKey)
496             ["encrypt", "wrapKey"])
497     })
498     .then(function(key){
499
500         console.log(thiscon.symmetric)
501         //console.log(thiscon.key)
502         //encrypt (wrap) symmetric key with server public key
503         return crypto.subtle.wrapKey(
504             "raw", //the export format, must be "raw" (only available
                sometimes)
505             thiscon.symmetric, //the key you want to wrap, must be able
                to fit in RSA-OAEP padding
506             key, //the public key with "wrapKey" usage flag
507             { //these are the wrapping key's algorithm options

```

```

508         name: "RSA-OAEP",
509         hash: {name: "SHA-1"},
510     }
511 )
512 })
513 .then(function(wrapKey){
514     //Create object for sharing: wrapped symmetric key amnd cipher
515     var wrapped = new Uint8Array(wrapKey);
516     var msg;
517     if(expkey){
518         msg = {res: true, wrap: wrapped, key: expkey};
519     }else{
520         msg = {res: true, wrap: wrapped};
521     }
522     console.log("Object", msg);
523     send(sock, prot, msg);
524 })
525 .catch(function(err){
526     console.error(err);
527 });
528 }
529 }
530
531 //Exit handling!
-----
532 process.on('SIGTERM', closeAll, 'SIGTERM');
533 process.on('SIGINT', closeAll, 'SIGINT');
534 //process.on('exit', closeAll, 'exit');
535 process.on(`uncaughtException`, closeAll, `uncaughtException`);
536
537 //Error-handling and gracious shutdown
538 async function closeAll (sig) {
539     console.log(`\nShutting down gracefully after %s :)!`, sig)
540     wss.clients.forEach(function(c) {
541         c.close();
542     });
543     await writeFile();
544     process.exit(sig);
545 };
546
547 //Write data to file
548 async function writeFile(){
549     var write;
550     //Stores the own email, then own private key, then list of know
551     //hosts and public-key-pairs.
552     await Promise.all(
553     [
554         crypto.subtle.exportKey(
555             "jwk", //can be "jwk" (public or private), "spki" (public
556             //only), or "pkcs8" (private only)
557             key.privateKey //can be a publicKey or privateKey, as long
558             //as extractable was true
559         )
560     ],
561     [
562         crypto.subtle.exportKey(
563             "jwk", //can be "jwk" (public or private), "spki" (public
564             //only), or "pkcs8" (private only)
565             key.publicKey //can be a publicKey or privateKey, as long
566             //as extractable was true
567         )
568     ]
569 )
570 .then(function (keys) {

```

```
564     write =JSON.stringify(keys[0]) + ";\n" + JSON.stringify(keys
565         [1]) + ";\n";
566     //TEST!
567     for(con in conn){
568         write = write + con + ";" +JSON.stringify(conn[con].key)+";\n";
569     }
570
571     try{
572         fs.writeFileSync("./keys.crp", write);
573     }catch(err){
574         console.error(err);
575     }
576 }).catch(function(err){
577     //Error-handling just in case
578     console.error(err);
579 });
580 }
```

Bibliography

- [1] *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the Protection of Natural Persons with Regard to the Processing of Personal Data and on the Free Movement of Such Data, and Repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA Relevance)*. May 4, 2016. URL: <http://data.europa.eu/eli/reg/2016/679/oj/eng> (visited on 03/30/2018).
- [2] Joseph Kiesel. *GDPR Compliance: Summary & Requirements You Need to Know*. URL: <https://linfordco.com/blog/gdpr-compliance-requirements/> (visited on 03/30/2018).
- [3] Author: Ron Ross (NIST) et al. *SP 800-171 Rev. 1, Protecting Controlled Unclassified Information in Nonfederal Systems and Organizations*. URL: <https://csrc.nist.gov/publications/detail/sp/800-171/rev-1/final> (visited on 03/30/2018).
- [4] L. D. Ibáñez et al. "Redecentralizing the Web with Distributed Ledgers". In: *IEEE Intelligent Systems* 32.1 (Jan. 2017), pp. 92–95. ISSN: 1541-1672. DOI: [10.1109/MIS.2017.18](https://doi.org/10.1109/MIS.2017.18).
- [5] Abdul-Rahman Alfarez. *The PGP Trust Model - Semantic Scholar*. URL: </paper/The-PGP-Trust-Model-Abdul-Rahman/e9aa5d8032c1d925ea6a02dd3be93f42e831c965> (visited on 03/26/2018).
- [6] Adam Bergkvist et al. *WebRTC 1.0: Real-Time Communication Between Browsers*. URL: <https://www.w3.org/TR/webrtc/> (visited on 03/26/2018).
- [7] Randell Jesup and Salvatore Loreto. *WebRTC Data Channels*. URL: <https://tools.ietf.org/html/draft-ietf-rtcweb-data-channel-13> (visited on 03/26/2018).
- [8] Hans-Christer Holmberg. "Web Real-Time Data Transport". Helsinki Metropolia University of Applied Sciences, Apr. 16, 2015. 55 pp. URL: https://www.theseus.fi/bitstream/handle/10024/94759/FinalThesis_hanschrh_final.pdf?sequence=1&isAllowed=y (visited on 03/23/2018).
- [9] *Firefox Test Pilot - Send*. URL: <https://testpilot.firefox.com/experiments/send/> (visited on 03/24/2018).
- [10] *A New Kind of Instant Messaging*. URL: <https://tox.chat> (visited on 05/26/2018).
- [11] *The TokTok Project - Protocol*. URL: <https://toktok.ltd/spec.html> (visited on 05/26/2018).
- [12] *I2P-Bote*. URL: <https://i2pbote.xyz/> (visited on 03/24/2018).
- [13] *I2P-Bote Introduction and Tutorial | Darknet Email*. URL: <https://thetinhathat.com/tutorials/messaging/i2pbote.html> (visited on 03/24/2018).
- [14] Chris Ball. *Serverless-Webrtc: A Demo of Using WebRTC with No Signaling Server*. URL: <https://github.com/cjb/serverless-webrtc> (visited on 03/26/2018).

- [15] C. Partridge. "The Technical Development of Internet Email". In: *IEEE Annals of the History of Computing* 30.2 (Apr. 2008), pp. 3–29. ISSN: 1058-6180. DOI: [10.1109/MAHC.2008.32](https://doi.org/10.1109/MAHC.2008.32).
- [16] Isnar Sumartono and Andysah Putera Utama Siahaan. "Base64 Character Encoding and Decoding Modeling". In: 02.12 (2016), p. 7.
- [17] *Send Attachments with Your Gmail Message - Computer - Gmail Help*. URL: https://support.google.com/mail/answer/6584?topic=1517&visit_id=1-636639395582587636-84449674&rd=1 (visited on 06/07/2018).
- [18] *Celebgate: Two Methodological Approaches to the 2014 Celebrity Photo Hacks*. URL: https://www.researchgate.net/publication/300884444_Celebgate_Two_Methodological_Approaches_to_the_2014_Celebrity_Photo_Hacks (visited on 06/07/2018).
- [19] *Cloud Leak: How A Verizon Partner Exposed Millions of Customer Accounts*. URL: <https://www.upguard.com/breaches/verizon-cloud-leak> (visited on 06/07/2018).
- [20] Dan O'Sullivan. *Black Box, Red Disk: How Top Secret NSA and Army Data Leaked Online*. URL: <https://www.upguard.com/breaches/cloud-leak-inscom> (visited on 06/07/2018).
- [21] Glenn Greenwald et al. "Microsoft Handed the NSA Access to Encrypted Messages". In: *The Guardian. US news* (). ISSN: 0261-3077. URL: <http://www.theguardian.com/world/2013/jul/11/microsoft-nsa-collaboration-user-data> (visited on 06/07/2018).
- [22] Bogdan Popa. *Skype Provided Backdoor Access to the NSA Before Microsoft Takeover [NYT]*. URL: <http://news.softpedia.com/news/Skype-Provided-Backdoor-Access-to-the-NSA-Before-Microsoft-Takeover-NYT-362384.shtml> (visited on 06/07/2018).
- [23] *Is It Safe to Transfer Files via Skype?* URL: <http://smallbusiness.chron.com/safe-transfer-files-via-skype-66706.html> (visited on 06/07/2018).
- [24] Denis Trcek. *Managing Information Systems Security and Privacy*. Springer Science & Business Media, Jan. 26, 2006. 245 pp. ISBN: 978-3-540-28104-7.
- [25] Mihir Bellare and Phillip Rogaway. "Introduction to Modern Cryptography". In: *UCSD CSE 207 Course Notes*. 2005, p. 207.
- [26] John R. Vacca. *Public Key Infrastructure: Building Trusted Applications and Web Services*. CRC Press, May 11, 2004. 446 pp. ISBN: 978-0-203-49815-6.
- [27] Philip R. Zimmermann. *The Official PGP User's Guide*. Cambridge, MA, USA: MIT Press, 1995. ISBN: 978-0-262-74017-3.
- [28] Li Li et al. "Who Is Calling Which Page on the Web?" In: *IEEE Internet Computing* 18.6 (Nov. 2014), pp. 26–33. ISSN: 1089-7801. DOI: [10.1109/MIC.2014.105](https://doi.org/10.1109/MIC.2014.105). URL: <http://ieeexplore.ieee.org/document/6879057/> (visited on 03/24/2018).
- [29] Audun Jøsang, Roslan Ismail, and Colin Boyd. "A Survey of Trust and Reputation Systems for Online Service Provision". In: *Decision Support Systems* 43.2 (Mar. 2007), pp. 618–644. ISSN: 01679236. DOI: [10.1016/j.dss.2005.05.019](https://doi.org/10.1016/j.dss.2005.05.019). URL: <http://linkinghub.elsevier.com/retrieve/pii/S0167923605000849> (visited on 03/22/2018).

- [30] John R. Douceur. "The Sybil Attack". In: *Peer-to-Peer Systems*. International Workshop on Peer-to-Peer Systems. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Mar. 7, 2002, pp. 251–260. ISBN: 978-3-540-44179-3 978-3-540-45748-0. DOI: [10.1007/3-540-45748-8_24](https://doi.org/10.1007/3-540-45748-8_24). URL: https://link.springer.com/chapter/10.1007/3-540-45748-8_24 (visited on 03/26/2018).
- [31] Jonathan Rosenberg et al. *STUN - Simple Traversal of UDP Through Network Address Translators*. URL: <https://tools.ietf.org/html/rfc3489> (visited on 03/26/2018).
- [32] Gonzalo Camarillo, Oscar Novo, and Simon Perreault. *Traversal Using Relays around NAT (TURN) Extension for IPv6*. URL: <https://tools.ietf.org/html/rfc6156> (visited on 03/26/2018).
- [33] Jonathan Rosenberg <jdrosen@cisco.com>. *Interactive Connectivity Establishment (ICE): A Methodology for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*. URL: <https://tools.ietf.org/html/rfc5245> (visited on 03/26/2018).
- [34] Sam Dutton Published: July 23rd, 2012 Updated: February 21st, and 2014 Comments: 7 Your browser may not support the functionality in this article. *Getting Started with WebRTC - HTML5 Rocks*. URL: <https://www.html5rocks.com/en/tutorials/webrtc/basics/> (visited on 06/11/2018).
- [35] Emil Ivov et al. *Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol*. URL: <https://tools.ietf.org/html/draft-ietf-ice-trickle-17> (visited on 03/26/2018).
- [36] Mark Handley, Colin Perkins, and Van Jacobson. *SDP: Session Description Protocol*. URL: <https://tools.ietf.org/html/rfc4566> (visited on 03/26/2018).
- [37] *A Study of WebRTC Security · A Study of WebRTC Security*. URL: <http://webrtc-security.github.io/> (visited on 06/24/2018).
- [38] Jonathan Lennox <lennox@cs.columbia.edu>. *Connection-Oriented Media Transport over the Transport Layer Security (TLS) Protocol in the Session Description Protocol (SDP)*. URL: <https://tools.ietf.org/html/rfc4572> (visited on 06/11/2018).
- [39] *WebRTC and Man in the Middle Attacks*. URL: <https://webrtcchacks.com/webrtc-and-man-in-the-middle-attacks/> (visited on 06/24/2018).
- [40] Vanessa Wang, Frank Salim, and Peter Moskovits. *The Definitive Guide to HTML5 WebSocket*. 1st. Berkely, CA, USA: Apress, 2013. ISBN: 978-1-4302-4740-1.
- [41] Audun Jøsang and Simon Pope. "Semantic Constraints for Trust Transitivity". In: *Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005 Vol. 43)* (), p. 10.
- [42] Taylor Kimmet. *Rtc-Pubnub-Fileshare: A File Sharing Demo Built Using PubNub and WebRTC*. URL: <https://github.com/tskimmet/rtc-pubnub-fileshare> (visited on 03/26/2018).
- [43] Mark Watson. *Web Cryptography API*. URL: <https://www.w3.org/TR/WebCryptoAPI/> (visited on 03/26/2018).
- [44] *Web Crypto API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API (visited on 03/26/2018).
- [45] "PKCS #1 v2.2: RSA Cryptography Standard". In: (2012), p. 63.

- [46] *Index | Node.js v7.10.1 Documentation*. URL: <https://nodejs.org/docs/latest-v7.x/api/> (visited on 03/26/2018).
- [47] *Chrome V8*. URL: <https://developers.google.com/v8/> (visited on 03/26/2018).
- [48] *Sctptransport.Cc - Code Search*. URL: https://cs.chromium.org/chromium/src/third_party/webrtc/media/sctp/sctptransport.cc?g=0 (visited on 06/23/2018).
- [49] *About Electron | Electron*. URL: </docs/tutorial/about> (visited on 03/26/2018).
- [50] Sindre Sorhus. *Clipboardy: Access the System Clipboard (Copy/Paste)*. URL: <https://github.com/sindresorhus/clipboardy> (visited on 03/26/2018).
- [51] Samuel Perrichon. *Electron-Prompt: Electron Helper to Prompt for a String Value*. URL: <https://github.com/sperrichon/electron-prompt> (visited on 03/26/2018).
- [52] *Chrome | WebRTC*. URL: <https://webrtc.org/web-apis/chrome/> (visited on 03/26/2018).
- [53] jQuery Foundation- jquery.org. *jQuery*. URL: <https://jquery.com/> (visited on 03/26/2018).
- [54] Mark Otto and Jacob Thornton. *Bootstrap*. URL: <https://getbootstrap.com/> (visited on 03/26/2018).
- [55] *Webcrypto: W3C Web Cryptography API for Node.js*. URL: <https://github.com/anvilresearch/webcrypto> (visited on 06/30/2018).