Direct And Automatic File Transfer
A secure alternative to email attachments utilizing WebRTC

Robin Lunde[1] and Keiji Takeda[1]

Keio University Shonan Fujisawa Campus
Kanagawa Prefecture, Fujisawa, Endo, 5322
252-0882, Japan
robin@keio.jp
https://www.robinlunde.tk
keiji@sfc.keio.ac.jp

Abstract. In today's technical environment there are countless new and secure ways to send files over the Internet. Yet most people still use traditional, outdated e-mail attachments to share files. In this paper we will propose a new system as an alternative to e-mail attachments, that seek to improve overall security and usability. We suggest a system utilizing the web of trust model combined with WebRTCs peer to peer functionality. By using the web of trust model, we give the end-user 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 drastically reduces the risk of an attacker gaining access to the file while also optimizing transfer-speed. We are working on an implementing this in an application called SendIt, to demonstrate the feasibility of our proposal.

Keywords: Usable security · Real World Cryptography · WebRTC · Trust management · File transfer

1 Introduction

1.1 Motivation and contribution

This paper seeks to combine the advantages of P2P (*Peer to Peer*) technology with the web of trust model, to increase every-day users efficiency and security, when transferring files over the Internet. We want to utilize new technology to improve the current situation, while keeping it easy to use. It is not rare that the decryption-key and cipher is sent over the same channel when sharing encrypted files. This holds true even for security specialists! This signals that there is a lack of good implementations currently available. The Internet began as a decentralized network but has gradually converged in to a more centralized network of servers.[1] Our goal is to break with this development and move towards a decentralized system where central services are only used when strictly necessary.

While SendIt might not be revolutionary in a technical aspect, it is an application that is simple to use and improves the current conditions regarding e-mail attachments. It is our belief that making such an application will demonstrate how this new technology can be used and show that even people not familiar with the technology can utilize it to protect their privacy and improve their overall security while transferring files.

P2P: 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.

Attack Mitigation: Another benefit of switching to direct communication is that it reduces an attackers options significantly when trying to steal information. First of all, there is two less places for the attacker to listen for data, as there is no information to the e-mail server nor from the e-mail server (regarding the file), because the only information that is sent via this channel is the connection details. The communication is also done in a much shorter timespan, as information about the file is only available once both parties are online and ready to start the transfer. These changes mean that the attack surface is lower compared to regular systems. WebRTCs P2P-communication is done over DTLS-SRTP (*Datagram Transport Layer Security over Secure Real-time Transport Protocol*) or DTLS-SCTP (*Datagram Transport Layer Security over Stream Control Transmission Protocol*) by default[2], which means passive listening attacks are not effective, even if the attacker can monitor one or both parties.

SendIt also reduces the threat and impact of automatic spreading of malicious files, as is currently a common attack-pattern. This is because it requires user-interaction on both ends for the transfer to take place. Also, there is no way to automatically forward files since SendIt does not have this functionality. There is also no support for communication other than one-to-one, which reduces the rate at which the threat can spread significantly. Finally, SendIt is a local program and does not connect to the internet before actively starting a file transfer. Once it does connect to the internet, it does so for a very limited time, and in an encrypted manner. This allows for no external tampering or code-manipulation at runtime. The fact that there is no server of any kind involved, also means that there is no way for an attacker to manipulate or leverage trust in the server to his advantage, as is an issue in server-based solutions.

Trust management: While these improvements are good, SendIt now lacks someone to trust. In common use-case scenarios the browser places trust in the website. This is commonly achieved using the PKI-model (*Public Key Infrastructure*). Since there is no server involved in our solution, it makes little sense to involve one just for authentication users. As such we based our system on the web of trust model originally proposed in combination with PGP (*Pretty Good Privacy*).[3] This allows for authentication of users without having a central authority verifying each identity. Trust is built over time and corroborated by other users. This allows for a flexible system that also has somewhat trusted entities and can authenticate these entities reliably. There exists known issues with this solution as well, but we will propose ways to handle such issues in a tolerable manner.

The main advantages of this model is that each user can decide which level of trust is satisfactory. The system will rate each identity based on certain criteria and assign each identity a level of trust. This trust will be reputation-based and will exist as a product of all interactions in the system. There will be no centralized management, but the system will naturally share the reputation of each identity and keep an overview of each identity's reputation. This system also allows you to use different identities depending on who you are communicating with.

It is our intention to display the level of trust for the identity you are about to communicate with. This is so the user can make an individual decision for each connection. This can be turned off in the options menu. It allows for a flexible system where the user is in control of each interaction, without feeling overwhelmed by too much information, which is fitting for our use.

Following sections: We begin by reviewing related works (Sect. 2). In Section 3 we seek to clarify the implementation and theory behind SendIt. We will go through each component and explain the different approaches as well as our chosen implementation, with both its positive and negative aspects. Then we will discuss the results, impact and limitations of SendIt (Sect. 4). Afterwards we discuss future work (Sect. 5) and rounded off the paper with a conclusion (Sect. 6).

2 Related work

The most important work related to our solutions is PGP [3] and WebRTC [4] as well as DataChannels [2,5]. The research and implementations stemming from these papers are vital for our own choices and implementation. PGP and WebRTC are the groundwork upon which we have built our system. The DataChannels, 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 our system. FireFox Send is a quite recent release (Jan. 8th 2017), that also runs on NodeJs. 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.[6] This is in contrast to SendIt's direct solution.

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. I2P-Bote is also hard to use, and even the instructions are difficult to understand for people without technical backgrounds.[7,8]

We also used an existing implementation as a reference for our application. The Serverless-WebRTC solution [9] was what sparked the idea of making a system no longer depending on servers, for sharing files. The solution is quite basic, but is a good proof of concept.

3 Proposed system and implementation

3.1 Key-management:

Theory: The encryption and authentication scheme used in SendIt is based on public key cryptography. It 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. SendIt also uses digital certificates, like the ones used in the web of trust model.[3] Certificates traditionally contain lots of 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.[3] (More on introducers and certificates in Section 3.2).

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 is 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 we need to keep more information than just the key. For example we also need to know who it belongs to. For our system, we also need to store information relevant to the trust-level of each key. The easiest way to do this, is to store the certificate of each key with the key. That means we have to re-calculate the trust in a key every time it is loaded. This is costly when it comes to computational resources, but allows us to re-calculate the trust we have 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.

The last part of the puzzle is: how do we authenticate a user based on this information? There are many different schemes but we will be focusing on the one suggested by PGP for securely exchanging messages.[11] The system is based

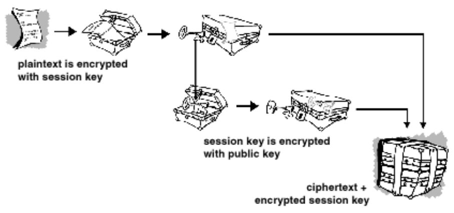
plaintext is encrypted with session key
session key is encrypted with public key
ciphertext + encrypted session key

Fig. 1. PGP encryption[10]

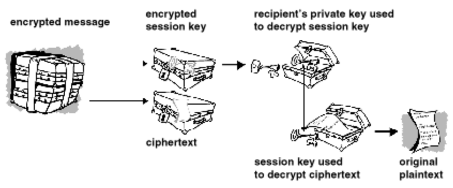
encrypted message
encrypted session key
recipient's private key used to decrypt session key
ciphertext
session key used to decrypt ciphertext
original plaintext

Fig. 2. PGP decryption[10]

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 PGP is shown in Figure 1 and Figure 2.

This solution is made with a one-directional exchange of data in mind. We will be using this system to authenticate both end-points, which means we will have bi-directional communication. We assume that both end-points already have each others public keys. This leads to SendIt not needing the session key. An in depth explanation will follow in the next section.

Implementation: In SendIt's implementation, we chose to use the Web Crypto API (SubtleCrypto module) for generating and handling keys.[12] This allows for an easy and reliable way to standardize the handling of keys, encryption and decryption in every system. It is available through the Chrome browser, from version 37.[13] Using a pre-developed and tested API for our cryptographic functions allows us to focus on the implementations of our system, rather than the implementation of the cryptographic functions. This is our reasoning for using the Web Crypto API.

Our implementation begins with creating our own 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, you can change to the identity you want to use during this connection. The key-pairs used consists of two RSA-OAEP 2048 bit keys, but this is trivial to change if the need arises. The choice of key-type was based on the recommendation of the WebCrypto API specification.[12].

The created key-pair is not stored on disk until the public key has been successfully exchanged with another identity, to avoid storing unnecessary keys. The key-exchange is done over the secure data channel created by WebRTCs PeerConnection.[4] Once a key exchange (successful connection) has taken place, the created key-pair and the other identity's public key will be written to disk. This file will be encrypted with a password, so that if the file is somehow stolen by an attacker, they can not trivially get access to the victims private key.

SendIt imports the file in to memory, once a key is needed. After which it extracts the keys stored, once the correct password is provided. If the wrong password is provided, the keys can not be accessed. Once the keys have been used and the communication has finished, SendIt will overwrite the existing file with new information. This allows users to change the password used to store keys between each use and also makes 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.

The file containing the keys contains information exceeding just the known keys. The identity associated with a key, and the key itself has to be stored. We also need to store the certificate for each key so we can re-evaluate the trust before use. The file will contain the fields: Name, Key and Certificate, for each key. The name and key fields are self-explanatory. The Certificate field will be described in following section.

3.2 Trust-model:

Theory: As previously stated, we are basing our model on the web of trust model. It suggests that we should split trust in to two groups: Trustworthiness of public-key certificate (How willing are we to trust this key pair when it is being used to communicate?) and Trustworthiness of introducer (How willing are we to trust this key pair when it is being used to introduce another key?).

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

There is also no recommendation of which trust level should be considered safe or unsafe. This is left up to each individual user.[3] There exist many ways to rate and evaluate trust, but we believe this simple point-based system to be the best fit for our use. The web of trust model has also been recommended by other researchers, albeit for a more traditional WebRTC authentication system.[14] For a look in to more options, Jøsang et al. 2007[15] gives a good overview over possible alternatives.

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 we will disregard information from identities we do not trust (their introductions mean nothing), while still valuing and receiving updated information from identities we 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.

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 our 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.[16]

Implementation: SendIt is built on the principal that the first exchange of offer/answer is trusted, even though it is unencrypted. All subsequent interactions is built on this trust. Only when there is reason to doubt the first exchange, will this be considered incorrect. One way for this to happen is for many others to have signed a certificate for an identity, and where the certificate-values match, while the key currently held differs.

We should reconsider our trust in a key, if one or more signatures of trusted identities lead to a key that is different from the one we are currently trusting. This initial trust is the main weakness of SendIt, but it allows for an easy and convenient way to get started when communicating with new partners. We leave it up to the users to consider how to and when to share this initial exchange, in a way that minimizes the risk of an attacker interfering.

Trust-building in our system is based on several factors. We intend to use trust transitivity [17] (Trusting in Alice and Alice trusting Bob, means that we trust Bob more, than if Alice was not involved). We also intend to use frequency of communication, and bi-directional communication (Acting as both sender and receiver with the same identities involved) as ways of increasing trust. Trust built by such activities will gradually accumulate, in contrast to transitive trust. Transitive trust is just a way of assigning an identity an initial trust level based on the introducer's 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 we try to communicate with a previous partner, but that partner is not authenticated according to the key we associate with that partner. We will be implementing SendIt in a way that gives more weight to reduction in trust. By this we mean 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 us maintain a balanced system that detects untrusted behavior rapidly and acts accordingly.

Trust is shared by using certificates. Unlike regular certificates (see Sect. 3.1), SendIt will use a minimal and simple solution. In our system a certificate is just a list of identities and the corresponding cipher. The cipher is made by encrypting the public key of the identity you are vouching for with your 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 described above. 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 we can assess how much trust the introducer has in the introduced key.

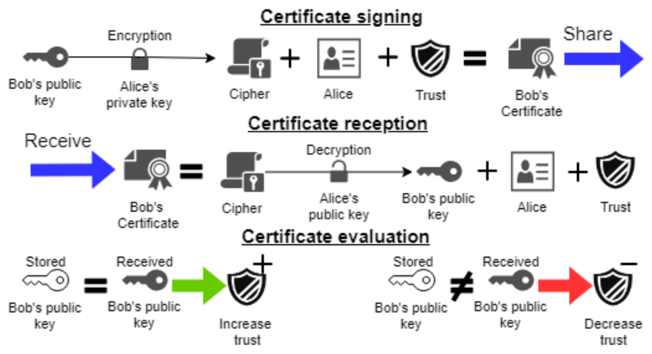

Fig. 3. Certificate operations example where Alice signs Bob's certificate

We are in the progress of determining the details of which approach to use and how to evaluate it, when it comes to transitive trust. The specific values and model for assigning trust has yet to be finalized, but the idea is to assign different levels of trust based on the amount of introducer's 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 end-point 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 for an example. Do note that one certificate can hold many signatures.

As stated above, we have yet to finalize the practical evaluation scheme for SendIt. Preliminary ideas are very close to the original model. We intend to assign these trust-levels based on a point system, where an identity accumulates points by getting verified by introducers that the user trusts. We plan to have at least five 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 will also depend on how much this introducer trusts in the key. This is so we can increase the level of transitive trust, since the system is based on nuances of trust. SendIt will allow for easy and intuitive evaluation, while also having a simple way to fairly assert the trust level of an identity. Unlike the web of trust model, we will not have different trust levels for communication and referring. There will only be one trust level assigned to each identity and it will apply to both uses equally.

The practical impact of the different trust levels will 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 will 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 will appear. SendIt will have values pre-configured so novice users can rely on the standard settings. If they want to change these settings, they can choose to do so. The users will also have an overview of all identities and 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 the theory section, the Sylbil attack is a problem in decentralized, reputation based networks. Our 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. We will do this by requiring interactions with other identities, as well as a signed certificate from trusted introducer's 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 it 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.

3.3 Key- and certificate-sharing

The exchange of keys and the trust each identity assigns these keys are central to the web of trust model, as described above (Sect. 3.2). Since the connection used for communication is already secure and each party authenticated, we can transfer this data without encryption, since the communications channel takes care of it. This section will describe how we intend to implement this in our system.

Request keys: SendIt will have an automated, bi-directional sharing of keys and 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 keys: While exchanging keys and certificates with all communication-partners is important to avoid biases, 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 in to 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 in Section 3.2. 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 a untrusted-user warning before communication occurs, or a connection terminated message, depending on the users settings. These keys will still be stored in the users key-chain and stay available for future evaluation, even if their current level of trust is low.

3.4 Connection setup:

Theory: To set up a connection using WebRTC, there are different variables one has to take in to account. First, what are the network conditions for each endpoint? To be able to create a connection, we need to find a way to address each end-point directly. To do this WebRTC utilizes STUN (*Session Traversal Utilities for NAT*)[18] and/or TURN (*Traversal Using Relay NAT*)[19], and ICE (*Interactive Connectivity Establishment*)[20].

The ICE-protocol works by trying to connect directly to the end-point. If an end-point is behind symmetrical NAT ((*Network address translation*), however, a TURN server is required. A TURN-server acts as a relay-server that both end-points connect to, and there is no longer a direct P2P-connection. Instead each end-point is connected to a server by a P2P-connection, and the server forwards information to the other end. It is the ICE-protocol that manages these issues and tries to find the best possible connection. [21]

ICE is a protocol for NAT-traversal used in offer/answer protocols. In regular WebRTC-applications, something called ICE-trickling is used. This allows the setup of the channel to happen first, then ICE-candidates to be shared. [22] In our application, however, this information is attached to the offer/answer and shared. ICE is used to be able to address and reach computers behind NAT.[20]

SDP stands for Session Description Protocol, and 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 end-point.[23] Since it is up to the users how they share the offer and answer, we have no transfer-protocol in our solution. The WebRTC data channels we use for direct communication uses DTLS-SCTP as their transport-protocol.[2]

The encapsulation of SCTP over DTLS, ICE- and SDP-protocols are complex. For those unfamiliar with the technology, an in depth explanation is given in Holmberg, H.-C. (2015)[5]. In summary DTLS-SCTP provides confidential, source authenticated and integrity protected data transfers and is encrypted using DTLS. In practice each end-point gathers the necessary connection data and then formats it according to the SDP-protocol, for session-negotiation and initiation.

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.

In SendIt, this channel does not exist. Instead of using a signaling-server, we collect all the information before sending an Offer/Answer. This means a slight delay from initiating the connection on each side, until the offer/answer is ready, but it allows the technology to function without any direct server-involvement. We still need a way to share the offer/answer, but we are leaving that up to the user to decide. E-mail and instant messaging are both viable options.

There is several reasons for the usage of a signaling-server. As mentioned, one is the ability to reconnect in the event of a failure in the established connection. Another is that it allows the offer and answer to not be created until both end-points are online, then immediately created and shared. This is not the case for SendIt, which is affected by the fact that network conditions can change rapidly and as such ICE-candidates may no longer be viable, leaving end-points with no way to make a connection or renegotiating the connection.

Most of our efforts are currently focused on finding a way to extend the lifetime of the connection offer/answer, since it allows for a bigger time frame in which it can be utilized. The main concern regarding this issue is that the router will close the ports opened for the connection, if no traffic has been detected for a while. While this varies from network to network, our goal is to implement a mediator, which can keep the port open while waiting for a connection to be made. See Section 4 and Figure 5 for results and analysis of experiments regarding the lifetime of a connection offer/answer exchange.

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/answer, as they contain information that allows a secure connection to be set up.

In SendIt we do not have a secure channel over which we exchange the offer/answer. That means it would be vulnerable to a MitM-attack (*Man in the Middle*). To combat this the offer and answer will be encrypted by each end-points public key, on all connections after the first (See Figure 4). With this protection in place an attacker can not get a hold of the offer/answer in time, without stealing a private key.

Implementation: In our implementation, we leave most of the work to the WebRTC-API. The connection is set up as demonstrated in 'section 7.2: Procedures' in Holmberg, H.-C. (2015) [5]. We only divert from this standard setup of WebRTC connections once. Namely, in the order of sharing the offer/answer and collection of ICE-candidates. We first create an offer/answer. Then we register a way to collect ICE-candidates. The code used is shown in Listing 1.1.

Listing 1.1. ICE collection

```
RTCPeerConnection.onicecandidate = function (e) {
  if (e.candidate == null) {
    var off = JSON.stringify(RTCPeerConnection.localDescription);
    $('#localDescription').html(off);
    }
}
```

When we receive an ICE-candidate that is null, it means the ICE-candidate collection is finished. Once it has finished, we show the user the local description as a whole (SDP and ICE-candidates), so the user can share it with the other end-point. The usual way is to separate the SDP-generation and ICE-gathering completely. The offer/answer is shared first, then the ICE-candidates are shared as they are gathered. If the above process is completed, a connection is established. There are scenarios, when this is not case though. Known causes of issues with completing the connection are:

– Set-up not completed within a certain time-frame (See figure in theory section above)
– One or both end-points are behind symmetrical NAT
– One or both end-points change network location (E.G. connect to a different network.)

When implementing SendIt, we chose 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. We hope that more people will start using IPv6, as NAT traversal will no longer be necessary. As for changes in network-conditions, there is nothing to be done on the application-side, except including a signaling-server. As such we assume that users will stay in the same network-conditions for the duration the connection is active. We are working on a 'resume transfer'-function, in case of network failure, as described in Section 3.5.

As discussed in the theory-section, the offer/answer has to be encrypted to avoid MitM-attacks. This brings up the issue of the first connection between identities that have never communicated or been introduced before. If nobody has communicated with this identity before, there is no established key by which you can encrypt communication. As such we define the first interaction between end-points as a trusted interaction and allow the setup of keys to happen. For more in-depth information, see Section 3.2.

SendIt's authentication scheme relies on a combination of public key cryptography and the offer-answer exchange that is necessary for WebRTC to work. In a similar way to the message-exchange in PGP explained in the theory part of Section 3.1, the offer and answer will be exchanged. The difference is that there is no session key involved. The offer or answer is directly encrypted using the public key associated with the current connection and then shared (See

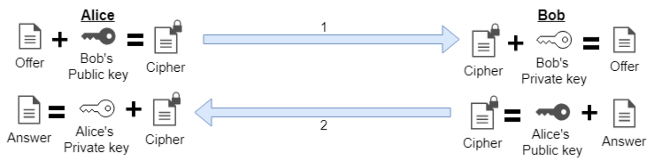

Fig. 4. Offer and answer exchange

Figure 4 for example). Only someone possessing the private key can decrypt the offer or answer, and as such, both parties can safely assume that they are communicating with the correct identity. If the answer does not match the offer, WebRTC terminates the connection.[4] As such all successful connections made are authenticated and can not be forged in any way.

3.5 Communication:

Theory: The P2P connection and communication in WebRTC works like any other P2P protocol, where the communication happens directly between two end-points. The difference from traditional P2P systems is that it requires signaling through a second channel to set up the connection. The actual protocol and connection type depends on what type of communication you will be doing. For SendIt the communication is done over DTLS-SCTP, since only a DataChannel is created. For information about how DTLS-SCTP and connection setup works, see Section 3.4.

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 in to chunks during reading and then transferring these chunks allows for infinitely big transfers. The default max size for NodeJS 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 NodeJS can have in memory at the time.[24,25]

As explained in Section 3.4 a signaling-server usually allows for renegotiation of the connection, in case the communication breaks down.¡this is not possible using SendIt, since it has no such channel. This leads to having to start the whole process over, every time the connection is broken.

Another side-effect is that if we are transferring large amounts of data, and the connection drops, nothing is stored on the receiving end. We want to implement a solution where each end-point stores records of communications. This is for storing information about what has been transferred previously and for storing information to allow resuming communication from the point it was interrupted. This will negate problems regarding large transfers being interrupted and also make SendIt more user friendly. To do this the receiver needs to be able to tell the sender how much of the previous transfer he received, before starting retransmission.

Implementation: Our implementation of communication is fairly straight forward. It uses the WebRTC API for sending and receiving all the data. We do, however, use another protocol based on PubShare[26] (See main.js) for communicating the status of file transfer. This protocol takes care of chunking the data, sending file meta-data, and verification of completed delivery. We expect the overhead of the file-transfer protocol to be negligible, but we intend to run experiments to compare the transfer-speed with and without the protocol.

As noted in the theory-section above, the support for large files is limited. Implementing chunking of files should be trivial though, so we will not be developing support for transport of data larger than the current limitations. If this is needed in the future, we will add it as an extra feature. The way to implement such a feature would be to read chunks of the file at a time. Then, once a chunk is completely transferred, the next chunk is read in to memory. This will allow both endpoints to handle smaller amounts of data at the time, whilst still receiving the whole file after the transfer is complete. It is our recommendation that this is not implemented until the 'resume transfer'-feature is implemented, as transferring large amounts of data without any way of resuming it in case of failure, is suboptimal.

Ideally, we would also solve the problem of re-negotiating a connection. Because of our way of implementing the system, this is not possible without server involvement, and as such we will instead be relying on the functionality to resume unfinished transfers as a tool to remedy this issue. The connection setup will still be required, but the data transferred will not be lost. This is a trade-off we believe is worth it, to avoid server involvement.

We plan to implement the 'resume transfer' function based on the communication record. Since every identity will have a list of previously transferred files, it will have file name included. If the transfer is not complete, we can store information about which chunk of the file was the last received, and request the transfer to be continued from there.

If the sender is not willing to resume the previous transfer, it will either start over, or another file will be transferred. This decision is up to the sender's settings and choices. If the sender choses to not resume the transfer, the data previously stored on the receivers local system will be removed, and the record updated as failed transfer. Our system will only allow for the requested file to be shared on the subsequent connection. By this we mean:

Alice tries to send Bob File A, but the connection is broken. If Alice contacts Bob again, but tries to send File B this time, the previously transmitted information stored by Bob is removed. 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. We plan to implement the record of

Table 1. Record of communication fields

Field:	Name	File(s)	Date	Completed
Example:	test@email.com	picture.jpg	2018-03-20	0

communications by creating a log that contains the fields indicated in Table 1. The last field will either be -1 (meaning failed), 0 (meaning success) or the last received chunk.

3.6 Application:

SendIt is developed in Javascript utilizing the Electron framework. We chose to use these technologies because they allow for easy implementation while supporting multiple operating system. It also allows for the use of existing libraries and standardizations developed for web-browsers, camouflaged in the appearance of a normal application.

By this we mean 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 (*Graphical User Interface*). 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 NodeJS for the back end and Chromium for the front end. [27] In practice, this means that we are using: HTML, Electron and Javascript, where Electron uses Chromium and NodeJS. This allows us to use modules and frameworks from any of the entities mentioned above, independently. Our implementation uses these libraries and frameworks:

- NodeJS API - Used for reading and writing to disk, finding correct files and folders, [24]
- Chrome Web Cryptography API - Handling the creation and export of keys, encryption and decryption [12,13]
- NodeJS Clipboardy - Used to automatically copy the generated offer/answer to the clipboard [28]
- NodeJS Electron-prompt - Used to create pop-ups requesting input [29]
- Chrome Native WebRTC - Used for creating and managing WebRTC connections [30]
- jQuery v3.2.1 - Managing front-end actions and dynamic updates [31]
- Bootstrap v3.3.7 - Managing front-end modules and dynamic updates [32]

The connection setup via WebRTC is based on this example.[9] It has been changed and customized to only work with WebRTC data channels and has also been extended significantly.

All in all, SendIt abstracts away all the intricacies of the implementation and displays the information through an easy to understand GUI. It makes choosing functionalities or modifying settings easy to do. Files are automatically stored in a pre-configured location. Sending can happen from a pre-configured location as well, or be manually added in the sending-screen. Key management, creation and updates are done automatically, but the users can access and modify these behaviors as they deem necessary from the settings screen.

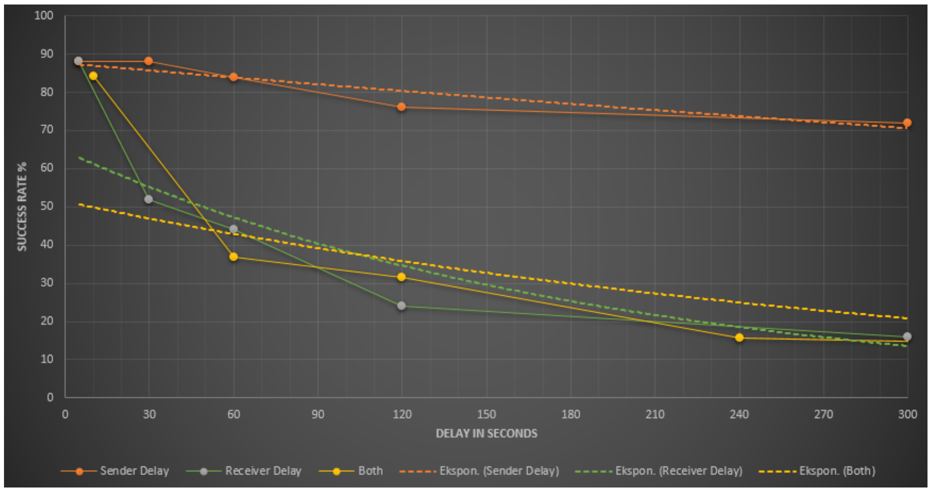

Fig. 5. Delayed WebRTC connection setup success rate

4 Results

In the experiment shown in Figure 5, we measured the success-rate of connections between a server and a client, using WebRTC, where we varied the time it took before the offer/answer was shared. You can see the success-rate represented on the vertical axis, and the total delay before trying to create the connection on the horizontal axis. In this experiment we are trying to mimic the setup between two end-points using SendIt, and as such the connection setup is done in the same way. We assigned the server the role of Sender, for all connections. The terminology is as follows:
Senders delay(S)**:** is delay from creation of the offer, until it is received by the client.
Receivers delay(R)**:** is delay from creation of the answer, until it is received by the server.
Both: is the delay split equally between both of the above. If the delay is 10 seconds, it means 5 seconds senders delay and 5 seconds receivers delay, for a total of 10 seconds delay.

The size of the dataset is 25 connections. We removed connections that failed all tests because the probability of failing all tests and still being a supported connection is very low. We assume these connections are not in the target group of computers eligible to use SendIt, and as such would skew the results.

Now let us discuss the findings and implications of Figure 5. As we can see, S is fairly irrelevant. Even at 300 seconds (5 minutes) the success rate is 72%. R is much more important. We can see that at 5 seconds delay, S and R has 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 receivers end. Splitting the delay equally between S and R (as indicated by 'Both'), 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 end-points. As such, another experiment should be done where the server acts as R, so we can compare the results and see if this changes our conclusion.

5 Future work

The need for better connectivity is quite clear from the current situation. While different implementations are possible, we believe that keeping with the core philosophy of a decentralized internet is important. As such, finding ways to improve connectivity without having to rely on a server is something that would improve our solution.

The problem of bad connectivity can also be negated by finding a way to let end-points reconnect without having to go through a whole new connection setup phase.

Another area where our solution is suboptimal is the initial trusted exchange. Finding a way to share this initial exchange in a way that ensures confidentiality and integrity without including a server would also heavily improve the system.

Finally, there is always room for improvement when it comes to trust building and evaluation. Optimizing the algorithms and policies to make sure that we get an optimal representation of trust for each identity will always be a challenge.

6 Conclusion

In this paper we have proposed a new prototype and system for transferring files. We are using P2P communication to avoid having to go through a server to share files. Since this makes the system vulnerable to multiple kinds of attacks, we suggest an authentication scheme based on public key cryptography.

Since we have no central authority to vouch for any of the keys or identities used, we combine the above technologies with a trust-system based on the web of trust model. This is a reputation based, decentralized model which allows each user to make their own choices, while still having a shared understanding of how trustworthy each identity is. All in all, we believe that SendIt is a good alternative to the way current e-mail file attachments work, while still having some weaknesses that should be looked in to.

References

1. Ibanez, L.-D., Simperl, E., Gandon, F., Story, H.: Redecentralizing the Web with Distributed Ledgers. IEEE Intelligent Systems. 32, 9295 (2017).

2. Jesup, R., Loreto, S. and Tuexen, M.: WebRTC data channels. draft-ietf-rtcweb-data-channel-13. txt, work in progress. (2015)
3. Alfarez A.: The PGP Trust Model. EDI-Forum: the Journal of Electronic Commerce **10**(3), 27–31 (1997)
4. Bergkvist, A., Burnett, D.C., Jennings, C., Narayanan, A. and Aboba, B.: Webrtc 1.0: Real-time communication between browsers. Working draft, W3C, 91. (2012)
5. Holmberg, H.-C.: Web Real-Time Data Transport, https://www.theseus.fi/bitstream/handle/10024/94759/FinalThesis_hanschrh_final.pdf?sequence=1&isAllowed=y, (2015).
6. Firefox Test Pilot - Send, https://testpilot.firefox.com/experiments/send/. Last accessed 20 Mar 2018
7. I2P-Bote, https://i2pbote.xyz/, Last accessed 20 Mar 2018
8. I2P-Bote Introduction and Tutorial — Darknet Email, https://thetinhat.com/tutorials/messaging/i2pbote.html, Last accessed 20 Mar 2018
9. Serverless-webRTC, https://github.com/cjb/serverless-webrtc, Last accessed 20 Mar 2018
10. How PGP works, https://users.ece.cmu.edu/~adrian/630-f04/PGP-intro.html, Last accessed 20 Mar 2018
11. Zimmermann, P. R. T.:The official PGP user's guide. MIT Press Cambridge, USA (1995)
12. Sleevi, R. and Watson, M.: Web cryptography API. W3C candidate recommendation, W3C. (2014)
13. Web Crypto API, https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API, Last accessed 20 Mar 2018
14. Li, L., Chou, W., Qiu, Z., Cai, T.: Who Is Calling Which Page on the Web? IEEE Internet Computing. 18, 2633 (2014).
15. Jøsang A., Ismail R., Boyd C.: A survey of trust and reputation systems for online service provision, Decision Support Systems **43**(2), 618–644 (2007). https://doi.org/10.1016/j.dss.2005.05.019
16. Douceur J.R.: The Sybil Attack. In: Druschel P., Kaashoek F., Rowstron A. (eds) Peer-to-Peer Systems. IPTPS 2002. Lecture Notes in Computer Science, vol 2429. pp. 251-260 Springer, Berlin, Heidelberg (2002).
17. Jøsang, A. and Pope, S.: Semantic constraints for trust transitivity. In Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling, vol 43. pp. 59-68. Australian Computer Society, Inc. (2005)
18. Rosenberg, J., Mahy, R., Huitema, C., and Weinberger, J.: STUN-simple traversal of UDP through network address translators. (2003)
19. Matthews, P., Mahy, R., and Rosenberg, J.: Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun). (2010)
20. Rosenberg, J.: Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal for offer/answer protocols. (2010)
21. Dutton, S.: Getting started with WebRTC. HTML5 Rocks **23** (2012).
22. Ivov, E., Rescorla, E. and Uberti, J.: Trickle ICE: incremental provisioning of candidates for the interactive connectivity establishment (ICE) protocol. (2013)
23. Handley M, Perkins C, Jacobson V.: SDP: session description protocol. (2006)
24. Node.js v9.8.0 Documentation, https://nodejs.org/docs/latest-v7.x/api/, Last accessed 20 Mar 2018
25. Chrome V8, urlhttps://developers.google.com/v8/, Last accessed 20 Mar 2018
26. PubShare, https://github.com/tskimmett/rtc-pubnub-fileshare, Last accessed 20 Mar 2018

27. Electron Documentation , https://electronjs.org/docs/tutorial/about, Last accessed 20 Mar 2018
28. Clipboardy, https://github.com/sindresorhus/clipboardy, Last accessed 20 Mar 2018
29. Electron-prompt, https://github.com/sperrichon/electron-prompt, Last accessed 20 Mar 2018
30. Chrome WebRTC, https://webrtc.org/web-apis/chrome/, Last accessed 20 Mar 2018
31. jQuery, https://jquery.com/, Last accessed 20 Mar 2018
32. Bootstrap, https://getbootstrap.com/, Last accessed 20 Mar 2018