

Development of a secure, peer-to-peer, decentralised anonymous chat system

Nico Schottelius (nico-zhaw.ch (o) schottelius.org)

June 19, 2012

Contents

1	Introduction	8
1.1	Preamble	8
1.2	Current status	8
1.3	Objectives	9
1.4	Tasks	9
1.5	Expected Results	10
2	Analysis and Comparison of Chat Systems	11
2.1	Internet Relay Chat (IRC)	11
2.1.1	History	11
2.1.2	Architecture	11
2.1.3	Security	12
2.2	Secure Internet Live Conferencing (SILC)	13
2.2.1	History	13
2.2.2	Architecture	13
2.2.3	Security	16
2.3	Extensible Messaging and Presence Protocol (XMPP)	16
2.3.1	History	16
2.3.2	Architecture	16
2.3.3	Security	17
2.4	Skype	17
2.4.1	History	17
2.4.2	Architecture	19
2.4.3	Security	19
2.5	Other	21
2.6	Security features and Comparison	21
3	Analysis of related communication protocols	23
3.1	Mix Networks	23
3.2	Tor	23
3.3	Freenet	24
3.4	I2P	25
3.5	Off-the-Record Messaging (OTR)	25

3.6	Reliable UDP (RUDP)	26
4	Analysis of features and security requirements	27
4.1	Chat features	27
4.1.1	Multi User Chat (MUC)	27
4.1.2	File Transfer	27
4.1.3	Voice Communication	27
4.2	Security Requirements	28
4.2.1	Anonymity	28
4.2.2	Confidentiality	29
4.2.3	Authenticity	29
4.2.4	Availability	29
5	Chat Protocol Definition	31
5.1	Version	31
5.2	Basic Data Types ("EOFbdt")	32
5.2.1	The zero byte	32
5.2.2	ASCII numbers	32
5.2.3	Strings in general	32
5.2.4	Fixed length strings	32
5.2.5	Variable length strings	32
5.3	Simple Data Types ("EOFsdt")	33
5.3.1	Command (command)	33
5.3.2	Version (version)	33
5.3.3	Identification string (id)	33
5.3.4	Peer name (name)	34
5.3.5	Group name (group)	34
5.3.6	Message text (msgtxt)	35
5.3.7	Peer address (address)	35
5.3.8	Peer fingerprint (keyid)	36
5.4	Messages Types ("EOFmsg")	37
5.4.1	Parameter Overview	37
5.4.2	3000: Drop packet	38
5.4.3	3001: Forward packet	38
5.4.4	3002: Message / drop packet	38
5.4.5	3003: Message / forward packet	38
5.4.6	3004: Acknowledge	38
5.5	Packets Type Overview ("EOFpkg")	39
5.6	OpenPGP	40
5.7	Transport Protocols	41
5.7.1	Network packets ("postcards")	41
5.7.2	Tunnelling	41
5.7.3	Multiplexing / Variable Addresses	41

5.7.4	Access Methods	41
5.7.5	List of Supported Transports	41
5.8	Onion Routing	44
5.8.1	Source Based Routing	44
5.8.2	Onions	45
5.8.3	Packet Sizes	45
5.9	Noise	47
5.9.1	Latency	47
5.9.2	Bandwidth Usage	49
5.9.3	Fixed interval for sending	51
5.10	Features	52
5.10.1	Anonymity	52
5.10.2	Authenticity and Confidentiality	53
5.10.3	Availability	54
5.10.4	Direct chat	54
5.11	Summary	54
6	Implementation of the Prototype	55
6.1	Environment	55
6.2	Usage	55
6.2.1	Command Line Interface (CLI)	55
6.2.2	Cryptographic Operations	56
6.2.3	Listener	58
6.2.4	Noise	58
6.2.5	Peer	59
6.2.6	Onion	59
6.2.7	Server	59
6.2.8	Transport Protocols (TP)	63
6.3	Code Examples	63
6.3.1	Modular Design	63
6.3.2	Sequence Numbers	63
6.3.3	Queuing	64
6.3.4	Onion Creation	64
6.4	Test of the Prototype	68
6.4.1	Onion Structure	68
6.4.2	Noise Sending	68
6.4.3	Encryption	70
6.4.4	Receiving Peers Can Decrypt Message	72
6.4.5	Receiving Peer: Authenticity Verification	72
6.4.6	Effective Bandwidth Usage	72
6.4.7	Performance	73
6.5	Features	75
6.5.1	Anonymity	75

CONTENTS

6.5.2	Authenticity and Confidentiality	75
6.5.3	Availability	76
6.6	Related Project: Chat User Interface	76
7	Conclusions	77
7.1	Review	77
7.2	Future Work	78
7.2.1	Prototype Enhancements	78
7.2.2	Transport Protocol Support	78
7.2.3	Impact of decreasing packet size	79
7.2.4	Adding Peers / Key Exchange	79
7.2.5	Using Trust Levels	79
7.2.6	Multi User Chat	80
7.2.7	Spread Usage	80

List of Tables

2.1	Chat system comparison with security features	22
5.1	EOF Message Parameter Usage	37
5.2	EOF Message Parameter Description	37
5.3	Transport protocols	41
5.4	Packet sizes (experimental)	45
5.5	Average latency based on number of proxy peers	48
5.6	Maximum latency based on number of proxy peers	48
5.7	Minimal Outgoing Bandwidth Capability	49
5.8	De-Anonymisation Probabilities (for one packet)	53
7.1	Upcoming transport protocols	78

List of Figures

2.1	Schematic Overview of IRC	12
2.2	Schematic Overview of SILC	14
2.3	Silc: Private Message Delivery With Session Keys	15
2.4	Silc: Private Message Delivery With Private Message Key	15
2.5	XMPP Network Architecture	17
2.6	XMPP Sending Server Attack	18
2.7	XMPP Receiving Server Attack	18
2.8	Skype Network Architecture	20
3.1	Mixes: Packet Flow	24
4.1	Sender Anonymity	28
4.2	Receiver Anonymity	28
4.3	Sender-Receiver Anonymity	29
5.1	Packet Types	39
5.2	Transport Protocol Tunneling	42
5.3	Address Multiplexing	42
5.4	Direct Access	43
5.5	Indirect Access	43
5.6	Onion Routing	44
5.7	An Onion (2 and 3 layers)	45
5.8	Noise Workflow	47
5.9	Peer and Network Bandwidth	50
5.10	Features and Technologies	52
6.1	Crypto Command Usage	56
6.2	Generation of Public/Private Key Pair	57
6.3	Show Public/Private Key Pair	57
6.4	Export Public Key	57
6.5	Import Public Key	58
6.6	Listener Command Usage	58
6.7	Add and list listener addresses	59
6.8	Init Noise Directory	59

LIST OF FIGURES

6.9	Peer Command Usage	60
6.10	Peer Add	61
6.11	Onion Command Usage	61
6.12	Create and send an Onion	62
6.13	Server Command Usage	62
6.14	TP Command Usage	63
6.15	TP List and Route	64
6.16	Modular Design	65
6.17	Implementation: Big Picture	66
6.18	EOFID Code Example	67
6.19	Queues / Polling Code Example	68
6.20	Onion Example Code	69
6.21	Socat command line for testing	70
6.22	Sending data at 0.250s interval	71
6.23	Sending data at 0.125s interval	71
6.24	Effective Bandwidth Usage (0.125s)	73
6.25	Effective Bandwidth Usage (0.250s)	74
6.26	Effective Bandwidth Usage (0.500s)	74
6.27	Packets per second (without sending limit)	75
6.28	Packets per second (without sending limit, plain text)	76

Chapter 1

Introduction

1.1 Preamble

This project originally started in 2007 as an idea of the hacker community *!eof*, which has its roots in the the IRC network.¹ Although various members of this community had written custom chat software to chat "securely", most members, even today, are using IRC as their primary communication method.

Some virtual and physical meetings took place to overcome this weakness and to develop a stronger, more appropriate protocol and software. Due to the distributed nature of the community and low prioritisation, not much progress was made. As I am one of the members of this community, this bachelor thesis and the support and interest of the company I am working for, *local.ch*, allows me to finish this project.

The original name for the project chosen was *Eris Onion Forwarding*, or short *EOF*.² I keep this name and thus several references and names in this document are influenced by the original project name.

1.2 Current status

A lot of chat systems are currently available, but none of these systems implements an anonymous, decentralised and secure architecture.

The need for anonymity arises when people need to hide to whom they are talking to. When sending confidential messages in untrusted networks, even the basic knowledge of who is talking to whom may cause unwanted actions. This is for instance true in extreme situations, i.e. when reporters located in dangerous areas submit their messages in danger of life threats, but also applies to a lot of daily situations, be it in a business context, in politics or even in people's private lives.

¹Hacking here is referring to free and open source software development as well as creative use of technology.

²Eris is the Goddess of Discordia as defined in the Principia Discordia.

A decentralised architecture helps to improve reliability because communication is permitted to flow via different channels. This supports the ability to continue to communicate even under attack from an enemy. This is a general requirement for reliable communication and can be crucial in emergency situations.

Secure chat provides the ability to hide message content from eavesdroppers, as well as to ensure message and sender authenticity. Without this technique an attacker could use confidential information in an unwanted way or change the content of the messages. This feature ensures the user's trust regarding the authenticity of the communication. For instance a love message may be altered by a jealous person or may make a third person jealous.

The client, Thomas Gresch from local.ch, is running Skype for internal core communications. Skype is a closed source, encrypted, proprietary chat system owned by Microsoft. Because of the intensive efforts from Skype to hide how its internals are working, the software is considered untrustworthy. The internal communication is a crucial part of daily business and an outage, whether caused by accident or intend, causes direct monetary loss for the company.

For this reason the research for an open alternative that may succeed the use of Skype is supported by local.ch.

1.3 Objectives

The objective of this thesis is the analysis, design and development of a secure, decentralised chat system. The main focus is on defining and documenting the mode of operation and the protocol definition. Additionally a prototype should be implemented. The thesis should be based on a detailed analysis of current chat systems. The analysis will not only consider chat systems, but also related secure and anonymous communication systems.

1.4 Tasks

1. Detailed analysis and comparison of open and legacy chat systems to summarise current chat system features and their security characteristics
2. Analysis of features and security requirements
3. Analysis of related communication protocols
4. Development of a new chat protocol
5. Development of chat prototype using the new chat protocol
6. Test of the prototype
7. Preparation of a live demonstration of the prototype

1.5 Expected Results

1. Report and comparison of current chat systems including strengths and weaknesses
2. Requirement analysis
3. Report of related communication protocols including strengths and weaknesses
4. Protocol definition paper (containing chat features, data types, transport methods, security measures)
5. Technical documentation and prototype for the new chat system
6. Description of test results
7. Presentation of a successful anonymous, decentralised chat session, which includes proof of the required security by using example attacks.

Chapter 2

Analysis and Comparison of Chat Systems

In my analysis of chat systems, I will start with the Internet Relay Chat (IRC). It seems a natural starting point for me, as I am still an extensive user of this chat system for historical reasons. What worries me is its low security standard, which I will briefly describe. I will then turn to SILC, which has been developed to address IRC weakness regarding security. Secure message delivery has been its main innovation. XMPP is of special interest, because it has been designed as an open and adaptive chat system featuring a high level of decentralisation. The last chat system in this comparison is Skype, the only commercial system analysed. Due to its questionable development history and present, it is a controversially viewed chat system, albeit with a huge user base. I will end this chapter by giving an overview of omitted chat systems and a comparison of the security features.

2.1 Internet Relay Chat (IRC)

2.1.1 History

IRC has been developed since 1989 and was first formally documented in May 1993 in RFC 1459[29]. It is still being widely used.[27] The current protocol is specified in the RFCs 2810-2813[19, 20, 21, 22].

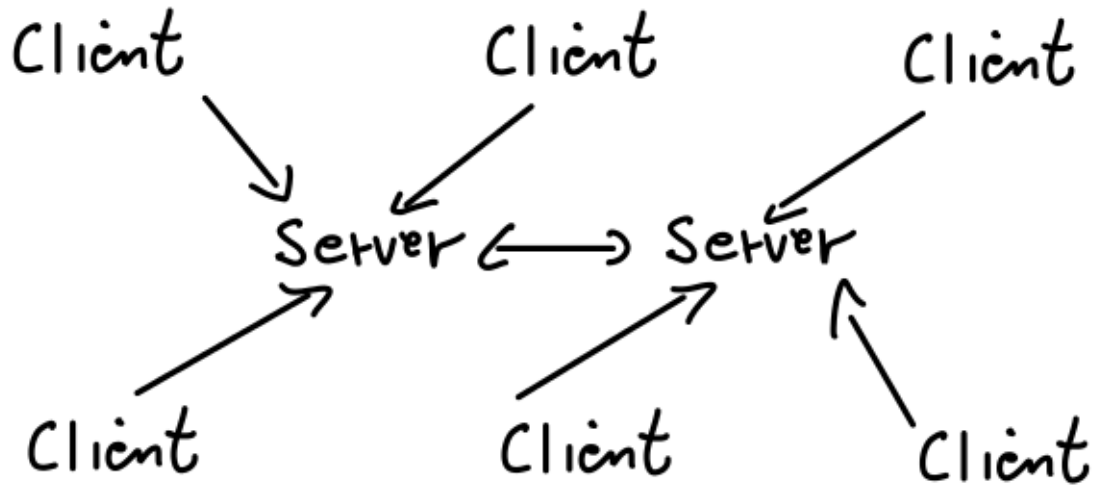
2.1.2 Architecture

IRC is organised centrally, as stated in [19]:

The IRC protocol provides no mean for two clients to directly communicate. All communication between clients is relayed by the server(s).

There is, however, an unofficial client extension named *Direct Client-to-Client (DCC)*[34, 3] available in most IRC clients that enables direct connections. Figure 2.1 shows the

Figure 2.1: Schematic Overview of IRC



schematic overview of an IRC network. IRC supports *direct chat* (one peer to another peer) as well as *group chat* (many peers to many peers). The group chat is realised by creating *IRC channels*, sometimes also called *chat rooms*.

2.1.3 Security

In general, all communications are based on a plaintext protocol. The TLS protocol [10] is being used for encryption in some networks, but its use is not standardised. Connections from clients to servers as well as servers to servers may be encrypted. Some networks support special channel modes to ensure all connected peers use an encrypted connection. There is no such mode for direct chat. For normal IRC channels, there is no guarantee that all peers are connected using an encrypted connection.

Due to this architecture, every server operator can listen to all messages, even if encryption is being used. As the network usually consists of many clients, but only a small number of servers, an attacker needs to concentrate on a small number of victim hosts only.

Usually IRC networks do not provide any kind of anonymisation features: every user can request detailed information about other users, including their IP address. The freenode network[14] however, supports so called *cloaks*, which transform the IP address information into a generic string like *gpm/telmich*, in which gpm describe the project and telmich the name. There are various other cloaking expressions available.

2.2 Secure Internet Live Conferencing (SILC)

2.2.1 History

The *Secure Internet Live Conferencing (SILC)* protocol has been developed by Pekka Riikonen since 1996. The first public release was made in 2000. The latest releases of SILC software (server and client) date back to 2009. SILC can be considered a more secure successor of IRC, as can be seen in the following quote:

... Many of the SILC features are found in traditional chat protocols such as IRC but many of the SILC features can also be found in Instant Message (IM) style protocols.

SILC combines features from both of these chat protocol styles, and can be implemented as either IRC-like system or IM-like system. ... [33]

As far as the official website reports, there is no version controlled source code repository available, only tarballs from 2009 are download-able. This may indicate that the development of the SILC software has stalled. Besides the official SILC Network, which is run by 7 SILC servers, there are about 12 other SILC networks being run.[51]

2.2.2 Architecture

Figure 2.2 shows that SILC network architecture looks similar to the IRC network architecture: Both rely on a central design orientated on a network of servers. In case of SILC, there are two different variants of message passing:

- Private Message Delivery With Session Keys
- Private Message Delivery With Private Message Key

In any case, all messages are travelling through supporting servers and are never transmitted directly from one to another client.

Private Message Delivery With Session Keys

When the SILC client uses session keys, every message is send encrypted with the session key of the next peer, decrypted at the peer and re-encrypted for the next peer in the chain, as can be seen in figure 2.3. In this scenario, every peer in the communication chain can read the content of the message.

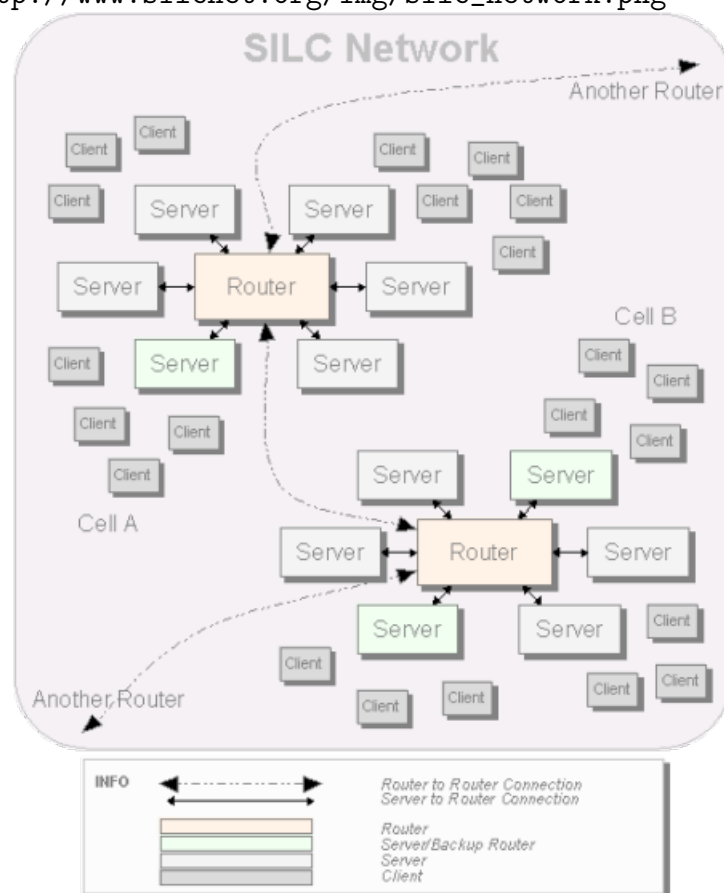
Private Message Delivery With Private Message Key

When using a a private message key, the servers in the communication chain only forward the message and the receiving peer is the only one who can decrypt the message. Figure 2.4 illustrates this behaviour.

2.2. SECURE INTERNET LIVE CONFERENCING (SILC)

Figure 2.2: Schematic Overview of SILC

Image source: http://www.silcnet.org/img/silc_network.png



2.2. SECURE INTERNET LIVE CONFERENCING (SILC)

Figure 2.3: Private Message Delivery With Session Keys

Image source: http://www.silcnet.org/img/silc_priv1.png

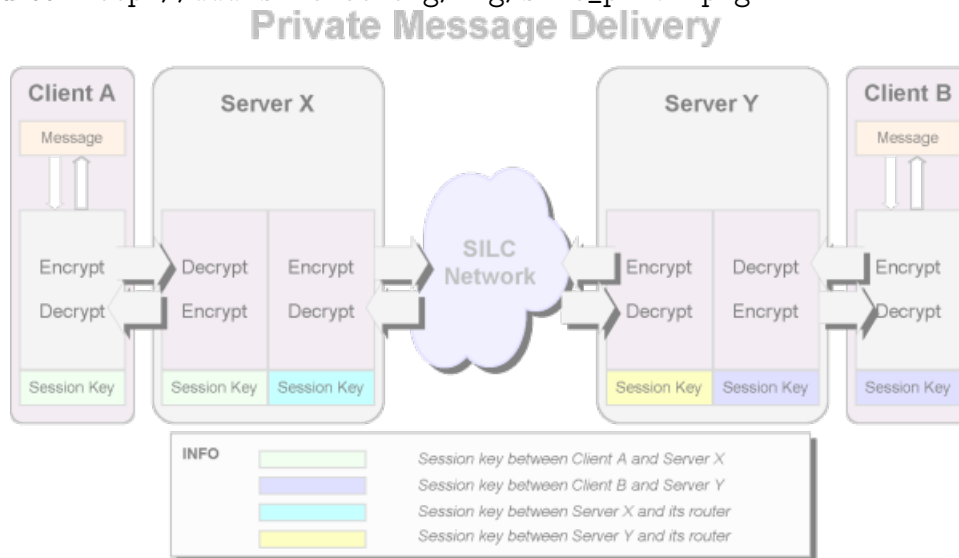
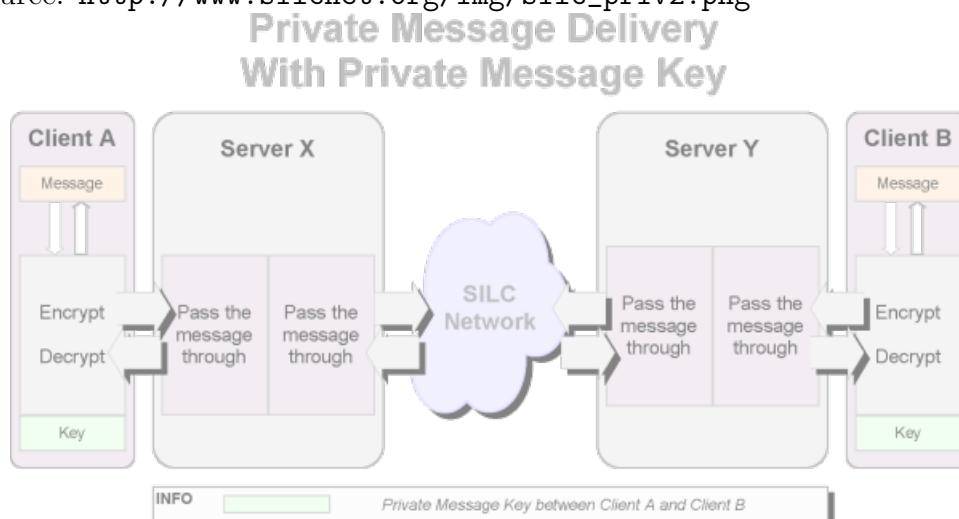


Figure 2.4: Private Message Delivery With Private Message Key

Image source: http://www.silcnet.org/img/silc_priv2.png



2.2.3 Security

As all messages in the SILC network are encrypted, an external attacker cannot read the message content. In case private message delivery with session keys is utilised, a compromised SILC server may be used to intercept and read all messages. This is not possible, if messages are sent using private message delivery with private message key. Due to the small number of servers (7 in the official network), an attacker could run a *Denial-of-Service (DoS)* attack to prevent users from chatting.

The SILC network does not provide any kind of anonymity services and allows to query detailed information about a peer including the IP address.

2.3 Extensible Messaging and Presence Protocol (XMPP)

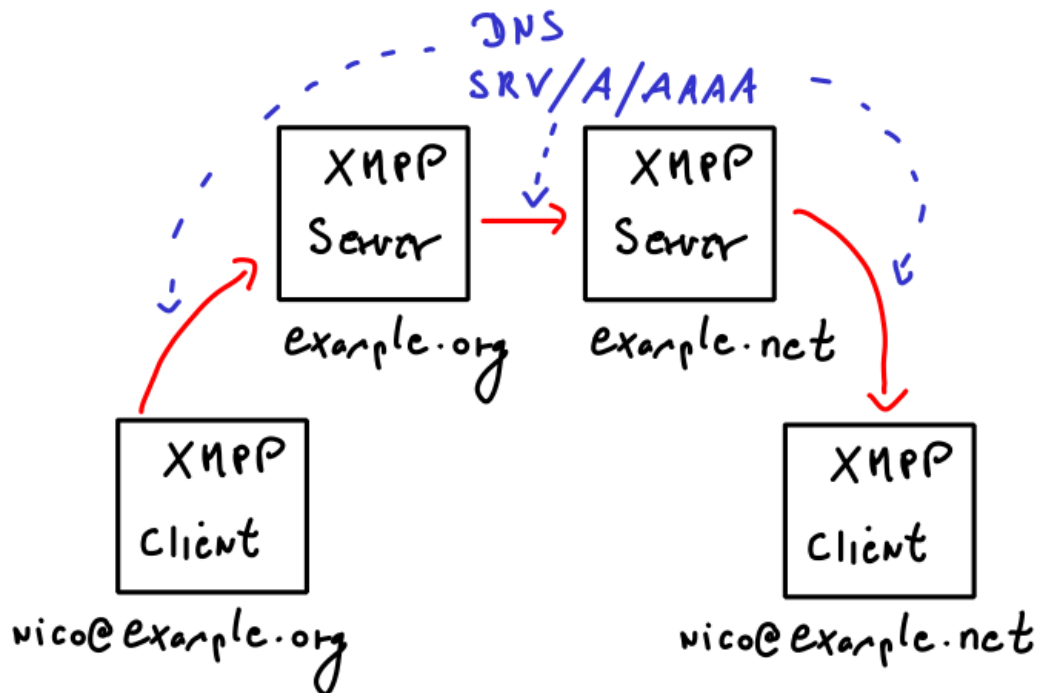
2.3.1 History

The *Extensible Messaging and Presence Protocol (XMPP)* was originally developed by Jeremie Miller in 1998 and released to the public as *Jabber* in 1999. In 2004 Jabber was transferred to the IETF, renamed to XMPP and is now defined in several RFCs[36, 37, 38, 35, 39, 40, 23, 41, 42]. Since then, the *XMPP Standards Foundation (XSF)* is responsible for standardising XMPP related protocols, the so called *XMPP Extension Protocols (XEP)*. By 2003 XMPP has been used by an estimated 10 million people[13]. AOL added experimental support for XMPP in the *AOL instant Messenger (AIM)* in 2008. In 2010 the social-networking site Facebook added support for third-party applications via XMPP. As of 2012, there are over 30 different client and about 20 server implementations available.

2.3.2 Architecture

The XMPP network is built upon the ideas of e-mail: Users are identified by e-mail like usernames like *nico@example.org*. The clients and servers find the correct server to talk to by making a DNS SRV[15] look-up. Clients are asking for the *xmpp-client* service and servers are asking for the *xmpp-server* service. Thus a client may ask for a SRV record `_xmpp-client._tcp.example.net.` or a server for `_xmpp-server._tcp.im.example.com`. If there is no SRV record, the client or server can fall back to request for the A or AAAA record, similar to how SMTP[1] behaves. Figure 2.5 shows how the client *nico@example.org* sends a message to *nico@example.net*. XMPP formats messages based on the *Extensible Markup Language (XML)* and supports authentication using the Simple Authentication and Security Layer (SASL)[24]. Clients never talk directly to each other in a XMPP network, but always use one or more servers in between. Compared to IRC and SILC, XMPP is the only protocol that supports connecting other protocols like AIM or IRC.

Figure 2.5: XMPP Network Architecture



2.3.3 Security

Within XMPP, TLS is being used for encryption and SASL for authentication. XMPP does not have a mechanism to support anonymity. To create a DoS attack, an attacker can choose to take either the sending or the receiving server down. Due to its openness, an attacker can find out which server needs to be attacked by doing DNS queries. Though when attacking the sending server, the sending client can just switch to another server, as figure 2.6 shows. If the attacker aims to bring down the receiving server, the message will not arrive while the server is down, as shown in figure 2.7.

2.4 Skype

2.4.1 History

The following excerpt[52] gives a rough overview of Skype's history:

Skype was founded in 2003 by Janus Friis from Denmark and Niklas Zennström from Sweden. The Skype software was developed by Estonians Ahti Heinla, Priit Kasesalu, and Jaan Tallinn, who together with Janus and Niklas were also behind the peer-to-peer file sharing software Kazaa. In August 2003, the first public beta version was released.

Figure 2.6: XMPP Sending Server Attack

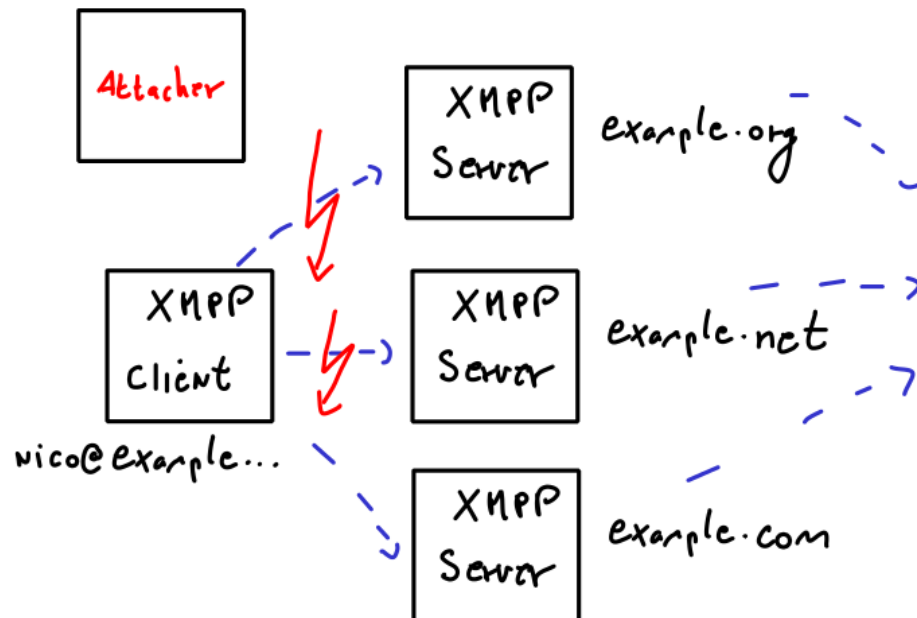
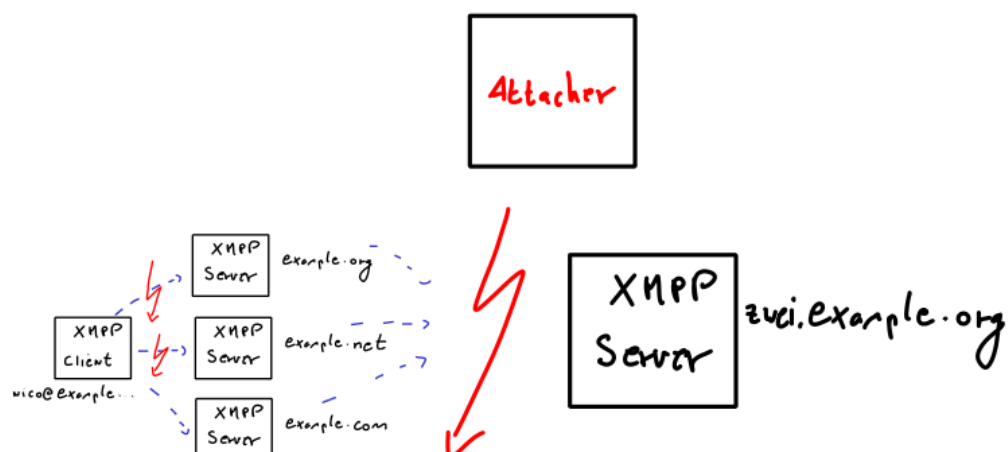


Figure 2.7: XMPP Receiving Server Attack



In April 2006, the number of registered users reached 100 million. In July 2010 several online newspapers announced that a hacker from the USA has successfully managed to reverse engineer the protocol. The suspected results can be found on [46]. In May 2011 Microsoft acquired Skype. There are various efforts made to keep up the reverse engineering of the program and the protocol, some driven by Russian scientists. During the research of this thesis multiple websites referencing protocol details have been made inaccessible, some of them were replaced by a website that claimed that the content is prohibited by the Digital Millennium Copyright Act (DMCA). Skype as of today is being used on many computers and mobile phones.

There is also a variant of Skype available called TOM, which has been produced for the Chinese market after Skype's initial banning from the China.

2.4.2 Architecture

Skype's network is a mixture of a centralised and peer-to-peer network. As figure 2.8 shows, it consists of three entities:

- Supernodes
- Ordinary Nodes
- Login Servers

The function of these three nodes is explained in [2]:

An ordinary host is a Skype application that can be used to place voice calls and send text messages. A super node is an ordinary host's end-point on the Skype network. Any node with a public IP address having sufficient CPU, memory, and network bandwidth is a candidate to become a super node. An ordinary host must connect to a super node and must register itself with the Skype login server for a successful login. Although not a Skype node itself, the Skype login server is an important entity in the Skype network.

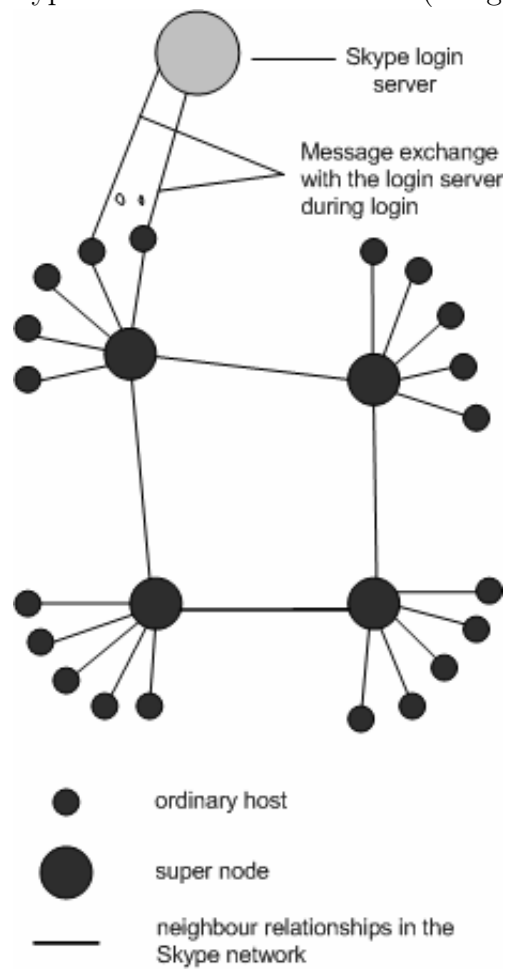
In case a client is behind a firewall that does *network address translation (NAT)*, Skype uses a variant of STUN[28]. In contrast to the previous three protocols, Skype use is more focused on voice chat. The connection is encrypted using AES-256, the keys are encrypted using 1536 up to 2048 Bit RSA. Furthermore, variants of RC4 are being used, which may be attackable. The website describing this attack is not reachable anymore, but is still quoted on Wikipedia.¹

2.4.3 Security

Although Skype makes use of secure encryption methods (AES, RSA), it has various other security threads: The software itself is programmed to detect debugging scenarios and

¹<http://www.enrupt.com/index.php/2010/07/07/skype-biggest-secret-revealed>

Figure 2.8: Skype Network Architecture a (Image source: [2])



contains encrypted code. This may lead to the assumption that the Skype developers are trying to hide information from the user. Although Skype is reported to support encryption between the two communicating peers[30], at least the Chinese variant TOM supports surveillance operations:[49]

The full text chat messages of TOM-Skype users, along with Skype users who have communicated with TOM-Skype users, are regularly scanned for sensitive keywords, and if present, the resulting data are uploaded and stored on servers in China.

These text messages, along with millions of records containing personal information, are stored on insecure publicly-accessible web servers together with the encryption key required to decrypt the data.

The captured messages contain specific keywords relating to sensitive political topics such as Taiwan independence, the Falun Gong, and political opposition to the Communist Party of China.

Our analysis suggests that the surveillance is not solely keyword-driven. Many of the captured messages contain words that are too common for extensive logging, suggesting that there may be criteria, such as specific usernames, that determine whether messages are captured by the system.

As the non-TOM Skype binary itself is encrypted, it is possible that it contains similar functions. Another uncertainty is caused by statements of government agencies, which claim to be able to intercept all Skype traffic. Skype is treated similarly as traditional phone companies, which indicates truth of the given statements. Furthermore recent reports indicate that Microsoft changes the supernode structure so that instead of having supernodes at the user, the supernodes reside centrally at Microsoft's datacenters to allow easy wiretapping.[47]

2.5 Other

There are further chat protocols, which have not been included into this analysis, either because of known weaknesses, irrelevance or small user base. ICQ, which has been published in 1996, is not being widely used anymore and its protocol, OSCAR[44], is based on plain text messages. ICQ used to be popular, but has been sold to the Russian Mail.ru group. Furthermore the old *Microsoft Messenger protocol (MSN)* has not been taken into account.

2.6 Security features and Comparison

The different chat systems and their architectures have shown a variety of strengths and weaknesses, which are summarised in the table below.

2.6. SECURITY FEATURES AND COMPARISON

Table 2.1: Chat system comparison with security features

Name	IRC	SILC	XMPP	Skype
Central server architecture	yes	yes	no	yes/no ²
Encrypted messages	optional	yes	yes	yes
Anonymity Support	yes/no ³	yes/no ⁴	no	no
Proprietary Protocol	no	no	no	yes
Software owned by one (commercial) company	no	no	no	yes
Network being run by one (commercial) company	no	no	no	yes
Source Code accessible	yes	yes	yes	no
Encrypted binary software	no	no	no	yes

²Supernodes take a special role and are being centralised by Microsoft's work.

³IP address cloaking

⁴IP address cloaking

Chapter 3

Analysis of related communication protocols

In this chapter I will outline the related communication protocols. The Mix Networks, Tor and Freenet all focus on providing anonymity to the user, which slightly different approaches. I2P seems to be another interesting anonymity enhancing service, which is only briefly discussed due to the lack of documentation. A completely different approach for security and indirectly for anonymity is taken by Off-the-Record Messaging (OTR), which focuses on short time keys. At the end of this chapter RUDP is shown, which aims to provide reliable communication over an unreliable link. Due to the different fields of application of the shown protocols, a comparison is omitted.

3.1 Mix Networks

Mix networks are also known as onion routing or digital mixes and were invented by David Chaum in 1981.[7] The base of Mixes is the multiply encrypted packet that sent through a number of peers (the mixes) as shown in figure 3.1. Every peer decrypts the message, which reveals which is the next peer to send the packet to. Using this method, every peer only knows about its predecessor and successor. For encryption Public-Key-Cryptography (RSA) is used. The original approach was focusing on Mixes for e-mail delivery.

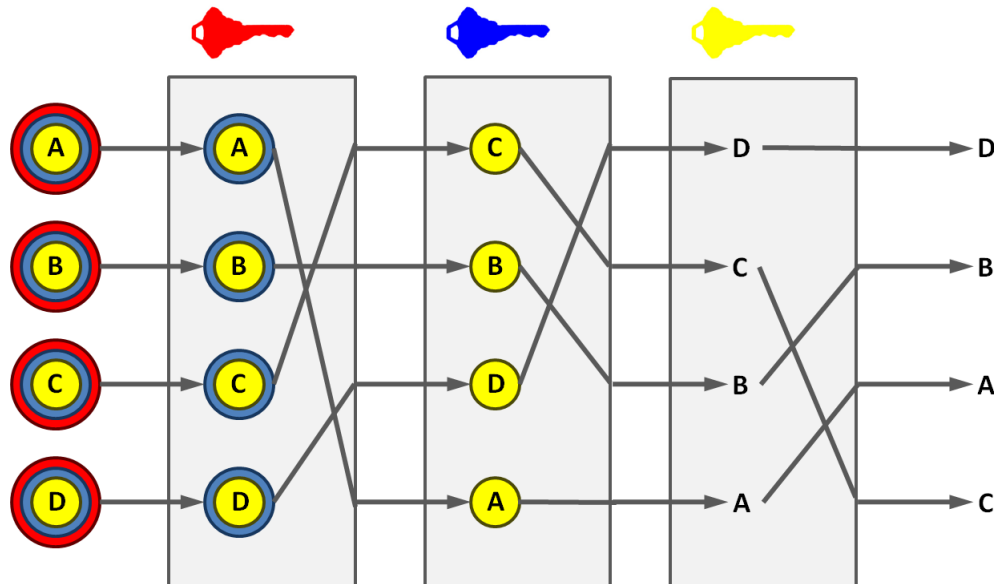
There are also various projects, which do not try to implement generic packet sending, but focus on sending only e-mails anonymously. Projects like Mixmaster[48], Babel[16] or Mixminion[9] implement this technology.

3.2 Tor

The Tor network has been published in 2004 at the 13th USENIX Security Symposium by Roger Dingledine, Nick Mathewson and Paul Syverson.[11] The idea behind Tor is to provide a general purpose, low latency anonymising network stack. They explicitly do not include application level anonymity (like stripping user agent information from a http

Figure 3.1: Mixes: Packet Flow

Image source http://upload.wikimedia.org/wikipedia/en/2/23/Decryption_mix_net.png



request) and reason this to leave it up specialised projects. Tor does (by design) offer no protection against end-to-end attacks (for instance using traffic analysis) to support low latency applications. The Tor software provides access to its network via a SOCKS proxy interface. Tor does not use the traditional mix approach:

Rather than using a single multiply encrypted data structure (an onion) to lay each circuit, Tor now uses an incremental or telescoping path-building design, where the initiator negotiates session keys with each successive hop in the circuit.¹

3.3 Freenet

Freenet has been published in 2001 by Ian Clarke , Oskar Sandberg , Brandon Wiley and Theodore W. Hong.[8] Its primary use the anonymous storage (and retrieval) of files. For this reason, it is not directly related to the requirements found in a chat system. Freenet is described as following:

Freenet is free software which lets you anonymously share files, browse and publish "freesites" (web sites accessible only through Freenet) and chat on forums, without fear of censorship. Freenet is decentralised to make it less

¹Quote from [11].

vulnerable to attack, and if used in "darknet" mode, where users only connect to their friends, is very difficult to detect.²

3.4 I2P

The I2P project, originally Invisible Internet Project, is supposed to be "a scalable framework for anonymous communication".[18] It is described on the website as following:

I2P initially began in Feb 2003 as a proposed modification to Freenet to allow it to use alternate transports, such as JMS, then grew into its own as an 'anonCommFramework' in April 2003, turning into I2P in July, with code being written in earnest starting in August '03. I2P is currently under development, following the roadmap.³

...

I2P is a scalable, self organizing, resilient packet switched anonymous network layer, upon which any number of different anonymity or security conscious applications can operate. Each of these applications may make their own anonymity, latency, and throughput tradeoffs without worrying about the proper implementation of a free route mixnet, allowing them to blend their activity with the larger anonymity set of users already running on top of I2P.⁴

Although the overall impression of the website indicates that it may provide a reasonable base for anonymous communications, the lack of a central architecture or design paper makes it hard to judge about it.

3.5 Off-the-Record Messaging (OTR)

Off-the-Record Messaging (OTR) has been introduced by Nikita Borisov, Ian Goldberg and Eric Brewer in 2004.[5] In contrast to other approaches OTR does not rely on long term keys as usually used in Public-Key-Cryptography. OTR instead uses short time keys, to prevent decryption of sniffed messages, if access to the private key is gained after some time.

Furthermore OTR includes measures to include repudiability, though provide authentication during a session using message authentication codes (MAC). OTR works as a drop in for existing chat programs and offers auto-detection if the other side is capable of using OTR as well.

²Quote from <https://freenetproject.org/whatis.html>

³Quote from http://www.i2p2.de/how_intro.

⁴Quote from <http://www.i2p2.de/techintro.html>.

3.6 Reliable UDP (RUDP)

Reliable UDP was specified in an internet draft at IETF written in 1999 by T. Bova and T. Krivoruchka.[45] The relevant features RUDP enhanced UDP[31] with are:

- Acknowledgement of received packets
- Retransmission of lost packets

RUDP relates to this thesis, because it provides a simple method to create reliability over an unreliable link. RUDP is featured by sequence numbers and a synchronisation at communication begin similar to TCP.[32] RUDP ensures reliability by providing a resend mechanism, which may be adjusted to resend messages infinitely until the the message has been acknowledged from the peer. Due to the use of sequence numbers packet duplication can be detected with RUDP as well.

Chapter 4

Analysis of features and security requirements

While in chapters 2 and 3 the chat systems and the related communication protocols are described, this chapter summarises the relevant features and security requirements and describes, if the given topic is taken care of in the proposed chat protocol. The implementation details of the features are covered in section 5.10.

4.1 Chat features

4.1.1 Multi User Chat (MUC)

Several chat systems support the multi user chat, in which one user sends a message which is received by a group of recipients. This thesis focuses on direct chat (1:1) and does not support multi user chat.

4.1.2 File Transfer

Some chat systems support sending and receiving of files in the protocol. As file transfer is just a special method of message sending and can be added by another protocol layer, this chat protocol does not explicitly support file transfer.¹

4.1.3 Voice Communication

Voice communication as seen in Skype is not the focus of the thesis and for this reason not supported.

¹File transfer could easily be implemented on top of the chat protocol by encoding and splitting files into chat messages and marking them with a special keyword.

Figure 4.1: Sender Anonymity

Image source: <http://www.cs.virginia.edu/crab/anonymity.ppt>

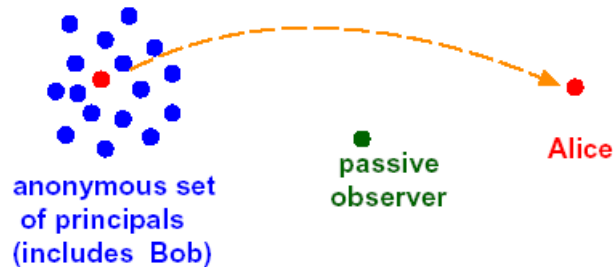
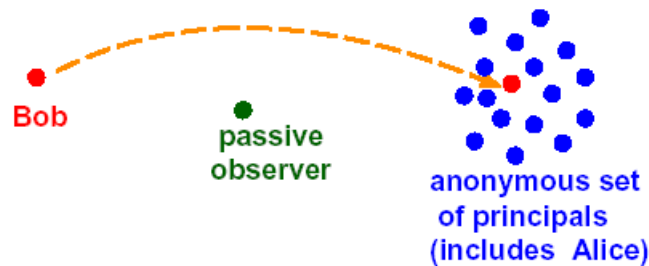


Figure 4.2: Receiver Anonymity

Image source: <http://www.cs.virginia.edu/crab/anonymity.ppt>



4.2 Security Requirements

4.2.1 Anonymity

There are four different types of anonymity:

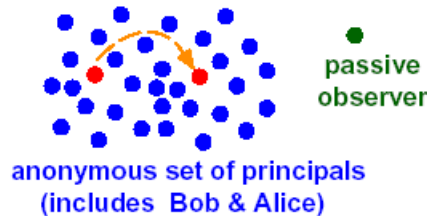
- Pseudonymity
- Sender anonymity
- Receiver anonymity
- Sender-receiver anonymity

Pseudonymity describes anonymity by using a different identity, for instance *telmich* instead of Nico Schottelius. Sender anonymity is present, if nobody can observe who the sender is. Receiver anonymity is given, if an observer cannot identify the receiver of a message. If the observer cannot find out whether a given pair of peers are communicating, Sender-Receiver anonymity is given. The three figures 4.1, 4.2 and 4.3 visualise the differences.

4.2. SECURITY REQUIREMENTS

Figure 4.3: Sender-Receiver Anonymity

Image source: <http://www.cs.virginia.edu/crab/anonymity.ppt>



Although a lot of communication protocols address the challenge to provide a level of anonymity, this topic has not been covered by the chat systems. On the other hand, there are various attempts to allow chat systems to be run in an anonymity network (like I2P), though this requires the user, and the peer she wants to talk to, to install and use the full fledged anonymity network.

The aim of this thesis is to close this gap, to provide a chat system that provides anonymity, without requiring an additional network stack to be present.

4.2.2 Confidentiality

Confidentiality is defined as allowing only authorised parties to access a given set of information. Besides hiding the real content of a conversation, confidentiality indirectly assists anonymity, because it is not possible to derive who is talking from the content.

For this reason all messages that are sent using the defined chat protocol have to be treated as confidential.

4.2.3 Authenticity

Authenticity ensures that

1. the message content has not been modified (integrity is consistent) and that
2. the receiver can verify that the message has been sent by the right person

Both requirements should be supported by the provided chat protocol.

4.2.4 Availability

Availability in a chat system is important in two directions:

1. Ensure that receiving messages is possible
2. Ensure that sending messages is possible

4.2. SECURITY REQUIREMENTS

Most traditional systems rely on central infrastructure to operate, in which a single party (such as the operator) can disable the service, either for an individual participant or for the whole network. Even if a decentralised architecture is given, no service can be run reliably, if an attacker with infinite resources is assumed.

Thus the requirement for this chat system is to survive a single denial of service attack, while continuing to be able to send and receive messages.

Chapter 5

Chat Protocol Definition

5.1 Version

This is the **first version** of the EOF chat protocol definition. Future versions may increment the version number to allow changes to the protocol.

5.2 Basic Data Types ("EOFbdt")

This section specifies the basic `datatypes`. They are further referenced as "EOFbdt".

5.2.1 The zero byte

The zero byte is a byte with the value 0.

5.2.2 ASCII numbers

ASCII numbers use the decimal string representation of a number. ASCII numbers are often used in a packet header. ASCII numbers are used to specify the length of the packet (excluding itself). Due to compatibility of UTF-8 and ASCII, ASCII numbers may also be referred to as *UTF-8 numbers*.

5.2.3 Strings in general

Strings are transmitted without termination (i.e. no new line, no 0 byte). The encoding to be used is **UTF-8**.

5.2.4 Fixed length strings

Fixed length strings contain exactly the specified number of bytes: A 128-byte fixed length string consists of at most 128 bytes of text. If the text it contains is shorter than the specified length, it must be padded with zero bytes.

5.2.5 Variable length strings

This protocol does not specify any variable length strings.

5.3 Simple Data Types ("EOFsdt")

The following sections define the simple datatypes. They are further referenced as "EOFsdt".

5.3.1 Command (command)

A command is represented as an ASCII number in a fixed length string of 4 bytes. It is used to identify the intent of a message.

Examples

- 1100
- 3000
- 2200

5.3.2 Version (version)

To identify which protocol version has been used, an ASCII number in a fixed length string of 2 Bytes is used. The version field must be "0" followed by a zero byte. Example:

- 0\x00

5.3.3 Identification string (id)

To identify a message, a message may contain an identification string, called the *EOFID*. This ID is an integer that is encoded based on the following characters:

- A-Z (alphabet in upper case)
- a-z (alphabet in lower case)
- 0-9 (the digits)
- ! (exclamation mark)
- - (minus)

The order of the characters is as follows:

{0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-!}.

The length of an EOFID is 6 bytes, which results in *68719476736* possible ids.¹. The given characters were selected to allow easy debugging.

¹ $(10 + 26 + 26 + 2)^6$

Examples

The following examples encode and decode integers into the specified format. Use is made of the Python prototype implementation:

```
>>> import ceof
>>> ceof.EOFID.int_to_id(42)
'000000G'
>>> ceof.EOFID.int_to_id(1)
'0000001'
>>> ceof.EOFID.int_to_id(64)
'000010'
>>> ceof.EOFID.id_to_int('000010')
64
>>> ceof.EOFID.id_to_int('!!!!!!')
68719476735
>>> ceof.EOFID.id_to_int('a-----')
11794116542
>>> ceof.EOFID.id_to_int('000000')
0
```

5.3.4 Peer name (name)

The peer name is a 128 byte fixed length string.

Examples

[illegible]

5.3.5 Group name (group)

The group name is a 128 byte fixed length string. It is currently not being used but reserved for future support of multi user chat.

Examples

[illegible]

[illegible]

5.3.6 Message text (msgtxt)

The message text is a 256 byte fixed length string.

Examples

[illegible]

5.3.7 Peer address (address)

The address of a peer, which is a 128 byte fixed length string. Peer addresses are specified as URLs as defined in RFC3986[4].

Examples

[illegible]

5.3.8 Peer fingerprint (keyid)

A (PGP) fingerprint² is a 40 byte fixed length string. As the fingerprint has a fix length of 40 bytes, there is never padding needed.

Examples

```
% gpg --fingerprint | grep "Key fingerprint =" | sed -e 's/.*=/' -e 's/ //g'
A35767A98CA9CC3CE368679AB679548202C9B17D
```

²See RFC 2440[6], 11.2. Key IDs and Fingerprints

5.4 Messages Types ("EOFmsg")

The following message types are defined:

- Drop packet (command: 3000)
- Forward packet (command: 3001)
- Message / drop packet (command: 3002)
- Message / forward packet (command: 3003)
- Acknowledge (command: 3004)

5.4.1 Parameter Overview

All messages have the same length and contain the same fields. Though not all fields are being used in every type, as seen in table 5.1.

Table 5.1: EOF Message Parameter Usage

Command	version	id	addr	group	msgtext
3000	x	-	-	-	-
3001	x	-	x	-	-
3002	x	x	-	-	x
3003	x	x	x	-	x
3004	x	x	-	-	-

Where

- - means **not used**
- and x means **used**.

In table 5.2 the messages parameters are described.

Table 5.2: EOF Message Parameter Description

Parameter	Type	Description	Example
version	EOFsdt	EOF Version	0
id	EOFsdt	Packet id	alg4f!
addr	EOFsdt	Address of next peer	tcp://123.123.123.132:8080
group	EOFsdt	The destination group	!eof
msgtext	EOFsdt	The message	Hallo, mein Freund!

5.4.2 3000: Drop packet

You are the last recipient and there's nothing interesting left. Just drop the packet and continue work.

5.4.3 3001: Forward packet

If a peer receives a packet with the command 3001, it simply forwards the message to the peer specified in the **addr** field. All data contained in the message is noise. After the message has been forwarded to the next peer, it should be dropped. If the peer is unreachable, the message should also be dropped.

5.4.4 3002: Message / drop packet

This packet contains a messages to be read and does not need to be forwarded anymore: You are the last peer in the chain.

5.4.5 3003: Message / forward packet

The command 3003 is a combination of command 3001 and 3002 and instructs the peer to read the message text and to forward the rest of the packet to the specified peer in the **addr** field.

5.4.6 3004: Acknowledge

Acknowledge the receipt of a received message. The ID must be the same as the one specified in the original messages packet. Every message packet must be acknowledged.

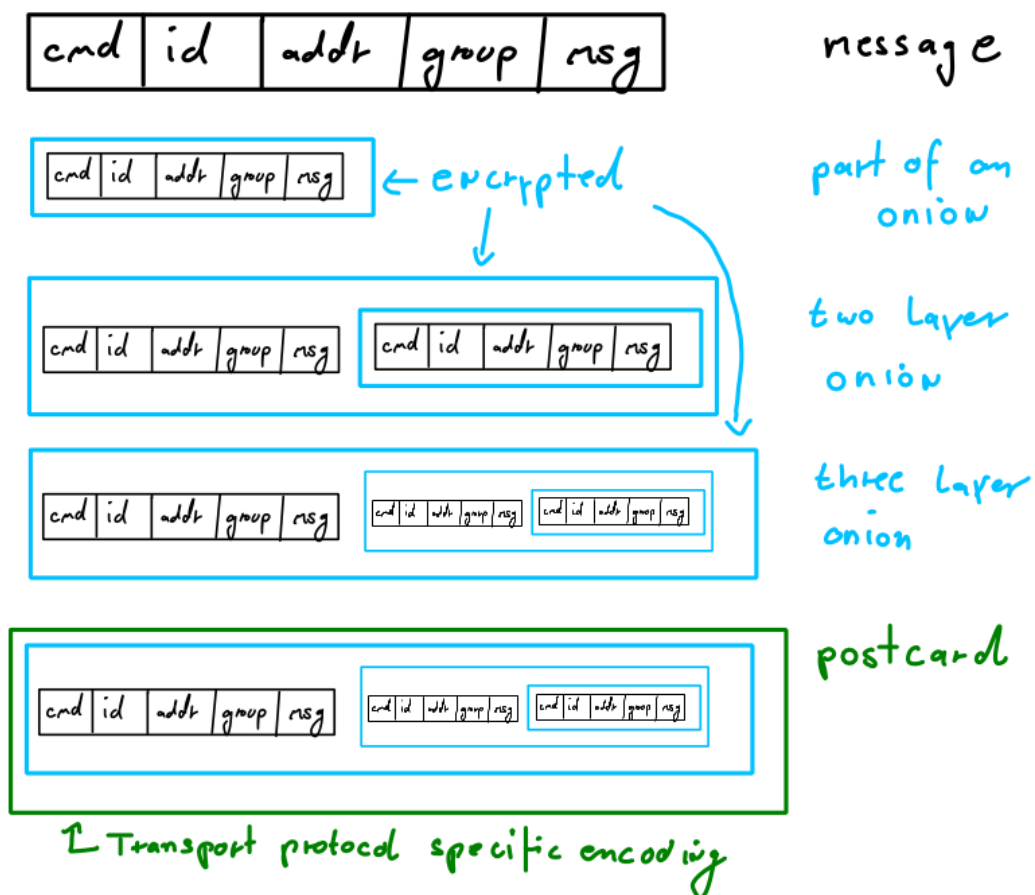
5.5 Packets Type Overview ("EOFpkg")

Based on the previous definition of data types and messages, one can distinguish between the following packet types:

- Messages: (internal) plaintext packets (5.4, p. 37)
- Onions: multiply encrypted packets (5.8.2, p. 45)
- Postcards: packets containing transport protocol dependent header (5.7.1, p. 41)

Messages are the innermost packet type and can only be seen within the implementation. Messages are then bundled into a multi layer onion. Each layer contains messages after decryption. Onions are put onto a *postcard packet* afterwards and are sent out on the network. Figure 5.1 visualises the differences between the different packet types.

Figure 5.1: Packet Types



5.6 OpenPGP

Message encryption and digital signatures are used as defined in the OpenPGP RFC.[6] All messages are encoded into ASCII armor as specified in section 6.2 of the RFC, to allow easy encapsulation into any transport protocol.

5.7 Transport Protocols

Transport protocols are used to wrap an onion into a postcard and to submit the postcard to the next peer. This chat protocol does not rely on a specific underlying transport protocol, it is *transport protocol independent*. It does however define which transport protocols can be used to ensure interoperability.

5.7.1 Network packets ("postcards")

A postcard "packet" contains one onion packet plus the transport protocol shell. Postcard packets are the only packet type that is seen by a possible attacker ("on the wire"). The name postcard was chosen to reflect the fact that anyone passing the postcard can read what is written on it.

5.7.2 Tunnelling

Due to the transport protocol independence, postcards can and should be written into a variety of different transport protocol types. This helps to circumvent firewall rules and blocking of a specific traffic type. A lot of potential can be found in the reuse of common protocols like HTTP and DNS.[25, 26, 12] Figure 5.2 shows how four different onions could be transmitted via four different transport protocols.

5.7.3 Multiplexing / Variable Addresses

One peer should consider the availability of itself via a variety of addresses, so the chance of not being reachable is minimised. Figure 5.3 shows example addresses of a peer. This multiplexing of addresses allows different transports to be used for one peer, so that there is a higher diversity in outgoing packets.

5.7.4 Access Methods

Transport protocols can either be accessible *directly* or *indirectly*. In case of direct access (figure 5.4) the sending peer directly connects to the receiving peer. In case of indirect access, the sender stores the postcard on a intermediate server and the receiving peer polls this server for postcards, as shown in figure 5.5.

5.7.5 List of Supported Transports

To ensure interoperability, clients which support a specific protocol version must support all listed transport protocols. This version supports all protocols specified in table 5.3.

Table 5.3: Transport protocols

Protocol	Description	Supported versions
tcp	Transmission Control Protocol	0 - 0

Figure 5.2: Transport Protocol Tunneling

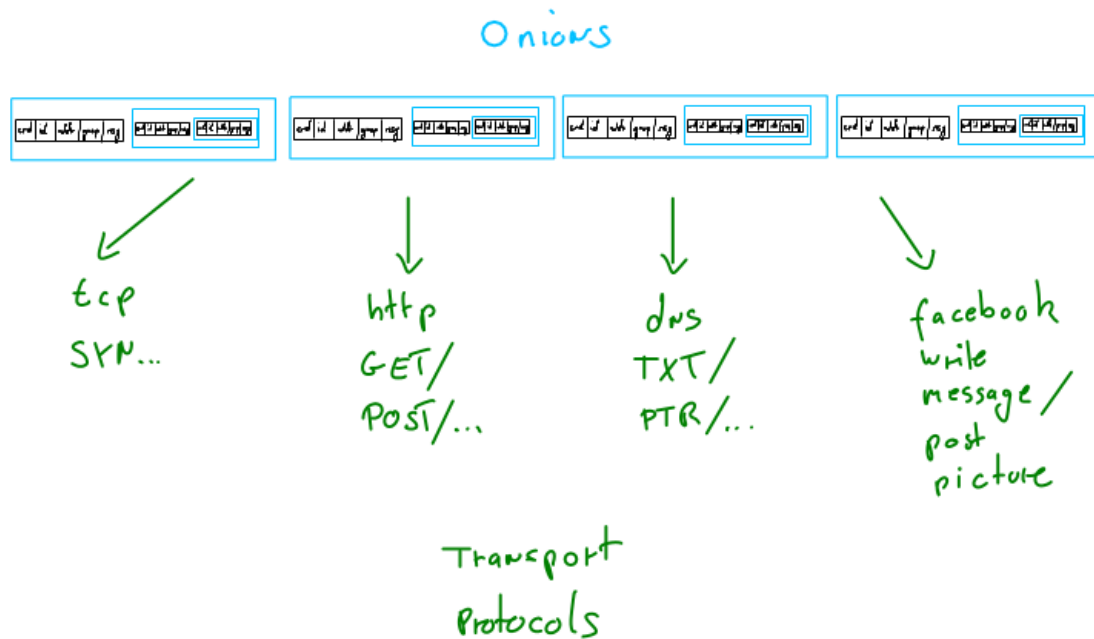


Figure 5.3: Address Multiplexing

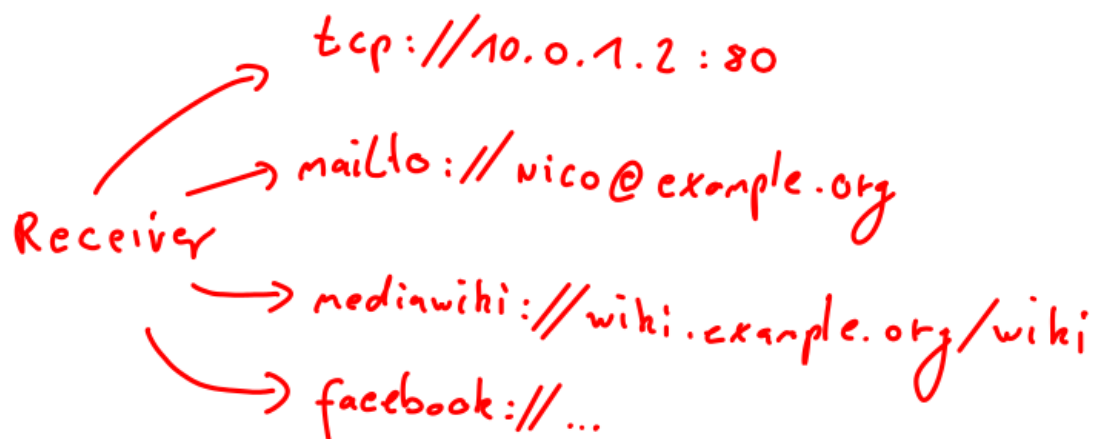


Figure 5.4: Direct Access

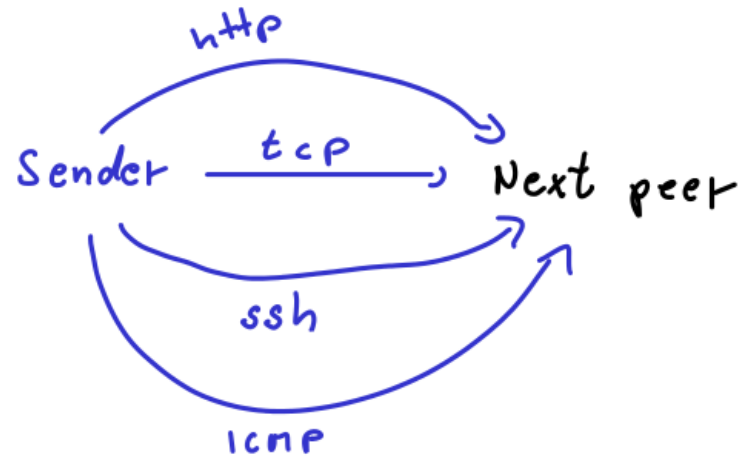


Figure 5.5: Indirect Access

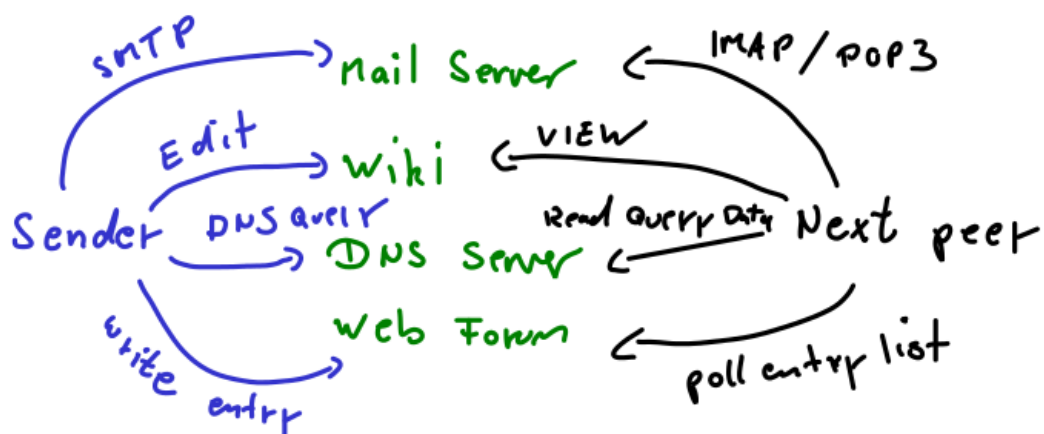
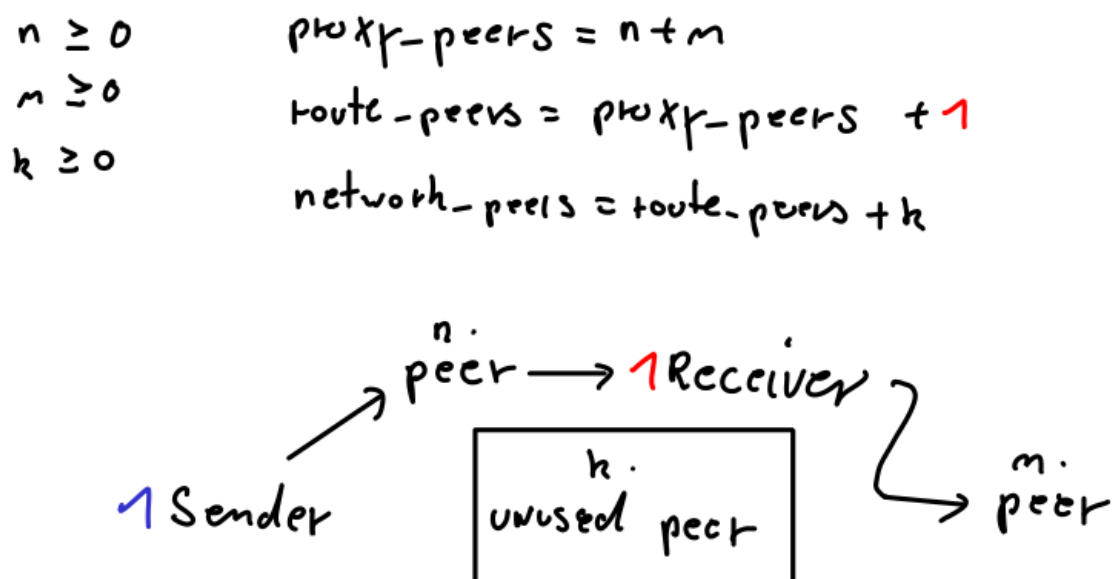


Figure 5.6: Onion Routing



5.8 Onion Routing

Onion routing is used to ensure sender-receiver anonymity. In contrast to the Tor network, there are no exit nodes and the receiver can be anyone in the onion chain (see fig. 5.6). The peers between the sender and receiver and after the receiver are called *proxy peers*. The number of proxy peers between the sender and the receiver and the number of proxy peers after receiver can range from 0 up to the maximum number of proxy peers. The sum of both proxy peers is the *chosen number of proxy peers* and usually stays consistent for one peer.

The recommended number of proxy peers is 5, as a trade-off between latency, bandwidth and anonymity. Section 5.9, p. 47 explains in detail why 5 is the recommended number.

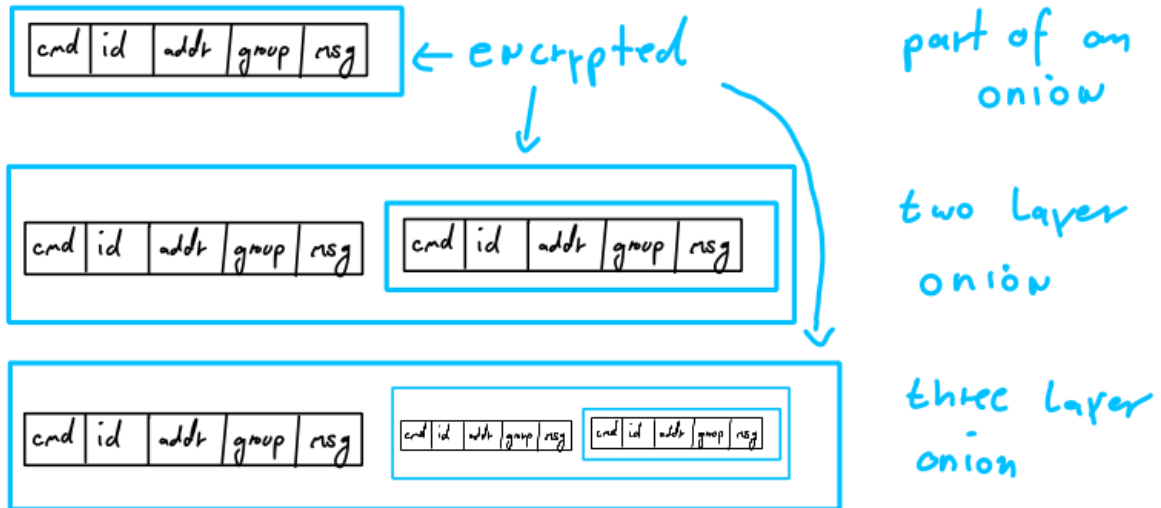
5.8.1 Source Based Routing

Before creation of an onion, a random route to the receiver is generated. The route is calculated as follows:

1. Select random peer from known peer list, repeat for the number of required proxy peers
2. Shuffle the list and insert the receiver at a random position
3. Retrieve a random transport protocol address from each peer

For each onion, a new random routes must be calculated.

Figure 5.7: An Onion (2 and 3 layers)



5.8.2 Onions

Onions build the base for the EOF protocol. The EOF messages described in section 5.4 are multiple times encrypted and assembled according to the previously calculated random route. The result is called an *onion* and an example onion is shown in figure 5.7. Every layer of an onion is encrypted for the specific peer and the previous encrypted layer is appended after the EOF message in re-encrypted. In case the onion layer contains the EOF message 3002 or 3003, the layer should also be signed. In all other cases the onion layer should only be encrypted. The following pseudo algorithms shows how an onion is created:

1. Create EOF message for the last peer
2. Encrypt EOF message for the last peer (first and innermost onion layer)
3. Redo steps one and two for every peer, but append the previous result

5.8.3 Packet Sizes

The packet size depends on the number of peers an onion was encrypted for. A list of average packet sizes can be found in table 5.4. It was generated by running the reference implementation (`ceof onion -m "test" peer0 | wc -c`) and increasing the number of proxy peers to be inserted. The resulting size includes the final onion, but does not include transport protocol headers.

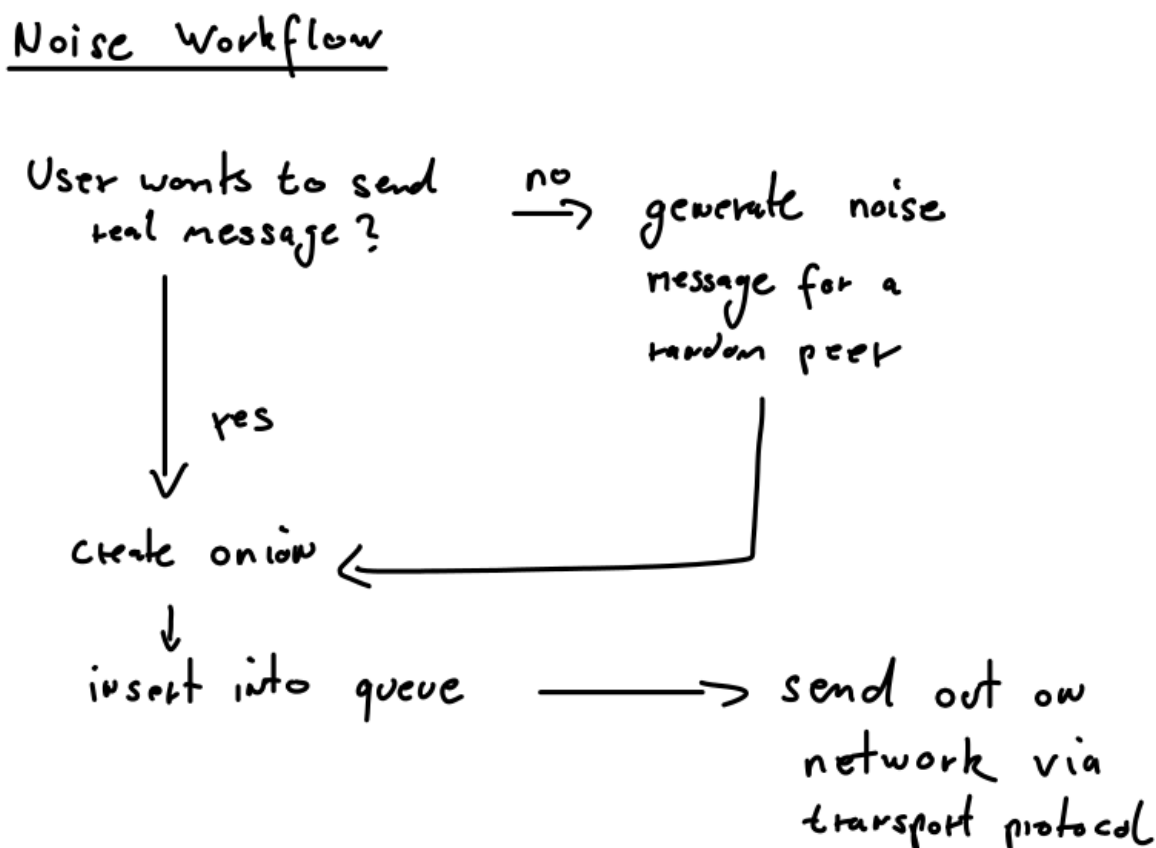
Table 5.4: Packet sizes (experimental)

Proxy peers	Packet size (in KiB)
1	1.2
2	1.9

5.8. ONION ROUTING

3	2.6
4	3.3
5	4.0
6	4.8
7	5.6
8	6.3
9	7.1
10	8.0

Figure 5.8: Noise Workflow



5.9 Noise

Noise is being used to constantly send out data on the network. This prevents statistical analysis of the behaviour of a user and makes de-anonymisation more impractical due to the amount of data that needs to be tracked and analysed. Furthermore, noise is used to fill in the unused fields in the EOF messages. Noise can be any type of random data. As the current random number generators are quite expensive, it is recommended to use a dictionary like old messages, log files, public emails, etc. for noise input. The reference implementation uses the archive of RFCs as base for noise. The noise workflow is shown in figure 5.8.

5.9.1 Latency

It is important for the user experience of the chat system to minimise latency. This is especially important after the initial *human handshake* of saying hello has been made (before that nobody knows when to expect a message). The paper about *System Response Time and User Satisfaction*[17] indicates that response times up to 6 seconds are tolerable. These numbers cannot be related directly to the tolerated delay, because the receiving peer does not exactly know when the sending peer wrote a message. Every additional proxy peers adds an amount of time to the

latency until the message arrives, as can be seen below.

Average Latency

The average latency for a peer is calculated as follows:

$$\frac{\sum_{i=0}^{(Proxypeerscount+1)} Proxypeerscount + 1}{Proxypeerscount + 1}$$

An additional count (+1) is needed to include the receiving peer itself. Depending on the sending interval this results in different average waiting times for a message to arrive, as expressed in the following formula:

$$Intervaltime * \frac{\sum_{i=0}^{(Proxypeers+1)} Proxypeers + 1}{Proxypeers + 1}$$

The average latency depending on the number of hosts and the sending interval used is shown in table 5.5.

Table 5.5: Average latency based on number of proxy peers

Proxy peers	Average latency (multiplied by delay times)					
	(# timeslots)	0.125s	0.25s	0.5s	1s	2s
1	0.5	0.0625s	0.125s	0.25s	0.5s	1s
2	1	0.125s	0.25s	0.5s	1s	2s
3	1.5	0.1875s	0.375s	0.75s	1.5s	3s
4	2	0.25s	0.5s	1s	2s	4s
5	2.5	0.3125s	0.625s	1.25s	2.5s	5s
6	3	0.375s	0.75s	1.5s	3s	6s
7	3.5	0.4375s	0.875s	1.75s	3.5s	7s
8	4	0.5s	1s	2s	4s	8s
9	4.5	0.5625s	1.125s	2.25s	4.5s	9s
10	5	0.625s	1.25s	2.5s	5s	10s

Maximum Latency

Furthermore besides the average latency, the maximum latency needs to be taken into account as well, which is shown in table 5.6.

Table 5.6: Maximum latency based on number of proxy peers

Proxy peers	Maximum latency (multiplied by delay times)					
	(# timeslots)	0.125s	0.25s	0.5s	1s	2s
1	1	0.125s	0.25s	0.5s	1s	2s
2	2	0.25s	0.5s	1s	2s	4s
3	3	0.375s	0.75s	1.5s	3s	6s

4	4	0.5s	1s	2s	4s	8s
5	5	0.625s	1.25s	2.5s	5s	10s
6	6	0.75s	1.5s	3s	6s	12s
7	7	0.875s	1.75s	3.5s	7s	14s
8	8	1s	2s	4s	8s	16s
9	9	1.125s	2.25s	4.5s	9s	18s
10	10	1.25s	2.5s	5s	10s	20s

5.9.2 Bandwidth Usage

Based on the calculated packet sizes (section 5.4, p. 45), the **minimal outgoing bandwidth capability**, as shown in table 5.7, is required. The highest needed bandwidth in case of 10 proxy peers sending at an interval of 0.125s results in 64 KiB/s (equivalent of 512 KBit/s). Today DSL connections usually exceed this limit by orders of magnitudes.

Table 5.7: Minimal Outgoing Bandwidth Capability

Proxy peers	Intervals / Bandwidth usage in KiB/s				
	0.125s	0.25s	0.5s	1s	2s
1	9.6	4.8	2.4	1.2	0.6
2	15.2	7.6	3.8	1.9	0.95
3	20.8	10.4	5.2	2.6	1.3
4	26.4	13.2	6.6	3.3	1.65
5	32	16	8	4	2
6	38.4	19.2	9.6	4.8	2.4
7	44.8	22.4	11.2	5.6	2.8
8	50.4	25.2	12.6	6.3	3.15
9	56.8	28.4	14.2	7.1	3.55
10	64	32	16	8	4

Even on mobile networks, HSPA Category 1 delivers 0.73 Mbit/s upstream bandwidth. Older ISDN and Modem connections would not suffice, though today there are a lot of alternative connections available.[50] If using only 9 proxy servers with 0.125s delay or 10 proxy servers with a delay of 0.25s, EDGE could be supported. The total required network bandwidth is the product of peers in the network multiplied by the outgoing bandwidth per peer:

$$RequiredNetworkBandwidth = NetworkPeers * PeerBandwidth$$

The relation between the bandwidth usage of a single peer and the required network bandwidth is visualised in figure 5.9. Thus in a network with 100 peers, all of them using 10 proxy peers and sending at a rate of 0.125s, the total required network bandwidth would be

$$RNB = 100 * 64KiB/s = 6400KiB/s = 6.25MiB/s = 50MBit/s$$


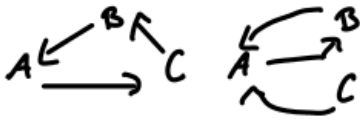
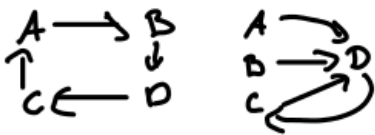
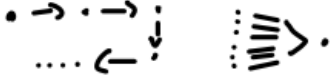
If also the receiving side is inside the same network, the required network bandwidth has to be multiplied by two and thus results in *100 MBit/s*. This is exactly half of what a fast Ethernet

Figure 5.9: Peer and Network Bandwidth

Bandwidth Usage

(number of postcards per interval)

Every peer sends
1 postcard per interval

		IN / PEER	OUT / PEER	NET TOTAL
2 peers :		1 (1)	1	2
3 peers :		1 (2)	1	3
4 peers		1 (3)	1	4
N peers		1 (N)	1	n

Average,
if distributed
equally

Maximum

switch can deliver, because fast Ethernet provides full duplex 100Mbit/s streams and thus a single fast Ethernet fabric would support 200 simultaneous users. As the network bandwidth cannot be controlled or influenced by the user directly, these calculations may simply indicate the network bandwidth usage for network providers. The user can focus on the bandwidth requirements found in table 5.7.

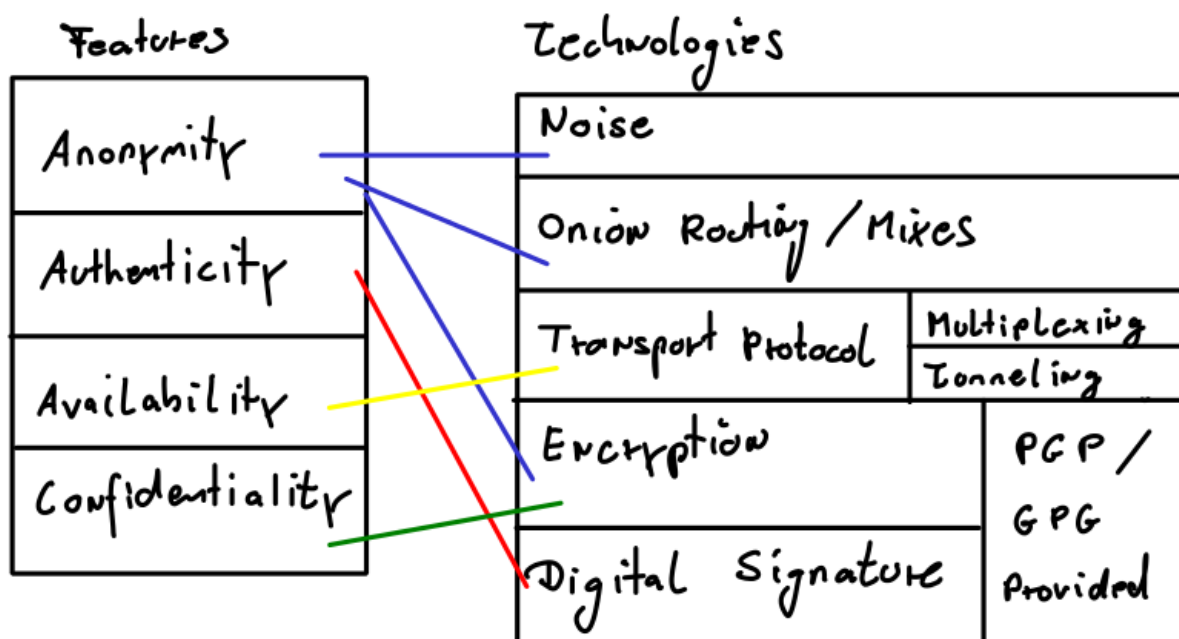
5.9.3 Fixed interval for sending

To prevent de-anonymisation by doing statistical analysis on the traffic, postcards are sent at a fixed rate. In case there is no message to be send, onions with no receiving peer (*noise*) are used instead.

To avoid problems with different sending intervals (queuing, buffer exceeding) a network wide fixed sending interval should be chosen. From the calculations made above, an interval of **0.25s** seems to provide a good trade-off between latency, anonymity and bandwidth usage.

The number of proxy peers can be chosen by every peer individually and can be used to focus on smaller bandwidth usage and less latency or a higher degree of anonymity. A value of 5 in a network of 100 peers would make de-anonymising less probable than winning the game Lotto (6 out of 49) in Germany (see table 5.8).

Figure 5.10: Features and Technologies



5.10 Features

This section describes which features are supported by the chat protocol. Figure 5.10 gives a quick overview of how the features relate to the used technologies.

5.10.1 Anonymity

One of the main objectives of this protocol is to provide a chat system that hides who is talking to whom (*Sender-Receiver Anonymity*). In practice there are limits on the degree of anonymity that can be reached. This protocol specifies the use of

- Onion Routing (5.8, p. 44),
- Noise (5.9, p. 47)
- and OpenPGP (5.6, p. 40)

to achieve a high degree of anonymity.

Degree of Anonymity

If an attacker controls all hosts that are part of the chat network, it is impossible to guarantee anonymity at all. Thus, to allow any degree of anonymity there must be hosts in the network, which are not run by the attacker besides the sender and the receiver. As specified by Onion Routing, there are a number of proxy peers used to hide the message receiver.

In a network in which the attacker does not run all nodes, there is a probability that the given route considers only the hosts run by the attacker. As soon as the route considers at least one different host, the attacker cannot distinguish between the real recipient and a proxy peer. So we have to consider the probability that the attacker controls **all** proxy peers for a given route only. This probability is the same as the one used to calculate the winning probability in the game of luck *Lotto*, in which the winning chances are expressed like this:

$$P_r = \frac{\binom{6}{r} \binom{N-6}{6-r}}{\binom{N}{6}}, r \in \{0, \dots, 6\}$$

Depending on the number of hosts in a network (number of possible numbers in lotto) and on the number of hosts in a specific route (number of correct numbers in lotto), the de-anonymisation probabilities (winning probabilities in lotto) are shown in table 5.8.

Table 5.8: De-Anonymisation Probabilities (for one packet)

Peers in network / Proxy peers	10	10 ²	10 ³	10 ⁴	10 ⁵
1	1:10	1:100	1:1000	1:10000	1:100000
2	1:45	1:4950	1:499500	1:4.9995e+07	1:4.99995e+09
3	1:120	1:161700	1:1.66167e+08	1:1.66617e+11	1:1.66662e+14
4	1:210	1:3.92122e+06	1:4.14171e+10	1:4.16417e+14	1:4.16642e+18
5	1:252	1:7.52875e+07	1:8.25029e+12	1:8.325e+17	1:8.3325e+22
6	1:210	1:1.19205e+09	1:1.36817e+15	1:1.38681e+21	1:1.38868e+27
7	1:120	1:1.60076e+10	1:1.94281e+17	1:1.97996e+24	1:1.98371e+31
8	1:45	1:1.86088e+11	1:2.41151e+19	1:2.47322e+27	1:2.47946e+35
9	1:10	1:1.90223e+12	1:2.65802e+21	1:2.74583e+30	1:2.75474e+39
10	1:1	1:1.73103e+13	1:2.6341e+23	1:2.74336e+33	1:2.75449e+43

As can be seen in this table, the probability of an attacker being able to de-anonymise in a network of 100 peers using 5 proxy peers is less than the probability of winning Lotto with 6 of 49 (1:13983816). To support a high chance of not hitting the attackers peers and thus avoiding de-anonymisation, the number of peers in the network as well as the number of proxy peers should be increased as much as possible. The number of peers in the network are depending on how many peers are actually using the network. This can be increased to a certain degree by adding artificial peers. The number of proxy peers cannot be increased indefinitely, as shown in section 5.9.1, p. 47.

5.10.2 Authenticity and Confidentiality

Digital signatures provide methods to ensure that a given message was composed by a given public key and that the content has not been modified. The process of encryption protects data from external viewing and thus ensures the confidentiality of a message. The features of OpenPGP (5.6, p. 40) are being used to guarantee authenticity.

5.10.3 Availability

To prevent easy denial of service attacks, this protocol specifies various measurements to make denial of service attacks harder. In particular the following techniques are used to support the availability:

- Transport protocol multiplexing (5.7.3, p. 41)
- Transport protocol tunnelling (5.7.2, p. 41)

5.10.4 Direct chat

This version of the chat protocol supports only direct chat (1:1), as opposed to multi user chat.

5.11 Summary

Summarised these requirements and features break down to:

1. Nobody, but the intended receiver(s) know(s) *that* you wrote a message.
2. Nobody, but the intended receiver(s) can view the *message content*.
3. Nobody, but the intended receiver(s) can *verify* the source of the message being you.
4. Nobody, but the intended receiver(s) know(s) *who* the message was sent to.
5. It should be hard (if not practically impossible) to block chatting.

Chapter 6

Implementation of the Prototype

The prototype was developed using the spiral software development model. It is named *ceof*. The source code can be found in the directory *src*.

6.1 Environment

The prototype was developed using the Python3 programming language. In addition to the standard libraries, the python-gnupg module is required. To create a python environment that is suitable for running and developing the prototype, a new virtualenv including the required python-gnupg module can be created using the following commands:

```
% virtualenv -p /usr/bin/python3 python-env
% . ./python-env/bin/activate
% pip install python-gnupg
% (cd python-env/bin && ln -s python python3)
```

Because python is an interpreted scripting language with interpreters available for all major platforms, the prototype should be runnable on all major platforms. All configurations are saved in the *cconfig*[43] format. The path to the configuration directory is usually derived by taking the content of the environment variable *HOME* and appending the subdirectory *.ceof*. On Unix this is referenced as *~/.ceof/*.

6.2 Usage

6.2.1 Command Line Interface (CLI)

The implemented prototype provides a command line interface to access all functionality. All operations are grouped into commands, which are handled by the executable *bin/ceof*. If a subcommand is followed by the parameter *-h*, then a usage screen is displayed.

6.2.2 Cryptographic Operations

All cryptographic operations are accessible by using the *crypto* command, as shown in figure 6.1. This module uses the python-gnupg module for cryptographic functionality. At first it is required

Figure 6.1: Crypto Command Usage

```
(python-env)[18:40] brief:src% ./bin/ceof crypto -h
usage: ceof crypto [-h] [-d] [-v] [-c CONFIG_DIR]
                  [--encrypt ENCRYPT [ENCRYPT ...]] [--decrypt] [-e] [-f]
                  [-g] [-i] [-l LENGTH] [--name NAME]
                  [--email-address EMAIL_ADDRESS] [-s]
```

optional arguments:

-h, --help	show this help message and exit
-d, --debug	Set log level to debug
-v, --verbose	Set log level to info, be more verbose
-c CONFIG_DIR, --config-dir CONFIG_DIR	Select configuration directory (\$HOME/.ceof by default)
--encrypt ENCRYPT [ENCRYPT ...]	Encrypt from stdin (specify recipients)
--decrypt	Decrypt from stdin
-e, --export	Export public key to stdout
-f, --fingerprint	Show key fingerprint
-g, --gen-key	Generate new private/public key pair
-i, --import	Import public key from stdin
-l LENGTH, --length LENGTH	Specify bit length for key generation
--name NAME	Name (for key generate)
--email-address EMAIL_ADDRESS	E-Mail-Address (for key generate)
-s, --show	Show private/public key pair

Get ceof at <http://www.nico.schottelius.org/software/ceof/>

to create a new public/private key pair using the *--gen-key* parameter (figure 6.2), which can be displayed using the *--show* parameter (figure 6.3) and exported using the *--export* parameter (figure 6.4). Importing keys is possible via standard input (*stdin*) and shown in figure 6.5. Furthermore decryption and encryption are supported.

Figure 6.2: Generation of Public/Private Key Pair

```
(python-env)[18:52] brief:src% ./bin/ceof crypto --gen-key
--name "Nico Schottelius" --email-address "nico@example.org"
```

Figure 6.3: Show Public/Private Key Pair

```
(python-env)[18:54] brief:src% ./bin/ceof crypto --show
{'dummy': '', 'keyid': 'C5FC26760DD842D6', 'expires': '', 'length': '2048',
'ownertrust': '', 'algo': '1',
'fingerprint': '77E54EF64A6395FF2769B2F4C5FC26760DD842D6',
'date': '1339174371', 'trust': '', 'type': 'sec',
'uids': ['Nico Schottelius (EOF42KEY) <nico@example.org>']}
```

Figure 6.4: Export Public Key

```
(python-env)[19:23] brief:src% ./bin/ceof crypto --export
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.19 (GNU/Linux)

mQENBE/SLeMBCACu5sWt3j/ZTqZZ5eZw+cTvkIG6DwWaeVZjv+A+Dd7xZhbMBeyZ
q70Cu0EURGLQUQQKtyT7bvTBjk81kL2zcgIJ2a/MQQneJc+fEqB+ovlPM+B14qLf
TIuBMPnI+10MOuTx0Agtys+6/9YaIdKaedtIqrZhVVsbaFAeE6MTHSm0i9bTtvyk
bH+X0JurCNL8nKEjf6SdSrQGdmohV/VyQTG1MZPaYG58LjCMKxbqWMb31VKsmyRr
N4bZFPePzqBJzmqyH/noyoNuzbSUhNvUw27JzTL51u1JfMm2kQmkG1NZgLwXg6/W
e5FYbVoVI3LMj9NDABZ42y3mCv0QJJp6LkvdABEBAAGOLk5pY28gU2Nob3R0ZWxp
dXMgKEVPRjQySOVZKSA8bmljb0BleGFtcGxlLm9yZz6JATgEEwECACIFAk/SLeMC
Gy8GCwkIBwMCBhUIAgkKCwQWAgMBAh4BAheAAAoJEMX8JnYN2ELW8rcH/3Hdanzp
mUNfF7Rq1U7sCwrGKFAB0vTZtWQBfURLfTW2kMZREViRu6t3aB0hK3g1HwASBzb
yXJmH6UznPkOG5gD+Y/FfMCiR7VZaiFXEZh9ukRWDwCytouoILDrey08Wr4YQEDf
+Ny38gLYu06Svnm25iQn3LiejTohCny5P0k0nxfyVx0EhQ6LUjai6j0bSKk05o62
b2ZdKpGHsBqo9eHLr4y83Jmo05pSXDBYBG0pu2Ukczey8BGUbwngUwEN/XKrl1xZ
aYHlpVNhCNSXthSAJdlag5Auju/t2S978yel04Ii411dyDPrYZKjd4TGWbWfeVpS
jXOLmX++gs2tPyM=
=LB0W
-----END PGP PUBLIC KEY BLOCK-----
```

Figure 6.5: Import Public Key

```
# Export key from test peer0 and import into normal
# configuration directory
./bin/ceof crypto --config-dir ../test/peers/0 --export |
./bin/ceof crypto --import
```

Figure 6.6: Listener Command Usage

```
(python-env)[18:41] brief:src% ./bin/ceof listener -h
usage: ceof listener [-h] [-d] [-v] [-c CONFIG_DIR] [-a ADD] [-l] [-r REMOVE]
```

optional arguments:

```
-h, --help            show this help message and exit
-d, --debug           Set log level to debug
-v, --verbose         Set log level to info, be more verbose
-c CONFIG_DIR, --config-dir CONFIG_DIR
                      Select configuration directory ($HOME/.ceof by
                      default)
-a ADD, --add ADD     Add an address to listen on
-l, --list            List listener
-r REMOVE, --remove REMOVE
                      Remove an address to listen on
```

Get ceof at <http://www.nico.schottelius.org/software/ceof/>

6.2.3 Listener

The *listener* command is used to configure to which addresses the prototype is listening to (figure 6.6). For the initial setup it is required to configure at least one listener using the *--add* parameter, which can be shown afterwards using *--list* parameter (figure 6.7). Protection against adding unsupported addresses is included.

6.2.4 Noise

The noise command does not accept any parameters and will output noise to standard output (*stdout*). To be able to generate noise, the prototype requires UTF-8 encoded files to be present in the noise directory (*~/ceof/noise*). Possible sources of noise are the archive of RFCs or the Linux Kernel sources, which are copied into the noise directory as shown in figure 6.8. The noise command is only provided for debugging purposes.

Figure 6.7: Add and list listener addresses

```
(python-env)[19:23] brief:src% ./bin/ceof listener --list
(python-env)[19:42] brief:src% ./bin/ceof listener --add tcp://0.0.0.0:42507
(python-env)[19:42] brief:src% ./bin/ceof listener --add tcp://0.0.0.0:42508
(python-env)[19:42] brief:src% ./bin/ceof listener --list
tcp://0.0.0.0:42507
tcp://0.0.0.0:42508
(python-env)[19:42] brief:src% ./bin/ceof listener --add foo://0.0.0.0:42508
Unknown protocol in address foo://0.0.0.0:42508
```

Figure 6.8: Init Noise Directory

```
% mkdir -p ~/.ceof/noise
% cd ~/.ceof/noise
% find ~/linux/linus -name '*.c' -type f -exec cp {} . \;
% find ~/rfc/mirror/rfcs-text-only/ -name '*.txt' -type f -exec cp {} . \;
% ls | wc -l
23287
```

6.2.5 Peer

The `peer` command is used to add, remove and list peers, as well as adding and removing addresses to peers. Its usage is shown in 6.9. When adding a peer, it is required to import its public key using the `crypto` command before (see figure 6.5). After adding a peer, addresses can be added or removed (see figure 6.10).

6.2.6 Onion

The `onion` command is used to create and send onions to other peers. The help page is shown in figure 6.11. To create and display the onion, which would usually be sent on the network, use the `--message` parameter (figure 6.12). To actually send it, use the `--send` parameter.

6.2.7 Server

The `server` command is used to start the server and does usually not take any arguments, though specific servers parts can be disabled (see figure 6.13). Disabling the listener makes it impossible to receive messages, disabling noise prevents the server from sending messages regularly and disabling the UI Server disables listening for chat UIs (see chapter 6.6).

Figure 6.9: Peer Command Usage

```
(python-env)[20:18] brief:src% ./bin/ceof peer -h
usage: ceof peer [-h] [-d] [-v] [-c CONFIG_DIR] [-a] [-r] [-l]
                [-f FINGERPRINT] [--add-address ADD_ADDRESS]
                [--remove-address REMOVE_ADDRESS]
                [name]

positional arguments:
  name                  Name of the peer (myself: you)

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           Set log level to debug
  -v, --verbose         Set log level to info, be more verbose
  -c CONFIG_DIR, --config-dir CONFIG_DIR
                        Select configuration directory ($HOME/.ceof by
                        default)
  -a, --add             Add a peer
  -r, --remove          Remove a peer
  -l, --list            List peers
  -f FINGERPRINT, --fingerprint FINGERPRINT
                        Specify fingerprint for peer
  --add-address ADD_ADDRESS
                        Add an address to a peer
  --remove-address REMOVE_ADDRESS
                        Remove an address from a peer

Get ceof at http://www.nico.schottelius.org/software/ceof/
```

Figure 6.10: Peer Add

```
(python-env)[21:45] brief:src% ./bin/ceof peer --add peer0
--fingerprint 729BD24186E4E3F7EA3872FCAB29961528ACE126
(python-env)[21:46] brief:src% ./bin/ceof peer
--add-address tcp://127.0.0.1:42222 peer0
(python-env)[21:46] brief:src% ./bin/ceof peer --list
peer0/729BD24186E4E3F7EA3872FCAB29961528ACE126/['tcp://127.0.0.1:42222']
(python-env)[21:47] brief:src% ./bin/ceof peer
--add-address tcp://127.0.0.1:42223 peer0
(python-env)[21:47] brief:src% ./bin/ceof peer --list
peer0/729BD24186E4E3F7EA3872FCAB29961528ACE126/['tcp://127.0.0.1:42222',
'tcp://127.0.0.1:42223']
```

Figure 6.11: Onion Command Usage

```
(python-env)[20:16] brief:src% ./bin/ceof onion -h
usage: ceof onion [-h] [-d] [-v] [-c CONFIG_DIR] [-m MESSAGE]
                [-r REPEAT_COUNT] [-s]
                [name]
```

positional arguments:

name Name of the peer

optional arguments:

```
-h, --help                show this help message and exit
-d, --debug               Set log level to debug
-v, --verbose             Set log level to info, be more verbose
-c CONFIG_DIR, --config-dir CONFIG_DIR
                           Select configuration directory ($HOME/.ceof by
                           default)
-m MESSAGE, --message MESSAGE
                           Create onion with this message for peer
-r REPEAT_COUNT, --repeat-count REPEAT_COUNT
                           Repeat action n times (used for timing/profiling)
-s, --send                Send message created to peer
```

Get ceof at <http://www.nico.schottelius.org/software/ceof/>

Figure 6.12: Create and send an Onion

```
(python-env)[21:57] brief:src% ./bin/ceof onion -m 'Hello dear peer0!' peer0
Onion chain: -----BEGIN PGP MESSAGE-----
Version: GnuPG v2.0.19 (GNU/Linux)

hQEMA0TOaZwAXdx8AQf/Uu8OGiVQRSTSW1ExuzFJpebZBHsHx7MdchDdF9Q1fW1V
[...]
```

Figure 6.13: Server Command Usage

```
(python-env)[22:09] brief:src% ./bin/ceof server -h
usage: ceof server [-h] [-d] [-v] [-c CONFIG_DIR] [-l] [-n] [-u]
                  [-a UI_ADDRESS] [-p UI_PORT]

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           Set log level to debug
  -v, --verbose         Set log level to info, be more verbose
  -c CONFIG_DIR, --config-dir CONFIG_DIR
                        Select configuration directory ($HOME/.ceof by
                        default)
  -l, --no-listener     Disable listener server
  -n, --no-noise        Disable noise sending
  -u, --no-ui           Disable UI server
  -a UI_ADDRESS, --ui-address UI_ADDRESS
                        Listen on this address for UI connections
  -p UI_PORT, --ui-port UI_PORT
                        Listen on this port for UI connections

Get ceof at http://www.nico.schottelius.org/software/ceof/
(python-env)[22:09] brief:src%
```

6.2.8 Transport Protocols (TP)

The *tp* command is used to manage transport protocols, its usage is shown in figure 6.14. The parameter *--list* shows the available protocols, the parameter *--route* generates a random route to a peer. Both commands are show in figure 6.15.

Figure 6.14: TP Command Usage

```
(python-env)[22:04] brief:src% ./bin/ceof tp -h
usage: ceof tp [-h] [-d] [-v] [-c CONFIG_DIR] [--chain-to] [-l] [-r] [name]

positional arguments:
  name                  Name of the peer

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           Set log level to debug
  -v, --verbose         Set log level to info, be more verbose
  -c CONFIG_DIR, --config-dir CONFIG_DIR
                        Select configuration directory ($HOME/.ceof by
                        default)
  --chain-to            Generate onion package for given peer
  -l, --list            List available transport protocols
  -r, --route-to        Generate route to given peer
```

Get ceof at <http://www.nico.schottelius.org/software/ceof/>

6.3 Code Examples

6.3.1 Modular Design

The implementation has been split into several python modules, which are stored below *src/lib/ceof* (figure 6.16). When running in server mode, the modular design helps to separate parts into own processes in such a way that every process could run on a separate computing core. The big picture is shown in figure 6.17. As can be seen further in this figure, the processes communicate with each other using queues. The optional chat user interface is also integrated using a separate *UIServer* process.

6.3.2 Sequence Numbers

Sequence numbers for packets are encoded in a custom base 64 encoding to define the used characters. Sequence numbers are stored in the ID field (section 5.3.3) and can range from 0 to

Figure 6.15: TP List and Route

```
(python-env)[22:05] brief:src% ./bin/ceof tp -l
tcp
(python-env)[22:07] brief:src% ./bin/ceof tp --route peer0
[<peer9/9205979C4E7D5F5896520E2941324091BA8B832A>,
<peer1/999C49CF1E9BBDAB0DE169324E1521D6E125E86C>,
<peer0/729BD24186E4E3F7EA3872FCAB29961528ACE126>,
<peer3/2FBD910F1C4C6032B399513B57647432847125F4>,
<peer6/4A85FEED9DDAAE37EE68F5DC1E4208D4E5FD84EB>,
<peer4/07D0FD80AF46E51DAEE4363144CE699C005DDC7C>]
(python-env)[22:07] brief:src%
```

68719476735 $((64^6) - 1)$.

The transformation from an integer to sequence number (also called *eofid*) is made based on transformation of a string and indexes. The *get_next()* method returns the next sequence number and takes care of range overflows. Code parts are show in figure 6.18.

6.3.3 Queuing

Several queues have been implemented for using of *inter process communication (IPC)* as shown in figure 6.17. The main server process creates queues on startup and polls them regularly. As it is not clear whether using the *select()* method for polling is interoperable, manual polling with a sleep timeout has been implemented. The queue and polling process has been programmed in such a way that the main server can use one loop to poll data on all queues and select the right handler based on a dictionary entry, which has the same name as the queue dictionary entry (figure 6.19).

6.3.4 Onion Creation

To support onion routing, the sender of a message needs to encrypt the packet multiple times, once for each host that receives the packet. The process to do so may look like this:

1. Create message (from noise or user input)
2. Create source path
3. Create packet for last peer
4. Create packet for last-1 peer including previous packet
5. Continue until first peer is reached
6. Sent packet to first peer

The actual implementation can be found in *src/lib/ceof/onion.py*. An excerpt on how the onion is created can be seen in figure 6.20.

Figure 6.16: Modular Design

```
[13:42] brief:.bachelorarbeit% find src/lib/ceof -name \*.py
src/lib/ceof/__init__.py
src/lib/ceof/config/__init__.py
src/lib/ceof/config/listener.py
src/lib/ceof/config/peer.py
src/lib/ceof/crypto.py
src/lib/ceof/eofid.py
src/lib/ceof/eofmsg.py
src/lib/ceof/noise.py
src/lib/ceof/onion.py
src/lib/ceof/server/__init__.py
src/lib/ceof/server/listener.py
src/lib/ceof/server/sender.py
src/lib/ceof/server/tcp.py
src/lib/ceof/server/ui.py
src/lib/ceof/test/__init__.py
src/lib/ceof/test/__main__.py
src/lib/ceof/test/id/__init__.py
src/lib/ceof/test/peers/__init__.py
src/lib/ceof/test/ui/__init__.py
src/lib/ceof/test/uiserver/__init__.py
src/lib/ceof/tp/__init__.py
src/lib/ceof/tp/tcp/__init__.py
src/lib/ceof/ui/__init__.py
src/lib/ceof/ui/main.py
src/lib/ceof/ui/net.py
```

Figure 6.17: Implementation: Big Picture

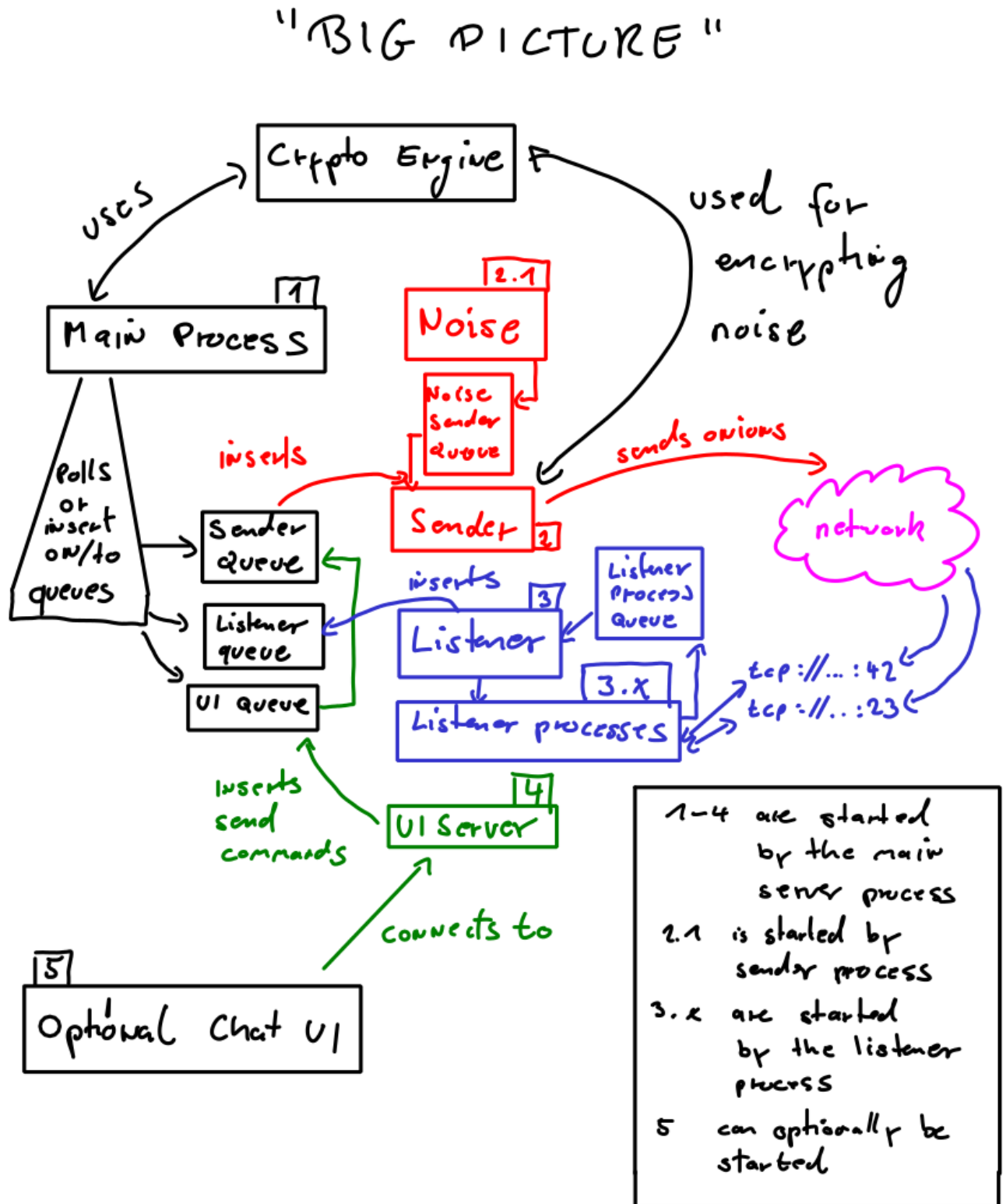


Figure 6.18: EOFID Code Example

```
...
EOF_ID_CHARS = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-!"
...

def int_to_id(to_convert):
    """Convert int to ID"""
    index = ceof.EOF_L_ID-1
    eofid = []

    while index >= 0:
        part = ceof.EOF_ID_BASE**index

        # Fits in? Record and subtract
        if (to_convert - part) >= 0:
            times = int(to_convert / part)
            to_convert = to_convert - (times*part)
        else:
            times = 0

        # Append selected symbol
        eofid.append(ceof.EOF_ID_CHARS[times])
        index = index - 1

    return "".join(eofid)
```

Figure 6.19: Queues / Polling Code Example

```
while True:
    for name, q in self.queue.items():
        log.debug("Polling on %s queue" % name)
        data = False
        try:
            data = q.get(block=False)
        except queue.Empty:
            pass

        if data:
            log.debug("Got message from: %s:%s" % (name, data))
            self.handler[name](data)

    time.sleep(ceof.EOF_TIME_QPOLL)
```

6.4 Test of the Prototype

Several tests were made to verify the correct behaviour of the prototype. For most tests the network sniffer Wireshark was used to verify what can be seen on the wire.

6.4.1 Onion Structure

This test verifies that an onion is constructed in the correct order and that the correct fields being setup and noisified for every peer. This test was done by running the implementation with the following parameters:

`./bin/ceof onion --plain --debug --message Test peer0`. After analysing the output, it was verified that the onion is created in the correct order and fields contain noise.

Test Result

- Success

6.4.2 Noise Sending

This objective of this test is to verify that the implementation

- (a) sends out packets regularly
- (b) selects different destination peers
- (c) selects different addresses of each peer

Figure 6.20: Onion Example Code

```
...
# Initialise parameters
    peer = ceof.Peer.from_disk(config.peer_dir, args.name)
    route = ceof.TransportProtocol.route_to(config.peer_dir,
        peer, ceof.EOF_L_ADDITIONAL_PEERS)
    chain = ceof.TransportProtocol.chain_to(route, peer, args.message)
    # Copy for debug
    orig_chain = list(chain)

    onion = cls(config.gpg_config_dir)
    onion_chain = onion.chain(chain)
...
def chain(self, chain):
    """Create an onion chain"""

    # Get our packet to work on
    pkg = chain.pop()
    log.debug("Onion: Encrypting for %s, chain = %s" % (str(pkg), str(chain)))

    # If there is more, call us again
    if chain:
        inner_part = self.chain(chain)
    else:
        inner_part = ""

    eofmsg = pkg['eofmsg']
    fingerprint = pkg['peer'].fingerprint

    onion = self.crypto.encrypt(str(eofmsg) + str(inner_part), fingerprint)

    return str(onion)
```

Test Setup

The implementation was configured to know about 11 peers, with each one having at least two addresses in the format of `tcp://127.0.0.1:4294`, where the port was a random port in the range of 4000-50000. The implementation was started using `./bin/ceof server --verbose`. The packets on the network were captured using the network sniffer. To be able to receive the packets, the `socat` command was started to listen on all ports on which the peers would listen. The `socat` command line is shown in figure 6.21.

Figure 6.21: Socat command line for testing

```
for port in $(cat ~/.ceof/peers/*/addresses | sed 's/.*://'); do
    ( socat TCP4-LISTEN:$port,reuseaddr,fork - & );
done
```

Test Result

- Success

Using the timeline of the network sniffer and a filter to match only new packets (`tcp.flags.syn == 1` and `tcp.flags.ack == 0`), it was revealed that the sending interval equals to the one defined in the chat protocol. To verify that all related component of the implementation depend on the specific internal sending interval, the sending interval was changed from **0.250s** to **0.125s**. The difference on the network traffic can be seen in figures 6.22 and 6.23. The selected destination addresses as seen in the network sniffer were compared with the addresses of the peers to verify that the implementation does indeed select different peers and different addresses of each peer.

6.4.3 Encryption

This test is used to verify that all network traffic is encrypted.

Test Setup

The setup was prepared identical to the one described in section 6.4.2. All packets seen within an interval of 60 seconds, in the network sniffer and on the receiving side, were analysed.

Test Result

- Success

All encrypted messages have been filtered out of the captured packet list. Zero packets have been left, no unencrypted traffic was produced by the implementation.

Figure 6.22: Sending data at 0.250s interval

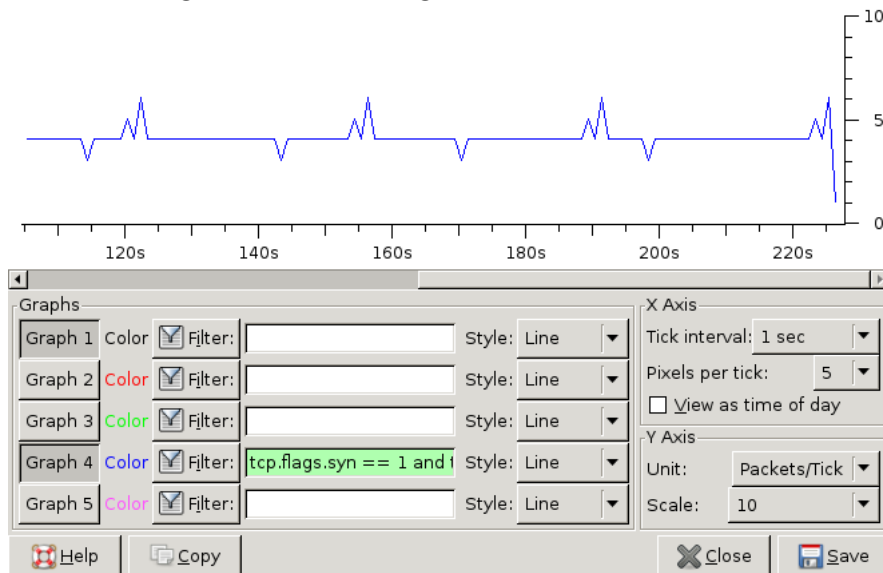


Figure 6.23: Sending data at 0.125s interval



6.4.4 Receiving Peers Can Decrypt Message

When sending out a message to another peer, the message should be decryptable and readable by the receiving peer, independent of the message type. This test verifies that messages can be decrypted.

Test Setup

In contrast to the tests described in sections 6.4.2 and 6.4.3, only one postcard was produced. The number of known peers was reduced to 6, so that every participant needs to receive a postcard. Afterwards a test instance for every peer was started using the script `start_peers.sh` (stored at `test/start_peers.sh`). Afterwards a message was sent manually using `./bin/ceof onion -m "hallo peer0" -s peer0`.

Test Result

- Success

As can be seen in the log files (`doc/logs/message-receive-log-of-peers-debug-log` and `doc/logs/message-send-debug-log`) the message was received by all peers and successfully decrypted, displayed (in case of `peer0`) and forwarded to the last peer.

6.4.5 Receiving Peer: Authenticity Verification

When sending out a real message to another peer, the message should be encrypted and signed. The current prototype does **not** contain the logic to sign messages or to verify signatures.

Test Result

- Fail

6.4.6 Effective Bandwidth Usage

The calculations in section 5.9.2 are based on experimental packet sizes as seen in section 5.4, which do not include protocol overhead. This test should reveal the effective required bandwidth.

Test Setup

To test the effective bandwidth usage, three steps are taken:

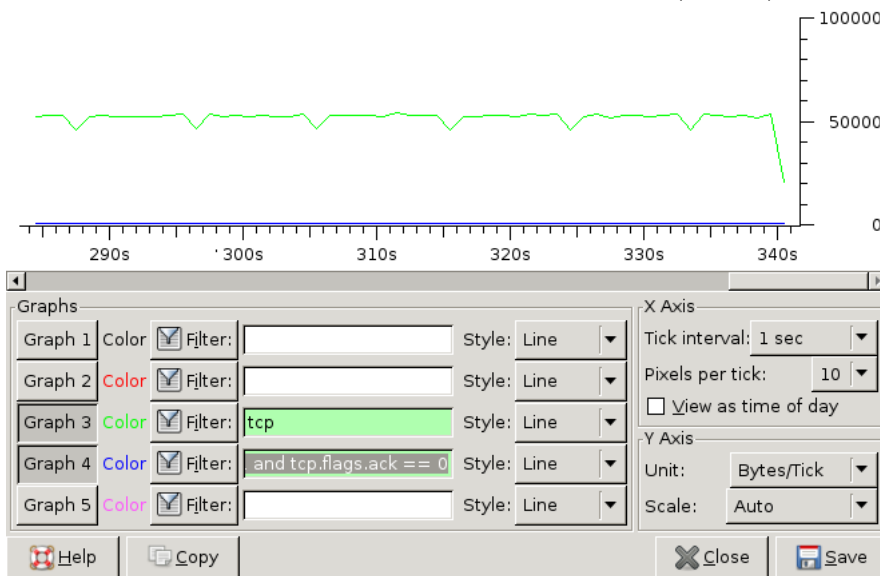
- Measure idle bandwidth (a)
- Measure bandwidth while running the prototype (b)
- Calculate effective bandwidth as a difference of b and a

To simplify the test, it was ensured that the idle bandwidth on the loopback interface was 0 Bit per second. The setup was prepared identically to the one described in section 6.4.2, additionally changing the sending interval to the values **0.125s**, **0.250s** and **0.500s**.

Test Result

When sending packets at the defined interval of **0.250s**, the effective bandwidth measured is approximately 25000 Bytes (24 KiB) per second. When changing the interval to **0.500s**, the effective bandwidth measured is 12000 Bytes (12 KiB) per second, when changed to **0.125s**, it is 50000 Bytes (49KiB) per second. Figures 6.24, 6.25 and 6.26 show the test results from the network sniffer. Comparing these numbers to the calculated bandwidth usage (section 5.9.2, p. 49), the effective bandwidth usage is 1.5 times higher than calculated ($50000/32768 = 1.52587890625$, $25000/16384 = 1.52587890625$, $12000 / 8000 = 1.5$). Thus, the measured bandwidth of 24 KiB/s or 192 KBit/s is the required outgoing bandwidth to run the chat system.

Figure 6.24: Effective Bandwidth Usage (0.125s)



6.4.7 Performance

It is important that the system generating noise is able to generate and send noise at the given interval of **0.25s**.

Test Setup

This test was inverted and instead of trying to send out data at an interval of **0.25s**, the artificial sending limit was removed and measured how many packets per second could be sent. The setup was prepared identically to the one described in section 6.4.2.

Test Result

- Success

When sending without a limit, a maximum of 13 packets / second was measured (see figure 6.27). The CPU usage of one core was around 38% during the test. As the encryption was suspected to

Figure 6.25: Effective Bandwidth Usage (0.250s)

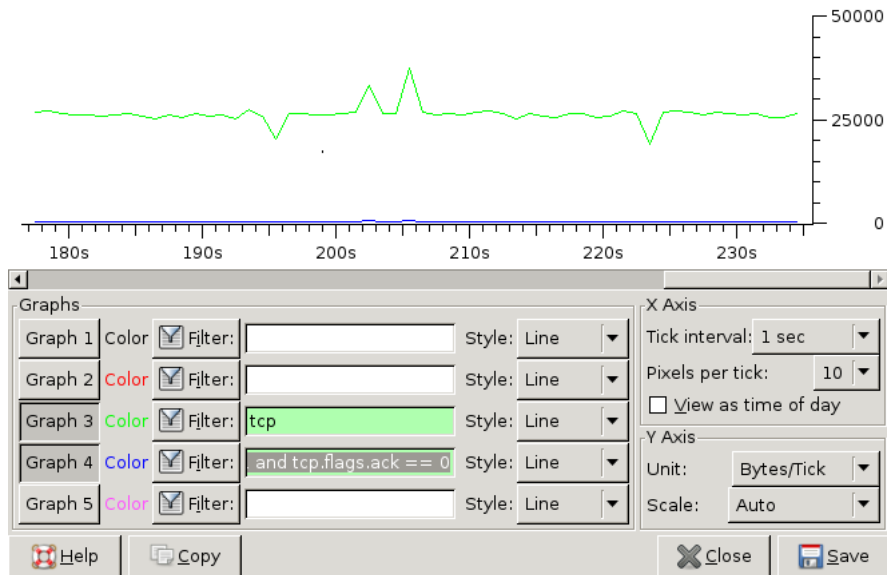


Figure 6.26: Effective Bandwidth Usage (0.500s)

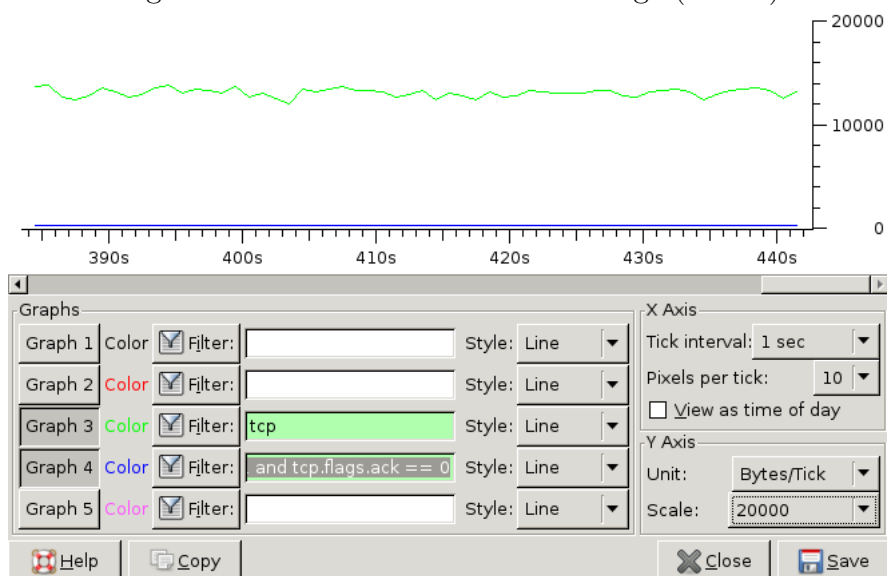
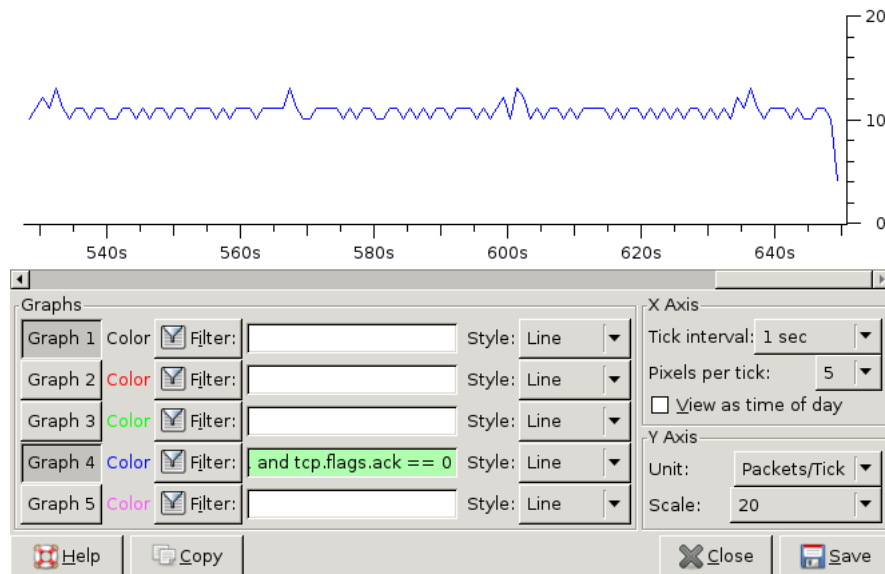


Figure 6.27: Packets per second (without sending limit)



limit the number of outgoing packets, another test with plain text packets (i.e. no encryption) was made. When changing the implementation to send out plain text packets instead of onions the CPU usage raised to 100% and around 250 packets / second were sent (see figure 6.28).

6.5 Features

Based on the requirements defined in section 5.10, p. 52, the following features have been implemented.

6.5.1 Anonymity

The existing prototype implements anonymity including all features as described in section 5.10.1. Currently the number of proxy peers is hardcoded to 5. This implementation limit can easily be changed and made a command line parameter.

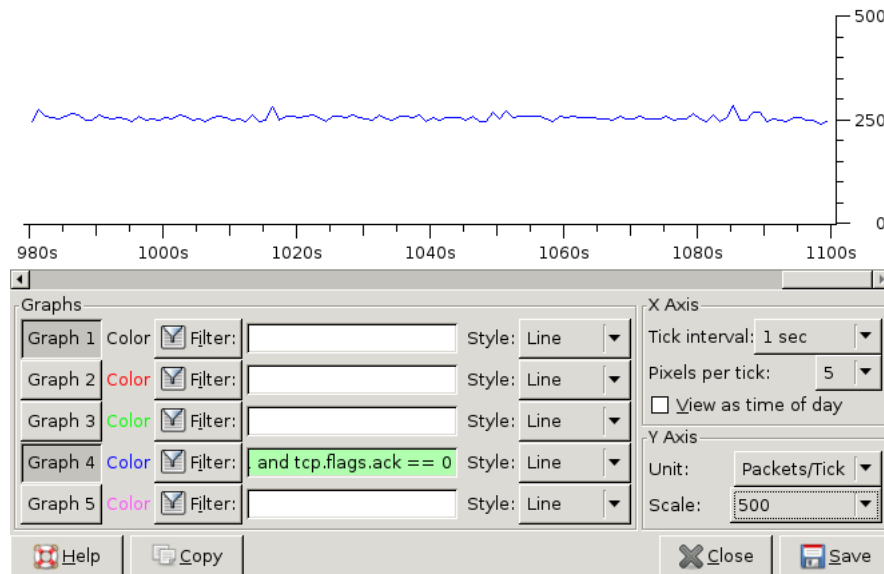
6.5.2 Authenticity and Confidentiality

The prototype does not sign messages and thus the authenticity is not verifiable. It does encrypt the messages multiple times, though. Adding authenticity requires two modifications to the source code:

- Sign real messages (EOFmsg 3002 / 3003)
- Check signature on real messages

Integrity checking can be done with any signed messages, sender verification is only possible for messages which have been signed with a trusted public key. As this includes the implementation of trust levels, the prototype does not support authenticity verification.

Figure 6.28: Packets per second (without sending limit, plain text)



6.5.3 Availability

Availability should be provided by

- Transport protocol multiplexing (5.7.3, p. 41)
- Transport protocol tunnelling (5.7.2, p. 41)

The current prototype supports only one transport protocol, but allows for listening on a number of tcp ports. Thus the onions are encoded and tunnelled into one transport protocol only. The source code allows to seamlessly integrate a new transport protocol, by creating a new folder below *src/lib/ceof/tp/* with the name of the transport protocol and adding a python module into it. The existing implementation of tcp can be used as an example.

6.6 Related Project: Chat User Interface

In addition to the described prototype, in a different project a chat user interface has been developed that can talk to the prototype using a chat server. This chat UI is documented in the file *doc/appendix/chat-ui.pdf*, which is included in the digital distribution.

Chapter 7

Conclusions

The given objectives as defined in section 1.3 have been reached. In the analysis of chat systems the bad feeling regarding Skype proved true. Information about the internals of the implementation and the protocol disappeared from the Internet during the development stage of this thesis and government agencies claim to be able to wire tap communication via Skype. Therefore the need for a secure anonymous chat system has been reconfirmed. The analysis of the related communications protocols showed that research for anonymity systems has been a long ongoing topic pursued by both academics and technology enthusiasts (hackers). The definition of the chat protocol and the development of the prototype implementation were tightly coupled together by use of the spiral software development model.

Summarised it is a great feeling to see that the theoretical ideas described in the chat protocol as well as the practical implementation have been realised. As far as I can see, this thesis produced a working prototype of a secure, decentralised and anonymous chat system. It is able to hide who is talking to whom.

7.1 Review

The analysis of existing anonymity systems generally revealed the following aspects:

- Anonymity systems have been built for a long time
- Anonymity systems often taking a conservative approach regarding resource usage
- Anonymity systems are usually built as an abstract network, not for a specific application

My thesis deviates from other approaches with regard to the latter two topics. Instead of avoiding to constantly send out data, this technique is actually used to improve the degree of anonymity. It does so in two ways:

1. Avoid statistical packet analysis
2. More traffic means more effort needed for potential attackers

However, it has to be acknowledged that there are situations in which this approach may not be desirable. For instance in a situation when high volume traffic is expensive (e.g. on some mobile

device contracts). Furthermore, due to the constant traffic, it is easy to detect and probably impossible to hide that someone is participating in this chat network. With the use of transport protocol multiplexing and transport protocol tunnelling it is possible to enhance the availability, but in case an institution forbids the use of this chat protocol, it can reliably detect people using it and terminate the complete network link.

Developing the anonymity system specific for an application has the advantage of ease of use, but the disadvantage of knowledge and resource duplication for the implementation. On the other hand, this work chat could be used as a basis for expansion: taking the defined protocol and its prototype, it is possible to generalise the protocol to support more features and to enhance the prototype.

7.2 Future Work

Although the defined chat protocol is in a usable state and the proof-of-concept implementation works, there are further enhancements that can be considered for future work.

7.2.1 Prototype Enhancements

Authenticity Checks

The current prototype does not support checking authenticity: It only checks whether the message can be decrypted, but does not check whether the message has been signed and whether the signature is correct.

Verify Cross OS Support

The prototype has been tested on Linux only. As the implementation does not contain any known OS specific calls, it should also work on other operating systems, but this has to be verified.

7.2.2 Transport Protocol Support

The current prototype only supports TCP for sending and receiving data. To effectively support transport protocol multiplexing, the number of supported protocols should be increased. Table 7.1 shows a proposal of some transport protocols which could be supported.

Table 7.1: Upcoming transport protocols

Scheme	Name	Format	Description
dns	DNS	<code>dns://[host:port]/domain/type</code>	Domain Name Service
http	HTTP	<code>http://host:port/resource</code>	Hypertext Transfer Protocol
https	HTTPS	<code>https://host:port/resource</code>	HTTP encrypted with SSL
mailto	E-Mail	<code>mailto://address</code>	Send message via e-mail
mediawiki	Mediawiki	<code>mediawiki://host:port/page</code>	Communication via Mediawiki
smb	SMB	<code>smb://[user[:password]@]host/path</code>	Server Message Block
smtp	SMTP	<code>smtp://[user[:password]@]host:port</code>	Simple Mail Transfer Protocol

7.2. FUTURE WORK

tcps	IP/TCP/SSL	tcps//host:port	TCP encrypted with SSL
udp	IP/UPD	udp://host:port	Plain UDP

In case a transport protocol is used for indirect traffic (see section 5.7.4), two separate addresses need to be specified, as can be seen in an example for e-mail:

1. a public address for announcing: (mailto://nico@example.org)
2. a private address for polling (imap://nico@example.org:password@imapserver.example.org)

7.2.3 Impact of decreasing packet size

The current approach does not take care of message size decreasing on its path due to removal of onion layers. This could potentially allow an attacker to gain knowledge about the communication. Thus the implications of missing padding remain to be analysed.

7.2.4 Adding Peers / Key Exchange

Currently all peers are added manually by the user. This process is tedious and prevents the network from growing quickly. There are several possibilities to support automatic discovery and addition of peers:

Peer Exchange Messages

The existing set of messages (see 5.4) could be expanded to a message type that allows sending peer information (name, address, keyid) to another peer. Furthermore, a new message type could be introduced to request a random peer from another peer, which may or may not include the public key and/or one or all addresses of the random peer as well.

PGP Key Servers

PGP keys are usually stored on public key servers. If all public keys that are used for the chat network have a special comment in them, like *EOF42KEY*, it is easy to search for participants of the network. After a peer selects a random key from a keyserver with the special comment, it can contact it by using the given e-mail address. The contacted peer can then reply with a *peer exchange message* (see 7.2.4) to submit all of its addresses.

Key Signing Parties

There are so called Key Signing Parties, at which PGP users exchange and sign their public keys, after identification verification. These parties could be upgraded to *peer exchange and key signing parties*.

7.2.5 Using Trust Levels

As the underlying technology, PGP, supports trust levels, one could use this technique to determine from which peers to accept real (non noise) messages to prevent spamming.

7.2.6 Multi User Chat

As most chat systems support multi user chat, this may be a feature to be added in a future version of the protocol. The previously described trust levels could be used to decide which people are allowed to join or send messages to a specific group.

7.2.7 Spread Usage

As can be seen in section 5.10.1 on page 52, the degree of anonymity greatly depends on the number of nodes in the network. Therefore to enhance anonymity, a certain set of nodes needs to be established, before a high degree of anonymity can be expected. To spread usage, semi or fully automatic key exchanges, as described in section 7.2.4, could be used. Additionally games or screensavers could be built, which participate in the chat network and support it by sending noise.

Bibliography

- [1] Simple mail transfer protocol. RFC 2821, Internet Engineering Task Force, April 2001.
- [2] Salman A. Baset and Henning G. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. Technical report, 4 2006. <http://www1.cs.columbia.edu/~library/TR-repository/reports/reports-2004/cucs-039-04.pdf>.
- [3] Klaus Zeuge Ben Mesander, Troy Rollo. <http://www.irchelp.org/irchelp/rfc/dccspec.html>.
- [4] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): generic syntax. RFC 3986, Internet Engineering Task Force, January 2005.
- [5] Nikita Borisov. Off-the-record communication, or, why not to use pgp. In *In WPES '04: the 2004 ACM workshop on Privacy in the electronic society*, pages 77–84. ACM Press, 2004.
- [6] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP message format. RFC 2440, Internet Engineering Task Force, November 1998.
- [7] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, February 1981.
- [8] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *INTERNATIONAL WORKSHOP ON DESIGNING PRIVACY ENHANCING TECHNOLOGIES: DESIGN ISSUES IN ANONYMITY AND UNOBSERVABILITY*, pages 46–66. Springer-Verlag New York, Inc., 2001.
- [9] George Danezis, Roger Dingledine, David Hopwood, and Nick Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *In Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 2–15, 2003.
- [10] T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, Internet Engineering Task Force, January 1999.
- [11] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *In Proceedings of the 13th USENIX Security Symposium*, pages 303–320, 2004.
- [12] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. J. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

- [13] XMPP Standards Foundation. Jabber instant messaging user base surpasses icq, 9 2003. <http://xmpp.org/xsf/press/2003-09-22.shtml>.
- [14] Freenode. Freenode irc network, 2012. <http://www.freenode.net>.
- [15] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, Internet Engineering Task Force, February 2000.
- [16] Ceki Gülcü, Ceki Gulcu, and Gene Tsudik. Mixing email with babel. In *Symposium on Network and Distributed System Security*, pages 2–16, 1996.
- [17] John A. Hoxmeier, Ph. D, and Chris Dicesare Manager. System response time and user satisfaction: An experimental study of browser-based applications. In *Proceedings of the Association of Information Systems Americas Conference*, pages 10–13, 2000.
- [18] I2P. I2p anonymous network, 2012. <http://www.i2p2.de/>.
- [19] C. Kalt. Internet relay chat: Architecture. RFC 2810, Internet Engineering Task Force, April 2000.
- [20] C. Kalt. Internet relay chat: Channel management. RFC 2811, Internet Engineering Task Force, April 2000.
- [21] C. Kalt. Internet relay chat: Client protocol. RFC 2812, Internet Engineering Task Force, April 2000.
- [22] C. Kalt. Internet relay chat: Server protocol. RFC 2813, Internet Engineering Task Force, April 2000.
- [23] A. Mayrhofer. IANA Registration for Enumservice 'XMPP'. RFC 4979 (Proposed Standard), August 2007.
- [24] A. Melnikov and K. Zeilenga. Simple Authentication and Security Layer (SASL). RFC 4422 (Proposed Standard), June 2006.
- [25] P. V. Mockapetris. Domain names - concepts and facilities. RFC 1034, Internet Engineering Task Force, November 1987.
- [26] P. V. Mockapetris. Domain names - implementation and specification. RFC 1035, Internet Engineering Task Force, November 1987.
- [27] netsplit.de. Irc networks - top 10 by comparison, 2012. <http://irc.netsplit.de/networks/top10.php>.
- [28] Network Working Group. RFC 5389 - Session Traversal Utilities for NAT (STUN). Technical report, IETF, October 2008.
- [29] J. Oikarinen and D. P. Reed. Internet relay chat protocol. RFC 1459, Internet Engineering Task Force, May 1993.
- [30] Robert Poe. Skype secrecy under attack again, 2 2009. <http://www.voip-news.com/feature/skype-secrecy-attack-022409/>.

- [31] J. B. Postel. User datagram protocol. RFC 768, Internet Engineering Task Force, August 1980.
- [32] J. B. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.
- [33] Pekka Riihonen. Silc protocol white paper, 10 2003. http://www.silcnet.org/support/documentation/wp/silc_protocol.php.
- [34] Troy Rollo. A description of the dcc protocol. <http://www.irchelp.org/irchelp/rfc/dccspec.html>.
- [35] P. Saint-Andre. End-to-End signing and object encryption for the extensible messaging and presence protocol (XMPP). RFC 3923, Internet Engineering Task Force, October 2004.
- [36] P. Saint-Andre. Extensible messaging and presence protocol (XMPP): core. RFC 3920, IETF, October 2004.
- [37] P. Saint-Andre. Extensible messaging and presence protocol (xmpp): Instant messaging and presence. Internet RFC 3921, October 2004.
- [38] P. Saint-Andre. Mapping the extensible messaging and presence protocol (XMPP) to common presence and instant messaging (CPIM). RFC 3922, Internet Engineering Task Force, October 2004.
- [39] P. Saint-Andre. Internationalized Resource Identifiers (IRIs) and Uniform Resource Identifiers (URIs) for the Extensible Messaging and Presence Protocol (XMPP). RFC 4622 (Proposed Standard), July 2006. Obsoleted by RFC 5122.
- [40] P. Saint-Andre. A Uniform Resource Name (URN) Namespace for Extensions to the Extensible Messaging and Presence Protocol (XMPP). RFC 4854 (Informational), April 2007.
- [41] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard), March 2011.
- [42] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 6121 (Proposed Standard), March 2011.
- [43] Nico Schottelius. cconfig, 08 2005. <http://nico.schotteli.us/papers/linux/cconfig/cconfig.pdf>.
- [44] Alexandr Shutko. Oscar (icq v7/v8/v9) protocol documentation, 2 2005. <http://iserverd.khstu.ru/oscar/>.
- [45] T. Krivoruchka T. Bova. Reliable udp protocol (internet-draft), 02 1999. <http://tools.ietf.org/html/draft-ietf-sigtran-reliable-udp-00>.
- [46] Unknown. Skype protocol source code. <http://cryptolib.com/ciphers/skype/>.
- [47] Unknown. Microsoft changes skype supernodes architecture to support wiretapping, 5 2012. <http://skype-open-source.blogspot.ch/2012/05/microsoft-wiretapping-on-skype-now.html>.

BIBLIOGRAPHY

- [48] Unknown. Mixmaster, 2012. <http://mixmaster.sourceforge.net/>.
- [49] Nart Villeneuve. An analysis of surveillance and security practices on china's tom-skype platform. Technical report, 10 2008. <http://www.nartv.org/mirror/breachingtrust.pdf>.
- [50] Wikipedia. List of device bit rates — wikipedia, the free encyclopedia, 2012. http://en.wikipedia.org/w/index.php?title=List_of_device_bit_rates&oldid=493954817, [Online; accessed 3-June-2012].
- [51] Wikipedia. Silc — wikipedia, die freie enzyklopädie, 2012. <http://de.wikipedia.org/w/index.php?title=SILC&oldid=103355799>. [Online; Stand 1. Juni 2012].
- [52] Wikipedia. Skype — wikipedia, the free encyclopedia, 2012. <http://en.wikipedia.org/w/index.php?title=Skype&oldid=495309950>, [Online; accessed 2-June-2012].