View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DSpace at Tartu University Library

UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Information Technology

Märt Bakhoff

Secure General Purpose P2P Overlay Network

Bachelor's thesis (6EAP)

Supervisors:
Meelis Roos
Margus Niitsoo

Author:	....	"......" 2013
Supervisor:	....	"......" 2013
Professor:	....	"......" 2013

TARTU 2013

Abstract

The Internet was designed to provide end-to-end connectivity between all the connected hosts. Due to the depletion of IPv4 addresses and widespread deployment of NAT, a lot of devices are no longer directly reachable over the Internet. This paper describes a secure peer-to-peer protocol that is capable of working around NAT by using unrestricted peers as relays. The protocol builds on common cryptographic tools to provide seamless authentication and encryption without requiring difficult key exchange procedures or in-advance key signing by using the hash of the peer's public key as his identity on the network. Some of the main security issues are discussed and a proof-of-concept prototype is implemented to demonstrate the functionality of the protocol.

Table of Contents

Abstract....2
Introduction....4
Chapter 1: Networking and security fundamentals....5
1.1 OSI model....5
1.2 Network Address Translation....5
1.3 Peer-to-Peer networking & Distributed Hash Tables....6
1.4 Symmetric and asymmetric cryptography....7
Chapter 2: Connection establishment....10
2.1 Working around NAT....10
2.2 Routing....11
2.3 Security....11
2.4 Similar solutions....13
Chapter 3: Proof of concept....16
3.1 Overview of the prototype....16
3.2 Overview of the functionality....16
3.3 Description of the prototype and program flow....17
3.4 Description of the protocol....21
3.5 Summary of the proof of concept....22
Conclusions....23
Summary (Estonian)....24
References....25
License....27
Appendixes....28
Appendix A: Specification of the packets types....28

Introduction

The Internet in it's current state is fundamentally broken [1][2][3] due to the widespread deployment of Network Address Translation (NAT), which makes end-to-end connections between many of the hosts impossible without extra configuration. In addition, a lot of commonly used protocols do not provide any authentication or encryption which is becoming increasingly relevant considering the sensitivity of information being sent over the Internet.

The purpose of this paper is to create a secure peer-to-peer protocol (and a prototype implementation) that works in OSI layer 5/6 and is capable of working around the connectivity issues created by NAT by using peers with unrestricted connectivity as transparent relays. The protocol will enable the secure communication between all participating nodes even when some of them are restricted by NAT or firewalls while requiring no special configuration of the routers and no central servers that could become a bottleneck. The protocol can be extensible to carry any user data, provide authentication of the peers and encryption of the user data.

While many protocols for traversing NAT, creating peer-to-peer networks and securing user data already exist, none of them provide reasonable security without central servers and without complicated key exchange processes. The protocol designed in this paper will attempt to create a secure network that's easy to use and requires very little configuration from the end user. It could then be used for a wide range of purposes from instant messaging and file sharing to building SSH-like security tunnels.

The work focuses mainly on the security issues related to sending information over the public Internet and establishing connections to other peers on the network. The advanced routing, path finding and data flow optimization is not the main focus of this paper and can be researched further as a separate topic.

Chapter 1: Networking and security fundamentals

1.1 OSI model

The OSI model [4], formally known as Open Systems Interconnection Reference Model, categorizes the different functions of a open communications network into a set of standardized abstraction layers. There are seven layers in total:

1. Layer 1: physical layer
2. Layer 2: data link layer
3. **Layer 3: network layer**
4. **Layer 4: transport layer**
5. **Layer 5: session layer**
6. Layer 6: presentation layer
7. Layer 7: application layer

For the Internet, the network layer represents the IP addressing system that is used to route packets globally around the Internet. A single IP address should identify a single host on the Internet and the host should be reachable by that address. The transport layer can be used to refer to protocols on top of IP such as TCP and UDP that deal with multiplexing multiple connections in between a pair of IP addresses. Finally, the session management and state tracking part of the TCP protocol is linked to the session layer in the OSI model.

1.2 Network Address Translation

The length of an IPv4 address is 32bits. This means that there are exactly 2^{32} possible IP addresses and not all of them are usable. 2^{32} is less than the number of computers in the world which means that not enough IP exist to connect every device in the world.

Sending a packet over the Internet requires the recipient to have an IP address and the sender knowing that IP address. If the sender is expecting a reply (which is usually the case) then the sender must also have an IP address that is included in the sent packet. Therefore if one of the parties does not have an IP address then they cannot communicate.

To fix the problem of some of the hosts not having an IP, parts of the IP address space were designated as private IP ranges [5] (10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16) that were to be not unique and not globally routable. Most of the hosts could then be allocated an IP address from one of the private ranges and all the hosts within the same private network could easily communicate with each other.

As long as the hosts with a private IP only need to communicate within the same private network, everything works. But usually the hosts also need to contact hosts outside their private network (such as web servers and other public services). Since they don't have a globally routable IP address they could not get an answer even if they would manage to send a request.

A solution called Network Address Translation (NAT) was invented to fix that problem. A single public IP address is assigned to the router connecting the entire private network and all traffic is routed through that single "gateway router". The gateway would process all packets going out through it, replacing the source IP of the packets with the public IP of the gateway (Illustration 1). It would then process the incoming packets, changing the destination IP to the private IP of the real destination host.

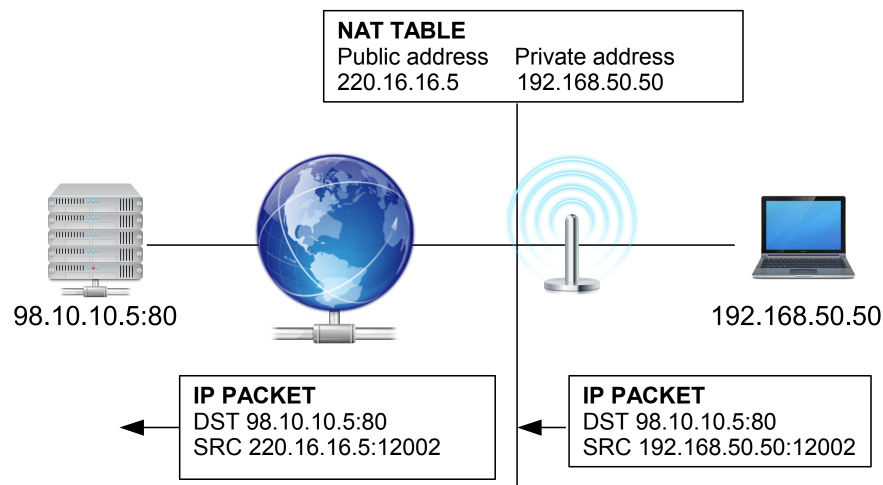

Illustration 1: NAT IP translation [6]

Different connections could be distinguished by their source and destination port numbers and everything would work.. as long as the connection was initiated from inside the private network. Incoming connections to the hosts behind the gateway however, would be impossible since the gateway would have no way of knowing which host is the real destination. That way NAT fixes the Internet but also breaks a fundamental [7] part of it – end to end connectivity between hosts.

1.3 Peer-to-Peer networking & Distributed Hash Tables

Peer-to-peer (P2P) networking is a communication model where every peer is equal from the protocol's point of view. It is used for many purposes such as sharing storage space, messaging, decentralized load distribution etc. The Bittorrent protocol has proven that P2P networks can be very scalable and resilient to network blocking and Skype has proven that

P2P networks can be reliable enough for real time communications. The best part of P2P networks is that they are very easy to set up – no dedicated server is needed [8], the peers simply need to be able to connect to each other.

Since P2P networks usually have a lot of peers, the peers need a way to find resources in the network in a efficient way. Some of the networks (Kad, Bittorrent) use a system called Distributed Hash Tables [9] (DHT) for looking up data on the network. The DHT principles can be easily explained by an example of a typical Bittorrent search.

Every peer in the network generates a pseudo random fixed length identifier for itself (peer_id). When creating a new torrent, the contents of the torrent are hashed and the fixed length hash (for example SHA1 has a length of 160bits) will become the identifier of the torrent (info_hash). To start sharing a torrent on the network the sharing peer will first look for peers with a peer_id closest to the info_hash of the torrent it wants to share. The search is recursive with every level of the search containing peers with a closer peer_id [10]. Once a set of peers with close enough peer_ids has been found, the sharing peer will save it's contact information with the found peers.

To look up a torrent on the network, all the searcher needs to know is the torrent's info_hash. The searcher will do exactly the same search as described above and find the same peers with a peer_id close to the info_hash being searched. It will then receive the contact information of the sharing peers from the peers that were found in the search.

Because of the randomness of the peer_ids and info_hashes, the searches are automatically distributed between different parts of the network so that no part of the network gets overloaded. It is also very difficult to disrupt the network without disabling a large part of the network at the same time which makes it resistant to outages.

1.4 Symmetric and asymmetric cryptography

1.4.1 Public key cryptography

The most common and intuitive way of securing data is through using shared secrets such as passwords. However using pre-shared secrets is not the only way of securing data.

Public key cryptography is a system where each party has special key pair that consists of a public key and a private key. As the names suggest, the private key is a key that must only be known to the owner of the key while the public key can and should be known by everyone. The keys work so that if one key of the pair encrypts some data then only the

other key of the pair can decrypt it and vice-versa. This has two useful options [11]:

- If some data is encrypted using someone's public key then only the person who has access to the corresponding private key will be able to decrypt the data.
- If some data can be decrypted using someone's public key then it could only have been encrypted by someone having access to the corresponding private key.

1.4.2 Digital signatures

Digital signatures are a way for a owner of a private key to authenticate a certain piece of data. Anyone with a copy of the data, the corresponding public key and the signature will then be able to mathematically verify [12] that the data was authenticated by the owner of the private key.

Creating a signature works by hashing the data using a commonly known hash function (such as SHA1) and encrypting the hash using a private key. The hash can later be decrypted using the public key, which indicates that the signature was created by the owner of the private key. To verify the hash, the verifier must hash the same data using the same hash function and compare the result with the hash that was decrypted using the public key. If the hashes match then the signature is valid and the data is authenticated.

1.4.3 Hash-based message authentication codes (HMAC)

HMAC is a way to verify the integrity and authenticity of a piece of data. It is the symmetric analogue of a digital signature as it uses a shared secret instead of a key pair for authentication.

To create a HMAC [13] the data is hashed using a common hash function, the resulting hash is XORed with a pre-shared secret and the result is once again hashed. To verify a HMAC the verifier will simply create a HMAC of the same data using the same secret and if the HMACs match then the message is authenticated (real implementations are a little more complicated but the idea is the same).

1.4.4 Encryption

Encryption is the process of turning some readable data (plaintext) into a unreadable ciphertext. The purpose of encryption is to protect the data from being read by unauthorized parties [14]. Encryption is done using a symmetric secret key or a asymmetric key and can be later reversed (decryption) using the same symmetric key or the counterpart of the asymmetric key.

The encrypted data is protected as long as the key is protected (assuming the algorithm is not broken). However if the key is leaked then all data encrypted using that key is exposed as well. This is especially a problem if an attacker is able to collect the encrypted data for a period of time and later discover the key.

1.4.5 Diffie-Hellman key exchange

Diffie-Hellman [15] key exchange is a method for two or more parties to agree on a shared secret using only plain text messages even in the presence of an eavesdropper. The method uses the discrete logarithm problem as it's core but the details are outside the scope of this paper. The Diffie-Hellman key exchange is only secure if the eavesdropper cannot modify the communications. Otherwise additional authentication of the messages is required to protect against man-in-the-middle attacks.

Chapter 2: Connection establishment

The main problem that must be solved in this thesis is connecting to a remote host that may be behind NAT or firewall. Reaching such host directly is impossible unless there has been prior contact and ports are already open. It can't be assumed that this is always the case which means an alternative way of connecting to the remote host is needed.

Having a central server for managing routes between hosts can be unreliable, costly and does not scale [16] well. However the problem of sharing resources in a distributed and scalable way has already been invented. DHT networks do not require a central server and are very tolerant of network failures. By considering nodes themselves as resources a DHT network can be built and used for searching routes to the remote host.

2.1 Working around NAT

Fortunately the central part of DHT model is unaffected by NAT and firewalls. During the initialization of every node the node runs a self search and stores his contact info in his neighboring nodes. To do this the node sends out packets to his neighbors using the UDP protocol. The firewall/NAT must then create a port mapping for the outgoing connection to allow responses to the packets sent out. Since UDP is stateless, it is a lot harder to block [17] incoming UDP connections because the firewall can't distinguish between new connections and existing connections.

The nodes can keep using these same UDP sessions for communicating and the same connections can also be used to relay communications. Any node can then use the neighboring nodes as proxies to connect to the node behind firewall (Illustration 2). While the firewalled node is able to connect to at least one neighboring node and establish a bi-directional connection, the firewalled node can be reached through that neighbor.

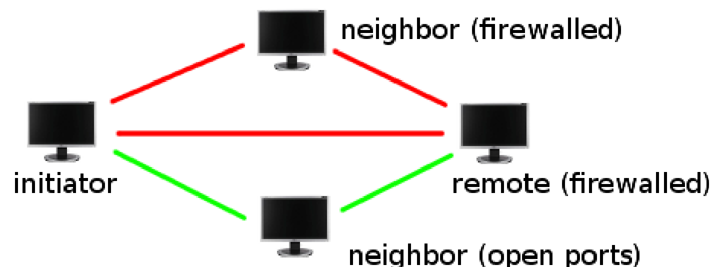

Illustration 2: reaching remote via neighbor

Simply tunneling data through unknown third parties (or over the public Internet) is a potential security issue and requires additional protection from the possible attacks. All

data can be freely [18] logged and modified which means that the described connection would be vulnerable to man-in-the-middle attacks. To avoid that, the connection would need some kind of integrity checks and encryption.

2.2 Routing

The only thing the initiator of the connection knows about the remote host is it's *id*. The initiator doesn't know the remote node's IP address or port number neither does it know if the remote node is behind a firewall.

The initiator can use the only thing he knows about the remote – it's *id*. The *id* always has a fixed length which is the same length as every other *id* on the network. This makes it a perfect node's identifier on the DHT network mentioned earlier. Using the *id* as the DHT network identifier allows the use of the DHT network to look up the remote node's information using only the node's *id*.

There are three cases that can happen:

1. the initiator finds the remote node and can connect to it,
2. the remote node is behind a firewall but a neighbor is found who has a connection to the remote node,
3. no neighbors can be found that have a connection to the remote.

If the initiator can connect directly then there's no problem. Otherwise we need to find a neighbor that has a connection to the remote node and use him as a relay. This can be done during the same search that we use to find the remote node with a simple modification to the usual DHT search. The search results should simply include whether the replying node has a direct connection to the searched node. The last case – not finding any neighbors in a reasonable search depth – indicates that the searched node can't make a outbound connection or is not in the DHT network.

2.3 Security

Some of the security risks the data must be protected against when traversing proxies [19] and insecure networks include:

1. host id spoofing – sending data using someone else's id as the sender id,
2. eavesdropping – intercepting data that is meant for someone else,
3. man-in-the-middle attacks (MITM) – modifying data that is on route to someone else,
4. packet data corruption – unexpected changes to data meant for someone else.

These problems are common in all connections going over the Internet and there is already a solutions for fixing most of them in the form of TLS. Unfortunately TLS only works with TCP which can be easily blocked by firewalls. There's also an UDP variant of TLS called Datagram TLS (DTLS) but it's a complicated RFC with the latest version (1.2) having almost no implementations (and no Java implementations/bindings at the time of this writing). DTLS also wouldn't work well through ad-hoc proxies since the IP addresses and port numbers could randomly change but the DTLS identifies [20] the remote host by it's IP and port combination.

Since we only need a small subset of TLS features we can easily implement them in Java using Java's built in JCA [21] (Java Cryptography API). More precisely we need authenticated integrity checks to prevent MITM, id spoofing and corruption and encryption to prevent eavesdropping.

To be able to do any kind of authentication we must establish some kind of public key infrastructure. For obvious reasons the usual CA approach [22] does not work. We need a way to easily create new identities, be able to share them by a simple *id* and also protect the *id*s against spoofing. There's a simple way of linking the *id*s to identities in a way that makes spoofing the *id* very difficult. If an asymmetric key pair defines the identity, the fixed length *id* can be easily and deterministically calculated from the public key using a standard hash function such as SHA1. This approach is already used by widely known and proven protocols such as SSH and PGP. The public keys can be distributed using the existing DHT network.

Using the idea of calculating the peers' *id*s from concrete public keys makes it a lot easier to use common cryptographic tools. Once a route to the remote peer and his public key have been found from the DHT network the initiator can create a reasonably secure encrypted tunnel to the peer. The first step is doing the standard Diffie-Hellman (DH) key exchange to agree on a session key. Using a DH session key with symmetric encryption instead of simply sending the data to the peer using public key encryption is required for perfect forward secrecy [23] – protecting the contents of previous sessions even in the event of a private key being exposed.

The only problem with DH key exchange is that it's susceptible to man-in-the-middle attacks. If an attacker can modify the packets sent between the parties doing the DH key exchange then he can also impersonate both parties. This allows the attacker to do a DH key exchange with both of the them and become a transparent proxy. The attacker can then

decrypt and modify all the data without neither parties noticing the proxy.

This attack is only possible because the DH key exchange doesn't authenticate the parties doing the key exchange. Since the peer initiating the session knows the *id* of the peer he's trying to establish a session with, he can also get the peer's public key from the DHT network. Since a peer's *id* is calculated from his public key, the received public key can be checked. Both peers can then require the DH key exchange packets to be signed using by the same keys that the *id* is calculated from. If the DH key exchange packets are signed then the attacker cannot impersonate the peers and the attack will become impossible.

Once the session key is agreed on, it can be used for encrypting further communications as well as for creating the HMAC integrity checks. Any symmetric encryption algorithm will do as long as both peers support it. This solves both the eavesdropping and packet corruption issues. This is also the last step of the tunnel setup.

2.4 Similar solutions

The idea of using UDP for passing firewalls and NAT is not new. There are many protocols and applications already using a similar approach. This chapter will give a brief overview of some of the other protocols and their weaknesses.

2.4.1 Interactive Connectivity Establishment

A lot of existing UDP NAT traversal techniques are using or are similar to RFC 5245 – Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. The purpose of ICE is the same as this protocol's – to enable connections between hosts behind NAT (and firewalls blocking incoming connections). It combines [24] Session Traversal Utilities for NAT (STUN; protocol for detecting NAT and obtaining one's public IP address) and Traversal Using Relay NAT (TURN; protocol for using relays to pass packets between hosts behind NAT) and first tries to establish a direct connection between hosts by first detecting their public IP addresses and if that fails it uses a relay as a fallback.

While ICE is a open standard and STUN/TURN servers are not bound to any single service provider, it still has the problem of requiring the use of special servers to be able to function at all. Protocols that require such non commodity resources for it's core functionality will always remain dependent on these resources and will fail once the resources become unavailable.

2.4.2 Skype

Skype is a well known peer-to-peer VOIP network that has proven to work well even with a lot of it's peers behind NAT. Unlike ICE that uses special STUN/TURN servers Skype designates some of it's peers as special "supernodes" [25]. Any node can be a supernode as long as it's publicly reachable and has enough free resources. Other peers can use these supernodes a lot like STUN/TURN servers to coordinate and relay communications (see Illustration 3).

The communication is also encrypted, allegedly using a combination of AES and RSA. Since it's a proprietary protocol and the details have never been published it's hard to tell how secure the encryption of Skype really [26] is. Also Skype is not entirely P2P – all of the authentication and login management is coordinated by a small number of Skype servers whose addresses are hardcoded into the Skype executable.

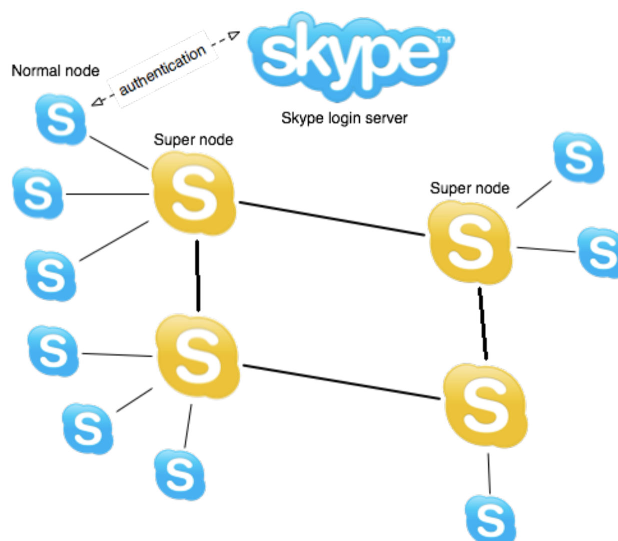

Illustration 3: Skype supernodes (source: [27])

2.4.3 Brunet

Brunet is a structured P2P system that uses DHT and recursive routing to create connections in the presence of NAT, firewalls and Internet outages. It is very similar to the idea presented in this paper but there are some important differences.

Brunet has a strong focus [28] on routing and it maintains a lot more detailed routing tables. It uses that routing table to send messages mainly through existing connections and only occasionally creates new shortcut connections (as seen in Illustration 4). This allows for faster connection establishment but unless a shortcut is created the latency of the

connection is equal to the sum of latencies along the chosen path. In contrast this paper recommends using the DHT network mostly for finding the peers to create a new direct connection and uses existing connections only when the peer is not directly reachable due to connectivity constraints.

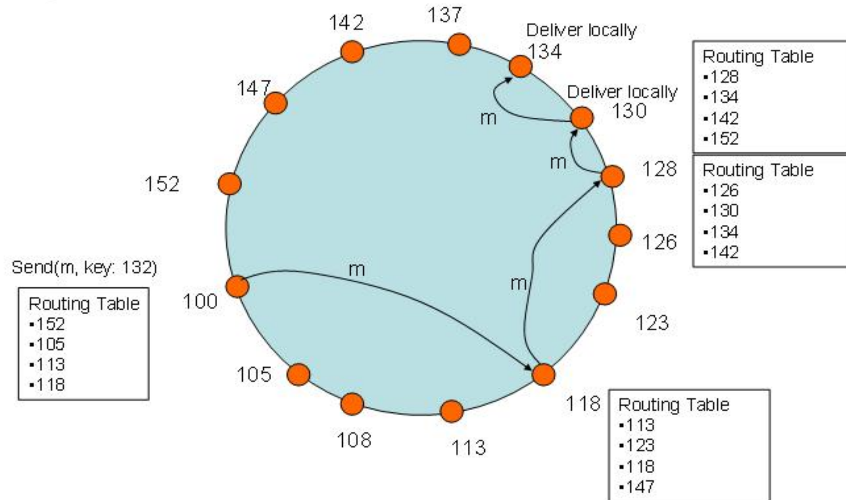

Illustration 4: Brunet routing (source: [29])

In the security aspect, Brunet has optional SSL support and it can build a secure overlay. However the resource identifiers of Brunet are not directly connected to the SSL keys which means additional key authority is required to coordinate security.

Chapter 3: Proof of concept

3.1 Overview of the prototype

To prove that the idea of using a secure P2P network to bypass NAT is possible and fairly easy to implement a prototype implementation was written.

The prototype was supposed to implement the minimum set of functionality to be able to handle most of the use cases of the proposed network. In the process of writing the prototype a binary protocol for transmitting data between the nodes was invented and tested. It turned out that writing the prototype and the protocol in parallel is a great way of finding errors and shortcomings in the protocol since everything can be tested quickly and all the problems come up early in the design process.

The implementation was written in Java programming language using plain UDP sockets, Java Cryptography API (JCA) and utilities from Apache Commons. Luckily JCA was able to provide all the cryptographic tools required for the implementation of the prototype and the final solution doesn't require any external dependencies besides the mentioned Apache Commons libraries. It can be easily built using maven and works with both Oracle's Java SE 7 and OpenJDK7.

The code for the prototype is available at https://bitbucket.org/mbakhoff/whatever. This paper describes the protocol as it is implemented in prototype version 1.1.

3.2 Overview of the functionality

For the peers to be able to securely and reliably communicate (whether directly or through relaying peers) the protocol would need to provide the following functionality:

1. detecting and retransmitting lost packets
2. exchanging public keys
3. authenticating peers using digital signatures
4. using DHT queries to find routes to a peer by his network *id*
5. negotiating the session key using Diffie-Hellman key exchange
6. encrypting packets and verifying the integrity of encrypted packets
7. relaying packets through peers already having a direct connection to the destination

There were three cases that were considered when writing the prototype:

1. both peers can be directly reached,
2. one of the peers can be directly reached,

3. both of the peers are behind NAT

3.3 Description of the prototype and program flow

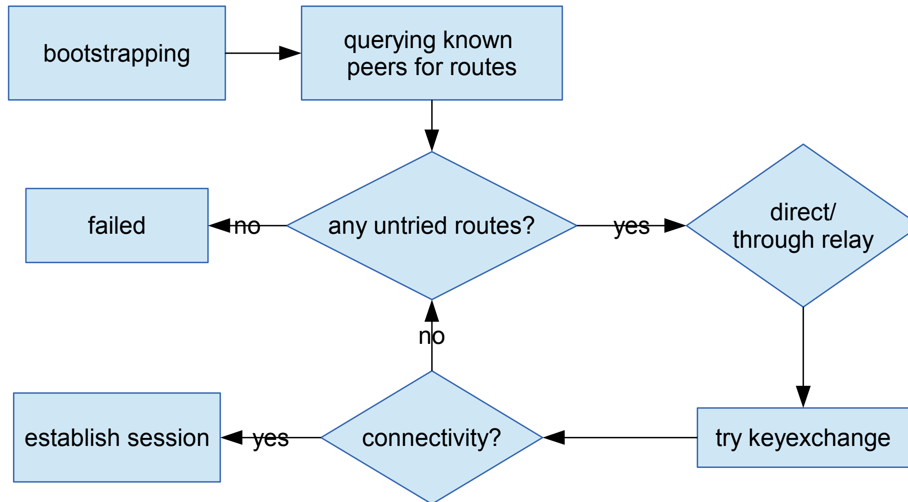

Illustration 5: general program flow

3.3.1 Bootstrapping

The program flow of the prototype is very simple. The general overview is shown on Illustration 5. The first step is always bootstrapping the network. Bootstrapping is the process of connecting to a known node in the peer-to-peer network. The new node can find routes to other peers and use it to advertise routes to itself through the bootstrap nodes. To bootstrap the program must know a route to at least one peer that can be directly reached. A route is the combination of the peer's IP address, port number and network *id*. Any other peer can be a bootstrap because the bootstrap process is like establishing any other connection inside the network.

For complete functionality the joining peer should push it's public key and routes to himself to his (key-wise) neighbors after bootstrapping the network. That can be accomplished by doing a self search on the DHT network and connecting to the closest results. Similar to the Kademlia [30] protocol, the network uses XOR to calculate the distance between nodes. After leaving his contact info to the the nodes found through self search any other node on the network could find routes to the peer by simply doing a search for him on the network and contacting the same peers as the original did.

The described self search and publish functionality is not implemented in the prototype. The purpose of the prototype is to demonstrate NAT traversal and a common bootstrap

node is all that's required for it. However the protocol implemented by the prototype would easily allow the implementation of recursive DHT search and the described self search could be added without changing the wire level protocol.

3.3.2 Finding routes

After bootstrapping the network the peer can start the process of looking up routes to it's destination. All that's required is the network *id* of the destination peer. Looking up the routes can be done by sending DHT queries to already discovered peers. To keep the search efficient the searching node must only send queries to peers whose network *id* is closest to the network *id* of the destination. When doing a recursive DHT search, each level of search would need to include peers whose *id* is less than the *id*s of the last search level. Each queried node would also return only the peers whose *id* is closest to the search target. In the case of the prototype implementation the amount of results is limited to 30 peers.

The responses to DHT queries contain a set of routes to the destination. There are two kinds of routes – direct routes and routes through proxies. A direct route includes the network *id* and socket address of the resulting peer. The socket addresses is the combination of an IP address and a port number. Both IPv4 and IPv6 addresses are supported. The types of addresses can be easily distinguished in the protocol by their length in bytes – IPv4 addresses are always 32bit long while IPv6 addresses are always 128bit long.

3.3.3 Detecting broken routes

After routes to the destination have been found the most complicated process begins. The search results should include one or more direct routes (IPv4, IPv6 etc) and one or more proxy routes. At first there's no way of knowing which ones are usable or fastest. All found routes must be tried until one that works is found. The only way to see if a route is up is to try to use it and that requires getting a confirmation from the destination. UDP does not provide [31] that functionality.

The protocol implements a simple system for tagging packets and acknowledging received packets. To enable acknowledgments each data packet sent must contain a transmission id (integer serial number). When the peer receives a data packet he must send a confirmation to the sender. If there no other data to be sent besides the confirmation, an "empty" confirmation packet is sent containing only authentication data and the transmission id being confirmed. In this case the transmission id of the confirmation packet is left empty

(value set to zero). Otherwise the confirmation is bundled with the response and no separate confirmation packet is sent (Illustration 6). The transmission ids don't have to be globally unique – since every packet is also tagged with sender's *id*, transmission ids can be distinguished by the related senders' *id*s.

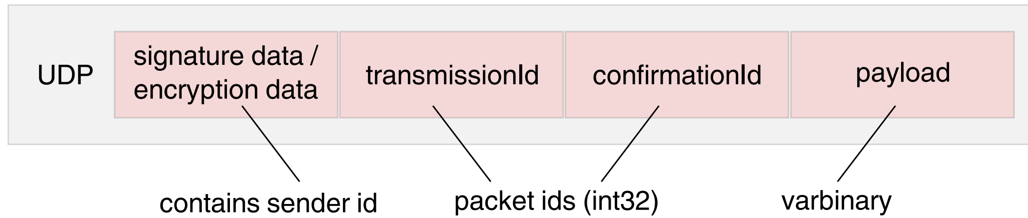

Illustration 6: confirmation packet structure (conceptual)

As long as all the packets are correctly tagged with transmission ids and all peers honor the confirmation system, the sending peer should get confirmations for every successfully delivered packet. That makes it possible to track the state of every packet, to create timeouts for every packet and to detect dropped packets by confirmation timeouts. Being able to detect dropped packets can be used to check if a peer is reachable by the route being tested.

3.3.4 Selecting the route

Once the program has a set of potential routes to a peer and a way of detecting broken routes, it need to find a route it can use. There are several ways to to that but let's just consider a naive and easy to implement solution and a smarter, more difficult solution.

The easy solution is to simply try each route sequentially. To be sure that random packet loss doesn't affect the results several retries can be made before giving up on a route. If no confirmations are received even after multiple tries it should be safe to assume that the route is broken and try the next route. However it turns out that this approach has some downsides.

Firstly if several broken routes are tried with multiple retries each having a long timeout, there might be a significant delay until a good route is even tested. Secondly, there is the problem of the remote peer having to find a reverse route for sending back the confirmations. If the remote peer hasn't already found a working route, he would also have to run the same route selection algorithm where each route is tried sequentially. This could mean that once the remote peer finds a good route the original packet has already timed out. Fortunately the remote peer can use the same proxy that forwarded him the packet to relay the responses back to the original sender. This wouldn't be the best route if the

original has ports open and could the reply could be sent directly (creating a hole in NAT for further use).

Currently the prototype implements the simple route selection described above but there's room for improvement. Theoretically the routes could be tested in parallel. The same packet could be sent through all potential routes and the route that would return the confirmation in the shortest amount of time would be selected for future communications. If non of the routes get a response the routes could be retried the same was as with the simpler solution and there would be no downsides. The implementation of this would of course be a lot more difficult as the sending peer would need to track multiple transmissions, create events for the transmissions timing out or succeeding and progressively rank different routes to every peer.

3.3.5 Public key exchange

Even the simple route selection algorithm will eventually manage to find a working route if there is one. When that happens the initiator can start a conversation with the new peer by sending him a KeyPush packet containing our public key. The response to KeyPush should be a packet containing the public key of the responding peer. Sending one's public key as the first message is only necessary on first contact but it's a good idea to start every new connection with a KeyPush because there's no way of knowing whether the peer still has the sender's key cached from the previous contact.

Ensuring that the receiving peer has the sender's public key is critical because the public key is used to check the digital signatures in messages that are sent before a session key has been set up. Not only does the signature protect the packet against corruption [32], it also is the only valid way for a peer to authenticate itself to another peer besides using a HMAC created with a valid session key. If the receiving peer does not have the sender's public key then it can't verify the sender any of the sender's messages which means it also can't send any confirmations or responses to the sender. From the sender's perspective no confirmations simply means that the route is broken and establishing the connection will eventually fail.

3.3.6 Session establishment

After the KeyPush is sent as the first packet and the response with the peer's public key has been received, several things can happen. If the initiator plans to send a lot of data to the peer or it wishes to use the peer as a proxy to itself, then they should establish a session

key that it can be used to encrypt the packets using symmetric encryption and also to calculate the message HMACs. Otherwise, if the initiator could simply continue sending the simple unencrypted but signed messages. For example this would happen if the initiator simply wishes to do a DHT query for the routes to another peer or to request the public keys of other peers.

To establish the session key the initiator would simply send a Diffie-Hellman request to the peer and receive the response with all the information required to complete the key exchange. The packets are protected by digital signatures [33] like the KeyPush packet was thus eliminating the main weakness of Diffie-Hellman – man-in-the-middle attacks. The Diffie-Hellman key exchange allows two or more parties to agree upon a mutual secret using plain text messages even in the presence of a eavesdropper. This shared secret can then be used as the session key for encrypting and verifying further messages. Using symmetric cryptography enabled by a shared secret is also good for performance [34] since asymmetric operations are usually much slower than their symmetric counterparts. Lastly, great care must be taken to dispose the public key after the session has ended. If a session key is reused, all previous sessions encrypted using that session key are a risk of being exposed.

Establishing the session is the last step in the protocol's connection establishment functionality. After that the route can be used in any way necessary to carry the user data in a secure manner. The protocol can also be easily extended to contain other types of messages. Extending the protocol is explained in the section "3.4 Description of the protocol".

3.4 Description of the protocol

This chapter will explain the protocol on the byte level. It will give an overview of the principles of building a packet, the details of encoding different data structures and the detailed descriptions of different packet types.

The protocol assumes that 1 byte equals 8 bits. Most of the data is encoded in big-endian [35] (network byte order) unless specified otherwise. All simple data structures have a predefined fixed length and all variable length fields are length-prefixed. The variable length data structures are divided into three categories: *ShortData,* which is an int16 prefixed variable length data field, *Data*, which is an int32 prefixed variable length data field and strings, which are UTF-8 encoded and int16 length prefixed. The detailed byte

level description of each packet of the protocol is included in Appendix A: Specification of the packets types.

The final UDP payload data is a chunk of structured data that consists of several components (called packets) that are nested inside each other. There are two kinds of packets that make up the final payload: terminal packets that contain the main data (such as public keys, user data etc) and container packets that contain some metadata and wrap another packet. For example, the Diffie-Hellman exchange request is a terminal packet that should be wrapped inside a container packet which contains the transmission id which in turn is wrapped inside a container packet that has a digital signature for all the wrapped data.

The outermost packet of a payload must always be either SessionData (encryption+hmac) or SignedData (digital signature) packet. Both of these packets cryptographically guarantee the identity of the sender. All packets without strong identification data must be dropped. Another requirement is that at least one of the inner container packets must be a TransmissionTag packet to support the packet confirmations scheme. Without the confirmations the peers would have no way of detecting broken routes and lost packets.

Since each packet starts with an unique opcode (container id), new functionality can be easily added by creating new packet types. The only requirement is that the opcode must be unique and both the sender and receiver must implement the packet type.

3.5 Summary of the proof of concept

The protocol and it's proof of concept implementation fulfill their goals. The protocol protects against all the attacks described in chapter 2.3 Security. The configuration required to join the network and contact other peers is minimal and doesn't require any complicated key setup. The prototype can also demonstrate the ability to contact peers behind NAT at least one peer with a open session can be found.

Conclusions

The original objective of this paper was to design a secure peer-to-peer network that is not broken by NAT. The objective was successfully completed – a protocol that fulfills these requirements was designed and implemented using commonly available tools and algorithms. The paper gives a byte level specification of the protocol and also demonstrates how the cryptographic algorithms built into the Java programming language can be used to implement the key components of the protocol.

The protocol is easy to use and does not require any complex key exchanges or in advance key signing. The authentication works by using the hash of a peer's public key as his identity on the network. This will make it very difficult to impersonate anyone and very easy authenticate everyone. Other security issues such as eavesdropping and man-in-the-middle attacks were also discussed and the protocol contains means for mitigating these basic attacks.

The created prototype can successfully route data between peers as long as at least one of the peers is directly reachable over the Internet and it can also successfully bridge IPv4 and IPv6 connections. In addition both the protocol and the prototype are easy to extend to carry any user data. The prototype may be further developed into a fully functional solution that has recursive route finding, flow controls and a easy to use API.

// Summary (Estonian)

Bakalaureusetöö: Turvaline üldotstarbeline võrdõiguslikest sõlmedest koosnev kattevõrk

Kuigi Internet on viimastel aastatel tohutu kiirusega arenenud, on suur osa Interneti potentsiaalist veel kasutamata, sest NAT ja tulemüürid muudavad otspunktide vaheliste otseühenduste loomise keeruliseks. Lisaks on Interneti üldine areng privaatsuse ja turvalisuse osas olnud pigem tagasihoidlik. Enamus kasutatavast tehnoloogiast saadab sõnumeid üle Interneti puhtal inimloetaval kujul, mis muudab nende pealtkuulamise triviaalseks.

Selle töö eesmärk on vabalt kättesaadavaid vahendeid ja algoritme kasutades luua turvalist andmesidet võimaldav protokoll, mis oleks võimalikult vähe mõjutatud erinevatest segajatest nagu NAT ja võrkukatkestused. Kirjeldatakse sellega seonduvaid probleeme ja olemasolevaid lahendusi. Protokolli kasutuskõlblikuse tõestamiseks implementeeritakse ka lihtne prototüüp, mis suudab demonstreerida protokolli põhifunktsionaalsust.

Fookuses on kaks suuremat probleemi: ühenduste loomine läbi NAT tulemüüride ja nende ühenduste turvalisuse tagamine ilma keerulise tehnilise ettevalmistuseta. Kuigi nii NAT kui ka turvalisuse probleemile on eraldiseisvaid lahendusi leitud, ei ole nende kombineerimine võimalik, sest nad eeldavad kas keskseid servereid, otspunktide vahelisi otseühendusi või olemas olevat CA põhist avaliku võtme infrastruktuuri (PKI). Omaette probleem on ka ühenduste turvalisuse tagamine ilma tavakasutajalt võtmete genereerimise ja allkirjastamise lootmise.

Lahenduseks disainitakse hajus protokoll, mis loob DHT võrgu, kus igal sõlmel on avalik võti ja sellest tuletatud *id*. DHT võrk võimaldab sõlme *id* järgi otsida ja pakub NAT taga olevate sõlmedeni jõudmiseks vahendajaid. Palju tähelepanu on pööratud turvameetmete integreerimisele ja automatiseerimisele. Selle tulemusena on ainus võrguga liitumiseks vajalik seadistus mõne olemasoleva võrguliikme IP aadress, pordi number ja *id*. Protokoll oskab andmeid krüpteerida, andmete terviklikkust kontrollida ja saatjat autentida. Diffie-Hellmani võtmevahetusega saadud sessioonivõtmete süsteemi kasutamine tagab varasemate sessioonide konfidentsiaalsuse isegi *id*-võtmete ründaja kätte sattumisel.

References

1. RFC 2993 "Architectural Implications of NAT" – checked 30.04.2013
   https://tools.ietf.org/html/rfc2993#section-6
2. "NAT and VOIP" – checked 30.04.2013
   http://www.voip-info.org/wiki/view/NAT+and+VOIP
3. "Problems due to widespread use of NAT and IPSEC considerations" – checked 30.04.2013
   http://www.uninet.edu/6fevu/text/IPSEC-NAT.SGML.html
4. "The OSI Model – What It Is; Why It Matters; Why It Doesn't Matter." – checked 30.04.2013
   http://www.tech-faq.com/osi-model.html
5. RFC 1918 "Address Allocation for Private Internets" – checked 30.04.2013
   https://tools.ietf.org/html/rfc1918#section-3
6. "An Overview of TCP/IP Protocols and the Internet" – checked 30.04.2013
   http://www.garykessler.net/library/tcpip.html#IPcidr
7. RFC 3439 "Some Internet Architectural Guidelines and Philosophy" – checked 30.04.2013
   https://tools.ietf.org/html/rfc3439#section-2.1
8. "Peer-to-Peer - World English Dictionary" – checked 30.04.2013
   http://dictionary.reference.com/browse/peer-to-peer
9. "Distributed hash table" – checked 30.04.2013
   http://www.infoanarchy.org/en/Distributed_hash_table
10. "Distributed hash table" – checked 30.04.2013
    http://www.organicdesign.co.nz/Distributed_hash_table#The_Kademlia_DHT
11. "An Overview of Cryptography" – checked 30.04.2013
    http://www.garykessler.net/library/crypto.html#pkc
12. "Digital Signature" – checked 30.04.2013
    http://www.tech-faq.com/digital-signature.html
13. RFC 2104 "HMAC: Keyed-Hashing for Message Authentication" – checked 30.04.2013
    https://tools.ietf.org/html/rfc2104
14. "Encryption - What is it and why is it necessary?" – checked 30.04.2013
    http://www.csee.umbc.edu/~wyvern/ta/encryption.html
15. "What is Diffie-Hellman?" – checked 30.04.2013
    https://www.rsa.com/rsalabs/node.asp?id=2248
16. "42 Monster Problems that Attack as Loads Increase" – checked 30.04.2013
    http://highscalability.com/blog/2013/2/27/42-monster-problems-that-attack-as-loads-increase.html
17. "Linux: The hole trick to bypass firewall restriction" – checked 30.04.2013
    http://www.cyberciti.biz/tips/howto-linux-iptables-bypass-firewall-restriction.html
18. "Wiretapping the Internet" – checked 30.04.2013
    https://www.schneier.com/blog/archives/2010/09/wiretapping_the.html

19. "Proxy Server Risks" – checked 30.04.2013
http://whatismyipaddress.com/proxy-risks
20. RFC 6347 "Datagram Transport Layer Security" – checked 29.04.2013
https://tools.ietf.org/html/rfc6347#section-4.1.1
21. "Java Cryptography Architecture (JCA) - Docs Oracle" – checked 29.04.2013
http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html
22. "Certificate Authority" – checked 30.04.2013
http://www.tech-faq.com/certificate-authority.html
23. "Perfect Forward Secrecy" – checked 30.04.2013
http://www.perfectforwardsecrecy.com/
24. RFC 6544 "TCP Candidates with Interactive Connectivity Establishment (ICE)" – checked 30.04.2013
https://tools.ietf.org/html/rfc6544#section-3
25. "Understanding Today's Skype Outage: Explaining Supernodes" – checked 30.04.2013
http://www.disruptivetelephony.com/2010/12/understanding-todays-skype-outage-explaining-supernodes.html
26. "Who Does Skype Let Spy?" – checked 30.04.2013
https://www.schneier.com/blog/archives/2013/01/who_does_skype.html
27. "Skype Proxy (IP over Skype)" – checked 30.04.2013
https://crypto.stanford.edu/cs294s/projects/skype.html
28. "Brunet: A structured P2P System for NATed Wide-area Environments" – checked 30.04.2013
https://www.acis.ufl.edu/content/brunet-structured-p2p-system-nated-wide-area-environments
29. "Brunet: A structured P2P System for NATed Wide-area Environments" (PDF) – checked 30.04.2013
https://www.acis.ufl.edu/sites/default/files/docs/P12.pdf
30. "Kademlia" – checked 30.04.2013
https://en.wikipedia.org/wiki/Kademlia#Accelerated_lookups
31. "Comparative analysis - TCP - UDP" – checked 30.04.2013
http://www.laynetworks.com/Comparative%20analysis_TCP%20Vs%20UDP.htm
32. "Ensure Message Integrity With Digital Signature" – checked 30.04.2013
http://www.toolsjournal.com/integrations-articles/item/286-ensure-message-integrity-with-digital-signature
33. "Digital Signature Implementation in Java" – checked 29.04.2013
https://www.owasp.org/index.php/Digital_Signature_Implementation_in_Java
34. "What are the Advantages and Disadvantages of Public-Key Cryptography Compared with Secret-Key Cryptography?" – checked 30.04.2013
http://x5.net/faqs/crypto/q4.html
35. "Big endian and Little endian computers" – checked 30.04.2013
http://www.linuxhowtos.org/data/6/byteorder.html

License

Non-exclusive license to reproduce thesis and make thesis public

I, Märt Bakhoff (date of birth: 07.08.1991),

1. herewith grant the University of Tartu a free permit (non-exclusive license) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright, "Secure general purpose P2P overlay network", supervised by Meelis Roos and Margus Niitsoo,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive license does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 13.05.2013

Appendixes

Appendix A: Specification of the packets types

A.1 SignedData

Field	Length	Value	Notes
#1	int16	0x01	container identifier (opcode)
#2	*Data*	payload data	
#3	*ShortData*	NetworkId of sender	
#4	*string*	signature algorithm	prototype uses *SHA1withRSA*
#5	*ShortData*	signature	

SignedData is a container packet that carries the signature of the wrapped payload. Field #3 specifies the network id of the claimed signer, but it is only used to select the public key for signature verification. The concrete signature algorithm is specified in field #4 and contains one of the algorithms defined in the JCA algorithm listing.

A.2 DHRequest (Diffie-Hellman Request)

Field	Length	Value	Notes
#1	int16	0x02	container identifier (opcode)
#2	int32	session id	
#3	*string*	DH algorithm (DH)	
#4	*string*	DH format (X.509)	
#5	*ShortData*	encoded DH key	ASN1 encoded p, g, y

The DHRequest is a terminal packet that is used to initiate the Diffie-Hellman key exchange. Field #2 contains a random number to be used as the session number and to link the session key to a packet in future SessionData packets. Field #3 always contains the string "DH". Field #4 specifies the encoding of the DH data. Field #5 contains the data required for the DH key exchange and if formatted as specified in field #4. The prototype uses ASN1 to encode the DH parameters p, g and y = (g^a) mod p where a is the secret of the initiator. The shared secret found through DH must be hashed using sha1 before use.

A.3 DHResponse (Diffie-Hellman Request)

Field	Length	Value	Notes
#1	int16	0x03	container identifier (opcode)
#2	int32	session id	

#3	*string*	dh algorithm (DH)	
#4	*string*	dh format (X.509)	
#5	*ShortData*	encoded dh key	ASN1 encoded p, g, y

The DHResponse is a terminal packet that is used to reply to the DHRequest packet. Field #2 contains the session number specified by the original request. Field #3 always contains the string "DH". Field #4 specifies the encoding of the DH data. Field #5 is identical to that of DHRequest packet's but y=(g^b) mod p where b is the secret of the replier. The shared secret found through DH must be hashed using sha1 before use.

A.4 PublicKeyRequest

Field	**Length**	**Value**	**Notes**
#1	int16	0x04	container identifier (opcode)
#2	int8	number of keys in request (n)	
#3..#(3+n)	*ShortData*	NetworkId of requested key	

A terminal packet used to request public keys from a peer. The packet contains a set of network ids that the sender is interested in.

A.5 PublicKeyResponse

Field	**Length**	**Value**	**Notes**
#1	int16	0x05	container identifier (opcode)
#2	int8	number of keys in response (n)	
#(3+i*3)	*string*	algorithm of public key	for i in [0;n[
#(4+i*3)	*string*	encoding of the key (X.509/PKCS#8)	
#(5+i*3)	*ShortData*	encoded key	

A terminal packet used to reply to a PublicKeyRequest. The packet contains a set of public keys that matched the request. Each key definition starts with the algorithm name of the key (as specified in JCA algorithm listing), the encoding of the key and the encoded key.

A.6 SessionData

Field	**Length**	**Value**	**Notes**
#1	int16	0x06	container identifier (opcode)
#2	int32	session id	session key is encryption key
#3	*ShortData*	NetworkId of sender	
#4	int8	key length (bytes)	prototype uses 128bit keys
#5	*string*	HMAC algorithm	prototype uses *HmacSHA1*

#6	*ShortData*	HMAC	
#7	*string*	encryption algorithm	prototype uses *AES/CBC/PKCS5Padding*
#8	*ShortData*	encryption IV	generated randomly each time
#9	*Data*	encrypted payload	

The SessionData packet is a container packet that encrypts it's payload using symmetric encryption and uses HMAC to verify it's integrity. Fields #2 and #3 specify the claimed sender id and a session id. The claimed sender id should only be used to find a suitable session key for decryption and HMAC verification. It must not be assumed that the claimed sender id is correct as it can be easily manipulated with a MITM attack.

Fields #5 and #6 specify the HMAC algorithm used to verify the payload. The first #4 bytes of the hashed DH secret are used as the HMAC key. If the HMAC succeeds then the peer related to the used session key must be considered as the sender.

Fields #7 and #8 specify the encryption algorithm and IV. The first #4 bytes of the hashed session key are used as the decryption key. The HMAC must be verified before attempting decryption. After the ciphertext is decrypted the decrypted data must parsed for inner packets.

A.7 Ping

Field	Length	Value	Notes
#1	int16	0x07	container identifier (opcode)

A simple no-op terminal packet that expects a Pong packet as the response. Can be used as UDP keep alive.

A.8 Pong

Field	Length	Value	Notes
#1	int16	0x08	container identifier (opcode)
#2	*ShortData*	IP address of the peer (network order)	4 bytes for IPv4, 16 bytes for IPv6
#3	int32	port number of peer	

A simple terminal packet for responding to the Ping requests. Field #2 should contain the IP address of the peer as seen from the responder regardless of relays.

A.9 DHTQuery

Field	Length	Value	Notes
#1	int16	0x09	container identifier (opcode)

#2	*ShortData*	NetworkId of requested peer	

A terminal packet for querying routes to a peer.

A.10 DHTResponse

Field	Length	Value	Notes
#1	int16	0x0a	container identifier (opcode)
#2	int8	count of response routes (n)	max 30
#3..3+i	*Route*	n routes, each can be either a direct route or a proxy route	i in [0; n[; routes described in A.10.1 Direct route and A.10.2 Proxy route

A terminal packet for responding to DHT queries. Contains a set of routes that were closest to the requested peer. May contain direct routes, proxy routes and routes to other peers with similar network ids.

A.10.1 Direct route

Field	Length	Value	Notes
#1	int8	0x01	route type
#2	*ShortData*	NetworkId of destination	
#3	*ShortData*	IP address of the peer (network order)	4 bytes for IPv4, 16 bytes for IPv6
#4	int32	port number of peer	

A packet component describing a direct route to a host.

A.10.2 Proxy route

Field	Length	Value	Notes
#1	int8	0x02	route type
#2	*ShortData*	NetworkId of destination	
#3	*ShortData*	NetworkId of the proxy	

A packet component describing a proxy route to a host. Proxy routes should only be advertised if the proxy has a established session to the peer being proxied.

A.11 KeyPush

Field	Length	Value	Notes
#1	int16	0x0b	container identifier (opcode)
#2	*string*	algorithm of public key	
#3	*string*	encoding of the key (X.509/PKCS#8)	
#4	*ShortData*	encoded key	

A terminal packet used to establish a connection to a peer. Includes the public key of the sender. Expects a PKResponse packet containing the remote's public key as the response.

A.12 RelayRequest

Field	Length	Value	Notes
#1	int16	0x0c	container identifier (opcode)
#2	*ShortData*	NetworkId of destination	
#3	*Data*	payload to be relayed	

A terminal packet requesting the payload to be relayed to the peer specified in field #2. Expects a RelayResponse packet as the response.

A.13 RelayResponse

Field	Length	Value	Notes
#1	int16	0x0d	container identifier (opcode)
#3	int8	0x00 if the request was denied; 0x01 if the request was accepted	

A terminal packet containing a response to a RelayRequest. Requests should be rejected if no direct routes are available to the destination peer or if no active sessions are found.

A.14 TransmissionTag

Field	Length	Value	Notes
#1	int16	0x0e	container identifier (opcode)
#2	int64	transmission id	0x00 if no payload (just a confirmation)
#3	int64	confirmation id	0x00 if first packet in the session
#4	*Data*	payload of the message	payload with 0 length if no payload

A container packet containing the transmission id and confirmation id. Used to guarantee reliable packet delivery by tracking packets and detecting timeouts. Also used to detect broken routes. Must be present in all transmissions.

A.15 SessionKeyExpired

Field	Length	Value	Notes
#1	int16	0x0f	container identifier (opcode)
#2	int32	session id	

A terminal packet used to signal a broken session key. This might happen if one of the peers is restarted and it's session key cache is cleared in the process (as it should). Must be wrapped in a SignedData packet.

A.16 RelayWrapper

Field	Length	Value	Notes
#1	int16	0x10	container identifier (opcode)
#2	*Data*	data that was sent by proxy	

A container packet wrapping the payload of a relayed packet. The receiver should pass the payload to the packet parsers and pass the sender of the RelayWrapper as the proxy id. The proxy should also be marked as a reverse route to the original sender.