Secure Share

A framework for secure social interaction

Gabor Adam Toth

May 2012

Abstract

The motivation of the work presented here is the need for social interactions over the internet in a scalable and privacy protecting manner. We have examined existing systems from this aspect and have come to the conclusion that they do not provide enough privacy or do not scale well enough for our requirements. We suggest a peer-to-peer (P2P) architecture for this use and present a social network based on the GNUnet P2P framework – which provides the lower layers of the network – extended with messaging and social semantics by the PSYC protocol.

Components of the system we have implemented are a service for the GNUnet framework providing social semantics and messaging functionality, a client API enabling application developers to write applications for the network with less effort, and client applications providing messaging, contact lists and profiles.

The implementation is in a prototype stage, basic functionality, such as person identities, friendship establishment and messaging in channels already work, but further work is needed to enhance the functionality of the system and improve the usability of the user interface.

Contents

List of Figures

1. Introduction

The Internet is getting more and more centralized with users' personal data hosted on servers of large service providers, which involves serious privacy concerns. As in most cases these systems do not provide end-to-end confidentiality, server operators have full access to user data and users are often unaware of how much data is stored about them or with whom their data is shared with. Such systems include email and instant messaging services like GMail and GTalk, social network services like Facebook, Google+ and Twitter, or file storage and sharing services like Dropbox.

It is possible to implement social sharing and messaging in a privacy protecting way. Chapter 2 describes previous attempts at this by federated social networks, problems with that approach and our requirements for secure communication.

We suggest a peer-to-peer architecture as a better basis for a social network system in Chapter 3. We show how social interactions would work in such a network while maintaining privacy of users.

In Chapter 4 we introduce core concepts of PSYC and show how we integrated it with P2P technology provided by the GNUnet framework, and tell more about implementation details of the prototype of Secure Share.

Chapter 5 describes the clients we have implemented and shows extension possibilities of Secure Share.

2. Requirements and Related Work

This chapter describes our requirements for a system that we can use to build a secure social network and introduces currently available alternatives to centralized social networks. This chapter is partly based on [8].

2.1 Privacy

Our goal is to provide a system for social interaction in a privacy-protecting and scalable manner. A truly private communication system we're aiming for should have the following properties:

- End-to-end encryption: only the intended recipients can read the messages, no server or network operators along the way between the communicating parties. To ensure this, it is not enough to use link-level encryption between a client and a server, end-to-end encryption is needed, which means that every participant in the system has to manage their own cryptographic keys on their own systems.
- Perfect forward secrecy: messages transmitted over the network can't be decrypted later if a user's private key is compromised. To achieve this, temporary session keys need to be used when encrypting messages.
- When logging a message to disk it should not contain a cryptographic signature of the sender, so if someone gains access to the log, it does not provide a proof that someone actually transmitted the messages.
- An observer cannot determine for sure when two parties are communicating and how much data they exchange with each other. This requires a trade-off: while sending packets through other participants in the network would ensure this, this also increases message delay.

- Padding of packets is necessary to prevent attacks based on statistical analysis of packet lengths. This is absolutely necessary when sending messages through multiple hops, otherwise it would be enough to monitor packet lengths to determine where a packet is forwarded to.
- Delayed forwarding is also necessary to prevent correlation of received and transmitted packets when forwarding. Sending multiple packets at once at certain intervals would help to prevent this.
- Private contact list: only visible to whom it needs be – typically other friends – not available publicly or managed on servers where server operators have access to it.
- Every component of the system should be open source, so one can ensure it really works as advertised. A closed component would be a security risk, as it could leak information or otherwise weaken the security of the system, which is harder to detect when no source code is available. This can be enforced with a copyleft license, such as the Affero General Public License (AGPL).

Currently available alternatives to centralized social network services are in most cases federated networks, which use a standardized protocol between servers enabling many service providers to take part in the network and communicate with each other. Examples for such systems include web-based platforms like Diaspora or Friendica, and others using a messaging protocol extended with social network functionalities – friendship establishment, status messages to friends – like OneSocialWeb, which is based on XMPP (Extensible Messaging and Presence Protocol) or PSYC (Protocol for SYnchronous Conferencing).

These federated systems intend to offer more privacy than centralized systems, but they still not fulfill most of the requirements above, in most cases they only provide link-level encryption. They still store personal data on servers unencrypted, just like centralized systems. Users can have a server themselves, but that requires server administration skills which average users do not have, so we'll end up with a few larger servers and several smaller ones, just like in the case of email. Privacy is an even more serious issue in this case as it's no longer enough to trust one company, there are several server operators in this architecture sharing personal data with each other – users' messages and profile data are transmitted to and stored unencrypted on servers of their friends as well. Even if some users run their own server, they would still communicate with people without their own server, exposing personal data to even more server operators this way.

It is possible to enhance privacy of these federated protocols by adding end-to-end encryption on top of them, this is what PGP (Pretty Good Privacy) does for e-mail and OTR (Off-The-Record Messaging) does for instant messaging protocols. While this prevents servers from reading the content of messages, they still know everything else about a message, e.g. its sender, recipient, and size. There's an additional overhead of base64 encoding, which is needed because the underlying messaging protocols often do not support binary data transfer. Furthermore PGP and OTR can only be used for one-to-one messaging, one-to-many and many-to-many messaging are not supported by them.

2.2 Scalability

Efficient message distribution is crucial in social networks, as one of their most prevalent features is sending one-to-many status updates, but many-to-many group messaging is frequently used as well. To deliver these messages most efficiently, multicast message distribution would be necessary. IP multicast does not scale to a large number of channels, as multicast routing tables would fill up very fast – at least one channel would be needed for a user's status updates, and similarly, at least one for each group – thus this has to be implemented on the application layer to make it work.

XMPP has a simple distribution strategy, it sends one message per recipient server, which is only efficient if there are many large sites. XMPP's scalability is also limited by the way it handles presence updates, the majority of inter-server traffic in the XMPP network consists of this type of messages.

XMPP's use of an XML stream as network protocol without any framing makes it less efficient, as it complicates parsing and makes it impossible to transport binary data without Base64 or similar encoding. Also, protocol extensions described in XML add a large amount of unnecessary verbosity to the protocol.

PSYC is another federated messaging protocol with a compact but extensible syntax, which enables fast parsing and small bandwidth usage. It is a text-based protocol with length prefixes for binary data. Benchmarks we made show that it outperforms XMPP and JSON when it comes to parsing speed [6].

PSYC sends out one message per recipient server when distributing messages, but it also has manual multicast tree configuration.

2.3 Peer-to-peer networks

Peer-to-peer (P2P) networks come closer to fulfilling these privacy requirements, as in many cases they're designed with security and privacy in mind from the ground up.

Projects such as Tor and I2P aim to create an anonymous overlay network, while Freenet and GNUnet focus on anonymous information storage and retrieval. GNUnet also provides an extensive framework for writing P2P applications, including packet-based communication over different transport mechanisms.

In a P2P network every user of the network runs the P2P software on their own computers (a computer in the P2P network is referred to as a node). This allows for creating a network architecture where servers are not needed to store and manage user data, every user can do so on their own node, giving them more control over their data. High-capacity servers we had in federated networks would be still useful in a P2P network, they can forward (and store when needed) encrypted data without being able to decrypt them, this way improving throughput, connectivity and stability of the network.

Combining peer-to-peer network technology with social network semantics allows for creating a scalable, privacy-protecting social network based on connections of trusted peers. The next section describes the architecture of such a network.

3. Architecture

Secure Share intends to implement a scalable P2P social network enabling real-time one-to-one, one-to-many and many-to-many message distribution for applications using the network while fulfilling the privacy requirements described in the previous chapter.

It provides private and group messaging, status updates and profiles in the first prototype version, while keeping the protocol extensible allowing various social applications to be built on top later.

By combining PSYC with a P2P network architecture we get an efficient and extensible protocol provided by PSYC and security and privacy properties provided by the underlying P2P network.

3.1 P2P network architecture

Many P2P networks use an architecture where nodes connect to arbitrary peers, no trust relation exists between them. A problem with this approach is that some nodes could use more resources of the network than they contribute to it (freeloaders), which can be alleviated by applying an economic model in the network. For instance GNUnet uses an excess-based economy: a node when idle does favors for free, but when busy it only works for nodes it likes and charges them for favors they request, which they can pay back by doing a favor in return.

Another problem that could arise in this architecture are malicious nodes who can perform various active attacks, including blocking access to parts of the network, or returning false information to certain requests. These can be avoided to some extent by randomized routing and by making it harder to create new identities in the network.

A different approach we use is a friend-to-friend (F2F) architecture where

nodes only connect to friendly peers whom they trust. This has the advantage that it avoids many attacks involving malicious nodes in the network. An attacker has to infiltrate a user's social circle to perform a successful attack, which is much harder. By adding a trust level metric to social connections we can further differentiate between more and less trusted nodes in the network.

Also, a F2F architecture gives better incentives to participants in the network: users help their friends by forwarding packets for them instead of random strangers. Nodes with high bandwidth and no connection restrictions – e.g. server machines in data centers – can improve throughput and connectivity in the network by serving their owner's social circle.

Other systems based on a F2F architecture include Freenet [1], Drac [2], Tonika, and GNUnet has a F2F mode as well.

3.2 Structure of the network

Another aspect of P2P networks is whether they're structured or not. In structured networks the structure of the network is predefined, the node ID determines the position of the node in the network, this information is enough to be able to route packets to their destination. Often a distributed hash table (DHT) is used in structured P2P networks which provides hash table functionality distributed over many nodes in the network.

A different approach is an unstructured network like the Internet, where arbitrary nodes can connect, no structure is imposed upon the nodes. In this case a routing table is needed to be able to route a packet to its destination.

A social network could be built purely using a DHT, LifeSocial [4] is an example of such a network. In this case every shared status message, image or document would become an entry in the DHT, and a profile consists of a collection of links to other DHT entries. To ensure only the intended recipients have access to private data, DHT entries are encrypted with a symmetric key, which is attached to the entry encrypted with every user's public key who should have access to the entry. This means that there's no forward secrecy in this network, if a user's private key is compromised all these entries can still be decrypted with that key. Even if noticed in time, re-encrypting all entries affected by a compromised key is quite a costly operation when the number of entries become larger after using the system over the years.

For our case either an unstructured network is suitable, or a structured network where the structure is only used for routing, and not for storing user data in a DHT. In our architecture data is pushed once to recipients who store it locally as long as they need it, which means all profile data, messages and received files are all available locally – even offline – and can be viewed and searched using local tools on the personal device.

3.3 Software components

In a P2P network every user runs the P2P software on their devices, so it's important that it is multi-platform, lightweight, and written in a compiled language, so we can easily run it on all popular desktop platforms and small devices as well, including plug computers, home routers, and even smartphones.

In our case the P2P software runs as a daemon – a background process – on the local machine or on another device on the network. Client applications connect to this daemon and integrate into the desktop or mobile GUI environment running on the system.

Server machines, home routers and plug computers act as intermediary nodes in the system, helping their owners' social network by forwarding packets for them.

Mobile phones require a different approach. Continuous network usage would drain the battery quite fast, so we'll have to minimize it by disabling packet forwarding for mobile nodes and connecting only to a trusted node with good connectivity – e.g. a server machine or a plug computer at home – which would forward the necessary packets for the mobile node.

3.4 Peer-to-peer framework

We have examined various P2P systems looking for an implementation that can serve as a basis for our social messaging platform. The criteria for a suitable P2P framework was:

- Free/libre/open-source software.
- Multi-platform, lightweight and written in a compiled language.

- Implements and provides an API for essential P2P features such as bootstrapping, addressing, routing, encryption and NAT traversal.

We have found GNUnet to be the most promising implementation out there satisfying these requirements. It is a modular P2P framework written in C, providing an API for essential P2P functionalities. It supports advanced NAT (Network Address Translation) traversal, which enables contacting nodes without a public IP address typically found in home or corporate networks. Furthermore it has several transport mechanisms with automatic transport selection, including TCP, UDP, HTTP(S), SMTP and ad-hoc WiFi mesh networks. It also provides various routing schemes and a distributed hash table.

It has three operation modes: in P2P mode it makes connections with any peer in the network, in friend-to-friend (F2F) mode only trusted nodes are connected, and in mixed mode a minimum number of trusted nodes are required to be connected at all times.

GNUnet currently has two options for routing packets in the network: the distance vector and the mesh service.

The distance vector (DV) service uses a fish-eye bounded distance vector protocol [3], which builds a routing table by gossiping about neighboring peers within a limited number of hops distance. It is a link-state routing protocol with improved efficiency: nodes only know about the state of a local neighborhood, and link state of nodes close to each other are updated more often than of nodes multiple hops away. The DV service also provides onion routing of packets through multiple hops, which improves network connectivity by connecting two peers behind NAT through an intermediary hop, and makes it harder for an observer to determine who is talking to whom.

The mesh service creates tunnels through several hops and supports multicast as well. Initial routes to recipients are discovered using the DHT. It is still being heavily worked on by the GNUnet team, for instance encryption is missing and has to be implemented for the multicast groups in order to make it useful for our purpose.

These routing methods only support delivery of packets to connected nodes, in order to provide offline messaging, we'll need a store-and-forward mechanism in the network. This can be implemented by storing encrypted packets on more stable nodes in the network, until the recipient comes back online.

GNUnet also has an anonymous file sharing component which uses a DHT together with the GNUnet Anonymity Protocol (GAP). For our use case – transferring files between friends – this is not needed, instead we transfer files just like other messages, using PSYC's multicast distribution channels. As the PSYC packet syntax supports binary data without any encoding, this causes no additional overhead. In order to transfer files, we would have to split them up into smaller fragments, as the maximum packet size supported by GNUnet is 64KB.

3.5 Messaging daemon

GNUnet's modular architecture allows us to extend it with a service that implements a messaging protocol, manages the connections between people, and provides a local client interface. This service – called psycd – uses the PSYC protocol for communication with both other peers and local clients.

Psycd sends messages through GNUnet core, which encrypts the message and passes it to the modular transport system, sending packets through one of its transport plugins.

In our prototype we use direct connections to peers. Users manually add their friends by exchanging hello messages, which contain their public key and current addresses. For the prototype version the focus was on the implementation of the messaging daemon, and we intend to work on the underlying routing mechanism in future versions.

See figure 3.1 for an illustration of the components used in the system. Dotted parts are not existing yet, only planned. The arrows depict the flow of messages between components.

3.6 Functionality

One of the core concepts of PSYC is programmable channels with their own subscription lists. Using this combined with custom user interfaces makes it possible to implement the usual functionality found in centralized and federated social networks, like private and group messages, status updates, photo and link sharing, as well as features not found in those networks, like sharing of files and custom content, or real-time notifications for custom events.

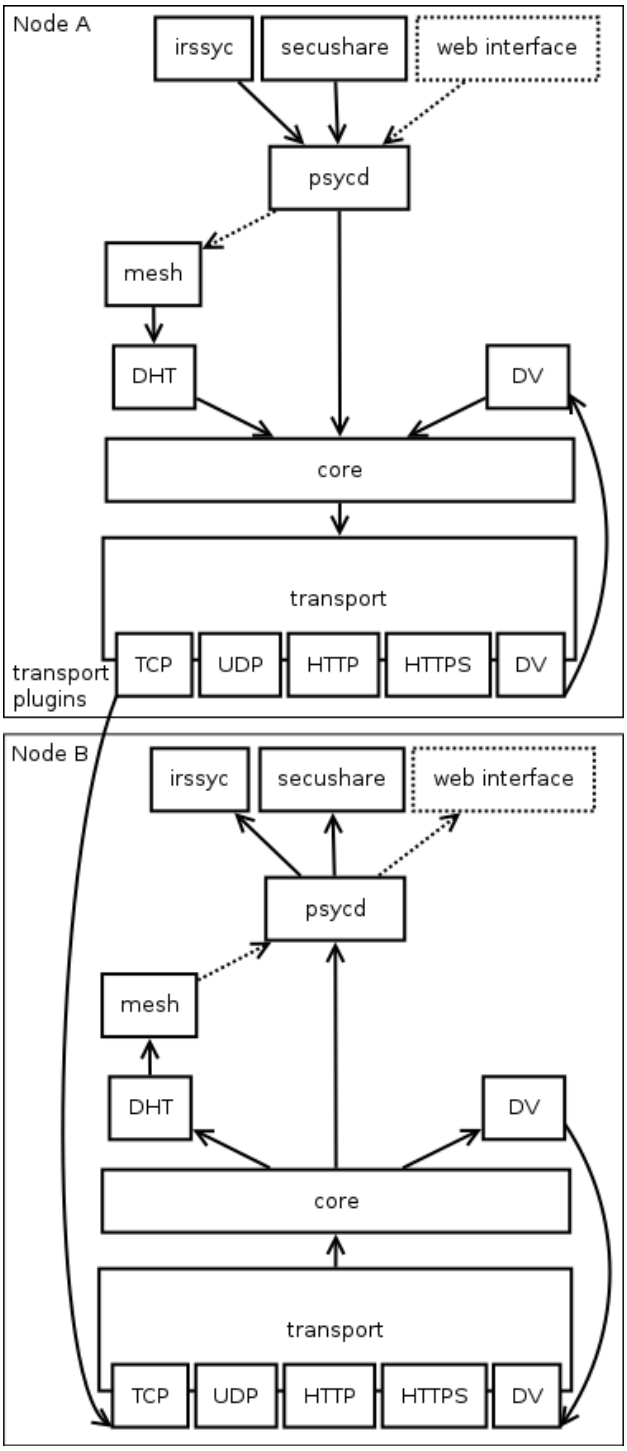

Figure 3.1: Components and message flow in GNUnet

As Secure Share runs on the users' own device and stores all incoming messages and data locally, this enables offline usage and local search in the data received from subscribed friends or groups.

4. Implementation

This chapter describes core concepts in PSYC, how they are applied in a peer-to-peer context and what changes we had to make to the federated PSYC [7] protocol to make it work in a peer-to-peer network.

Federated PSYC is the existing implementation of the PSYC protocol designed for a federated architecture. It is implemented as a stand-alone daemon process written in the LPC language.

P2P PSYC is the new implementation we have developed and the one we use in Secure Share. The messaging daemon – called psycd – is implemented in C as a service in the GNUnet framework. It uses GNUnet libraries for communication with the rest of GNUnet, and libpsyc for the parsing and rendering of PSYC packets. It stores data in an SQLite database.

4.1 Syntax

PSYC is a text-based protocol with length prefixes for binary data, which makes it possible to transmit any kind of content in PSYC packets efficiently while keeping the protocol extensible. Its syntax is described in Appendix 1.

An example packet looks like this:

```
:_context        psyc://J61VSCQA:g/#test
:_source_relay   psyc://I0GCD93U:g/
70
=_simple_var     value
:_binary_var 5   value
_method_name
Packet
body
here.
|
```

A packet contains a routing header, followed by the length of the rest of the packet, context state modifiers, the method name and the packet body.

4.2 Identifiers

In federated PSYC a server is identified by its DNS domain name. A server hosts person and group entities, each of which can manage several channels. Uniforms serve as identifiers for entities or channels, described with a URI (Uniform Resource Identifier) syntax:

```
psyc://host[:port[transport]][/[entity-type]entity[#channel]]
psyc://example.net/~alice#friends
```

In peer-to-peer PSYC DNS is not employed, a public key is used instead to identify node, person or group. GNUnet uses a SHA-512 hash of the public key as node identifiers, we use a similar method for identifying entities. The ASCII-encoded version of this hash becomes the host part of the uniform, with no port number and 'g' as transport identifier:

```
psyc://pubkey-hash:g[/[entity-type]entity[#channel]]
psyc://I0GC...L29G:g/#friends
```

As these identifiers are very long and not user-friendly, they can be aliased to shorter nicknames. The aliases are only used in client applications, they do not appear on the protocol level.

In the prototype version GNUnet's host keys are used for identifying person entities as well, this simplification allows only one person per node. A more elaborate identification scheme is to be implemented later.

Each user will have a master key which serves as the identifier of the person, its purpose is to sign subkeys used by various devices of the person. If a subkey gets compromised, the master key can be used to prune messages sent with the compromised key.

These subkeys are assigned to person entities. A GNUnet node can host one or more entities. When using the distance vector transport, node and entity IDs are added to the DV routing table, and nodes gossip about available peers and entities in a local neighborhood up to a limited number of hops away, in the social circle of users. When using the mesh service, user ID to current node ID mappings are stored in the DHT.

4.3 Circuits

A circuit is a virtual connection between two PSYC nodes, packets are sent and received over circuits. When sending packets the circuit type is determined by the transport specified in the target uniform.

In federated PSYC we had TCP, UDP and TLS transports. In P2P PSYC psycd implements two circuit types so far: TCP circuits for local clients and GNUnet circuits for remote peers. Unix sockets, TLS and possibly UDP circuits are planned for later.

4.4 Contacting peers

In federated PSYC it was enough to know the uniform of a person or group to establish contact. The uniform contains the host name, port number and transport method, which is all the information needed to establish connection to the remote entity.

When using PSYC over P2P, two nodes have to know each other's public key and know how to reach the node associated with the public key. GNUnet introduces nodes to each other using hello messages which contain a public key and various transport methods and addresses which can be used to establish contact with the node. In case of the DV transport a hello message contains the identifier of another node through which it can be reached. The DV routing protocol gossips about connected nodes and entities in the network so they become reachable by their social network.

When two users want to talk to each other, they should have received a hello message from the other party beforehand. When using the DV transport they might already know about each other if they are connected through common friends and received a gossip message about the other node. If they are on the same network they would discover each other through IPv4 broadcast or IPv6 multicast, or when using the WLAN transport a WiFi mesh network is created from the present nodes. Otherwise a hello message can be exchanged manually between users, using e.g. email or a USB stick. When sending a hello message over an insecure channel it should be encrypted using a shared secret in order to maintain confidentiality and integrity of the information contained within. Usually it's enough to exchange hello messages manually once when establishing connection for the first time, after that more stable, longer running nodes would be available to bootstrap a reconnecting node.

When connection is established between two users, they set appropriate trust levels for each other – which can be used in routing decisions in the network – and they subscribe one or more channels of the other party.

4.5 Entities

Entities are addressable objects in the PSYC network. Entity types include place entities which are used for group communication or news feeds, and person entities which can make friendships between each other and subscribe to other entities. Each entity manages one or more channels with different subscription lists.

Psycd implements person entities enabling clients to link to their entity, send and receive messages and manage membership of various channels. It also has a simple implementation of place entities providing dedicated group messaging.

4.6 Multicast contexts

PSYC uses multicast contexts for efficient distribution of messages. A context is managed by the context master at the top of the distribution tree. Context members send packets to the context master which distributes them to context slaves on the next level in the multicast tree, which distribute them further down the tree. Figure 4.1 shows such a tree.

Entities manage multiple channels, each of which is a separate multicast context having different membership and multicast distribution tree. Social interactions, such as status updates, group and private messaging can be modeled using these channels. An entity manages membership of its channels, in case of a person entity this could be used to create different circles of friends using a channel for each of them, or provide different channels for various topics to which interested friends – or if desired anyone who can contact the person – can subscribe to. Ad-hoc group and private chats with friends can be modeled as well with channels of a person entity.

Federated PSYC only implemented manually configured multicast distribution trees so far, this should be made fully automatic in the peer-to-peer version. When multicast routing is added, every node becomes a multicast routing hop serving several multicast contexts. A node can join a multicast

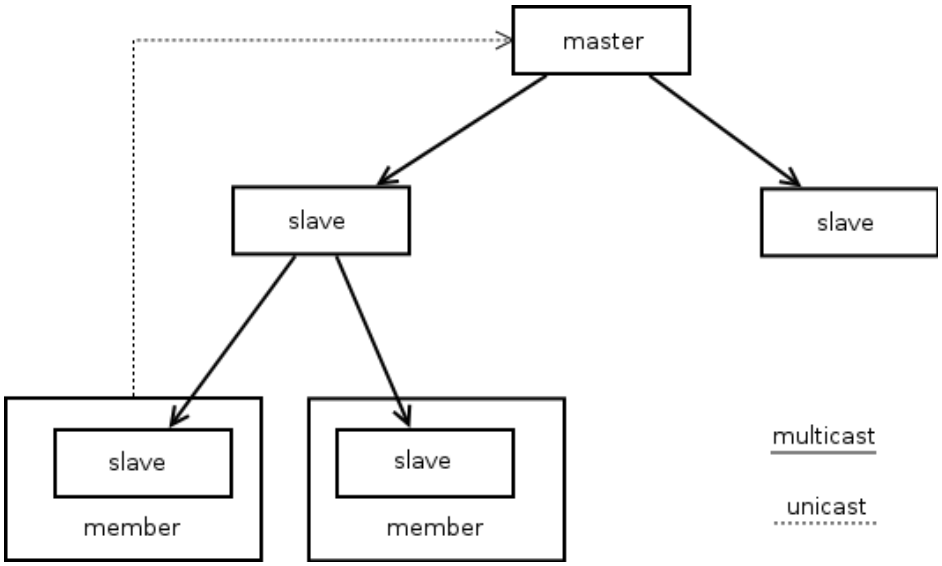

Figure 4.1: Multicast context distribution tree

context at any other node already a member of that particular context. By adding encryption to multicast contexts any node can help in the multicast routing process without being able to decrypt message contents. This way receiving packets for a multicast context does not necessarily mean that the given node can decrypt the packets sent to it. In its simplest implementation multicast encryption involves a symmetric key distributed by the context master to all the members which has to be changed periodically, and when a member joins or leaves.

In [5] Hordes, an anonymity protocol based on IP multicast is suggested. While we're not using IP multicast, part of their analysis could be applied to application-level multicast implemented in a P2P network.

The prototype does not implement actual multicast yet, multicast contexts are modeled but messages to contexts are distributed to each member by unicast.

4.7 Distributed state

PSYC has the concept of distributed state, a set of key-value pairs – state variables – are assigned to each multicast context and distributed to every member. It is used to model profile data, context membership, or any other

data related to a context. Advantage of this approach is that it avoids unnecessary request-response operations as members have an up-to-date version of the state data most of the time, and allows local browsing of profiles of contacts, even offline. We have implemented distributed state for P2P PSYC in psycd – a feature federated PSYC has long planned for but still lacked.

Context state is kept in sync using state modifiers provided by the PSYC syntax. A state modifier adds, removes or modifies a state variable. State changes are distributed to context members only once, which means it is very bandwidth efficient. Using state modifiers require reliable, in-order delivery of packets. Packet loss can be detected with the help of a `_counter` variable in the routing header of packets. As the name suggests, it is a counter incremented by one for every packet sent to the context. When there's a missed packet, a node can re-request it from its parent node in the multicast distribution tree. After a node has joined a context, a full state synchronization is necessary to bring the node up-to-date.

Syntax of a state modifier in Augmented Backus-Naur Form (ABNF):

```
entity-modifier = operator variable entity-arg
entity-arg      = simple-arg / binary-arg / LF

operator        = "=" / ":" / "+" / "-" / "?" / "!" / "@"
variable        = 1*kwchar
simple-arg      = HTAB text-data LF
binary-arg      = SP length HTAB binary-data LF
length          = 1*DIGIT
binary-data     = *OCTET
```

Operators:

- `:` (set) – set variable just for the current packet, state is not modified
- `=` (assign) – assign value to state variable
- `+` (augment) – concatenate string or add list/dictionary element, depending on type
- `-` (diminish) – remove list or dictionary element
- `@` (update) – update an item in a list or dictionary
- `?` alone on a line: request state synchronization, all state variables are returned in the response
- `=` alone on a line: reset state, i.e. remove all previously stored state variables

- the rest of the operators are reserved for future use

4.7.1 Syntax changes

The state implementation involved some syntax changes: we have added a dictionary type in order to be able to store key-value pairs in a state variable, and modified the list syntax to make it consistent with the new dictionary syntax, allowing us to specify types for list elements as well. We have also added a new update modifier, which allows for updating individual list and dictionary elements.

These syntax changes were necessary to represent more complex data structures, such as context members or alias mappings.

4.7.2 List syntax

A list is a list of ordered elements. Its syntax in ABNF is specified as the following:

```
list       = [ default-type ] *list-elem
list-sep   = "|"
list-elem  = list-sep [ "=" type ] [ SP list-value ]
list-elem  =/ list-sep "=" type ":" ] [ length ] [ SP *OCTET ]
list-value = %x00-7B / %x7D-FF  ; any byte except "|"
```

Examples:

```
=_list_one      _type| elem1| elem2| elem3
=_list_two      |=_type1 elem1|=_type2 elem2|=_type3 elem3
```

Inserting list elements

For inserting values before a specified index the + operator is used. Index of the first element is 1, index of the last is -1. 0 means the end of the list, which is the default if the index is omitted.

Syntax of the value part:

```
list-insert     = [ list-index SP ] list
list-index      = "#" 1*DIGIT
```

Example:

```
+_list_fruits   | banana| mango
+_list_fruits   #0 | banana| mango
```

Removing list elements

For removing elements the - operator is used. Parameters are the start index which defaults to -1, and the amount of elements to be removed which defaults to 1.

Syntax of the value part:

```
list-remove     = ( list-index SP uint | list-index | uint )
```

Example:

```
-_list_fruits   #1
-_list_fruits   #1 1
```

4.7.3 Dictionary syntax

A dictionary is a set of key-value pairs. Its syntax specified in ABNF is:

```
dict            = [ type ] *dict-item
dict-item       = dict-item-key dict-item-value
dict-item-key   = "{" ( dict-key / length SP *OCTET) "}"
dict-item-value = type [ SP dict-value ]
dict-item-value =/ [ length ] [ ":" type ] [ SP *OCTET ]
dict-key        = %x00-7C / %x7E-FF     ; any byte except "{"
dict-value      = %x00-7A / %x7C-FF     ; any byte except "}"
```

`type` is the default type for elements which do not have a type specified.

Examples:

```
=_dict_one      _type{4 key1}6 value1{key2} value2{key3}6 value3
=_dict_two      {4 key1}=_type1:6 val1{key2}=_type2 val2{key3}6 val3

=_dict_avatars  _picture{alice}3 \o/{bob}7 \oXoXo/
```

The `struct` type can be used to define dictionary values with less repetition. The structure is first defined once, then used for one or all elements. It works like a C struct, a list of types are defined in a specific order, after that we don't have to specify the types again when specifying the values.

```
=_struct_member |=_nick|=_picture
=_dict_members  _struct_member{13 psyc://alice/}12 | alice| \o/
=_dict_members  {psyc://alice/}=_struct_member | alice| \o/
```

Adding dictionary entries

The + operator is used for adding entries to an existing dictionary. The syntax is equivalent to the initial assignment of entries. If a key already exists in the dictionary, its value is overwritten.

Removing entries from a dictionary

The - operator is used for removing entries, syntax is the same as assignment but only the keys are listed.

Example, removing 2 entries:

```
-_dict_members  {psyc://alice/}{psyc://bob/}
```

4.7.4 Update syntax

For updating specific entries in a list or dictionary the `@` operator is used. It has the following syntax:

```
update          = 1*index SP op [ type ] [ ":" length] [SP value]
index           = ( dict-item-key / index-list / index-struct )
index-list      = "#" 1*DIGIT
index-struct    = "." type
```

Examples:

```
@_list_gallery  #-1 =_picture:7 \oXoXo/
@_list_gallery  #-1 =:7 \oXoXo/
@_list_fruits   #1 = pear
@_list_prices   #2 =_int 1000

@_dict_gallery  {alice} =_picture:7 \oXoXo/
@_dict_gallery  {alice} =:7 \oXoXo/
@_dict_members  {psyc://alice/}._nick = Alice
@_dict_members  {psyc://bob/}._nick + Bob
@_dict_members  {psyc://foo/}._int_score + 2
```

4.8 Storage

Incoming and outgoing packets, state variables and channel configuration are stored in an SQLite database. This allows for persistent storage of context state as well, which is restored after a restart of the node. Packets are stored for two purposes: it provides a message history for contexts and it can be used later to resend lost packets to nodes requesting it.

SQLite is used mainly because of its efficient memory handling and wide platform support.

The database consists of two tables with the following schema:

- **contexts** (**uni** blob primary key, **state** blob, **config** blob, **created** timestamp default current_timestamp)
- **packets** (**context** blob, **source** blob, **target** blob, **counter** unsigned int, **fragment** unsigned int, **packet** blob, **created** timestamp default current_timestamp,
  primary key (context, source, target, counter, fragment))

We store information about subscribed and hosted contexts in these tables. The contexts table is used for storing configuration and state of contexts, whereas the packets table is for storing packet history. All this information is stored in PSYC packet format in the database.

5. Clients

Clients implement a user interface for interacting with the PSYC network. They connect to the PSYC daemon and link with a person entity. After successful linking they receive all the packets destined for that person and can send packets originating from the person. In the current implementation of psycd no authentication is required for linking, so it's only suitable for localhost use, later we'll provide password authentication as well.

We have developed a client library – called libpsycclient – providing a simple API for clients. It implements the core logic used by clients to interact with the PSYC network. It allows clients to establish a connection to psycd, so they can send and receive packets for their person entity. Clients can define callback functions for handling incoming packets and various events, e.g. handling linking and unlinking or adding and removing aliases. The library also provides various commands used in clients, such as entering and leaving contexts, sending messages, setting aliases, or querying and manipulating the context state.

By using the client library, implementing new clients is much simpler. With the library providing all the underlying logic, client developers can focus on the GUI, implementing message display and UI elements performing various commands provided by the library.

5.1 Desktop clients

We have implemented two clients so far: a text-based client and one with a graphical user interface (GUI).

The text-based client, irssyc (figure 5.1), is implemented in C as a module for Irssi, a popular chat client. It is more suited for advanced users and for development and testing purposes. It shows each subscribed channel in one

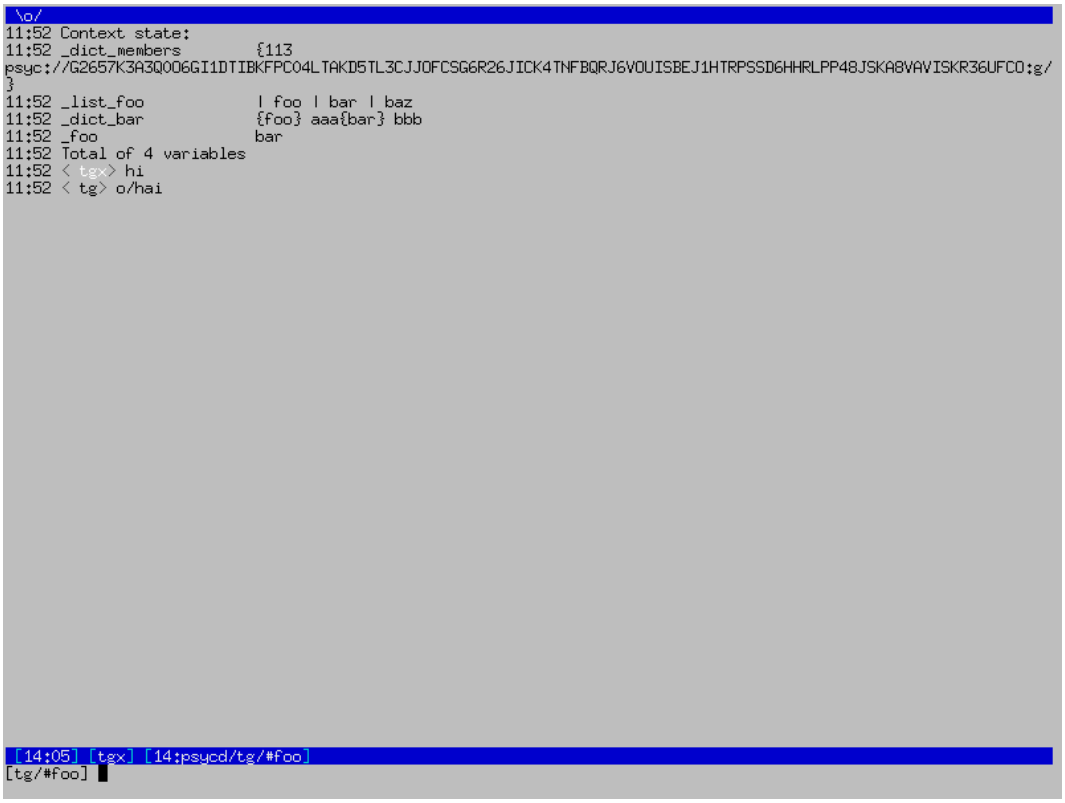

Figure 5.1: irssyc, a text-based client

of its windows and provides access to commands implemented by the client library.

The GUI client, secushare (figure 5.2), is implemented using Qt in C++. It uses Qt's relatively new declarative user interface (UI) description language, QML. The C++ part of the application implements data models used by QML components to display data – such as the contact list or messages in a channel – and provides access to the commands implemented by the client library from QML.

The reasons for choosing Qt were its extensive platform support and its declarative UI description language, QML, which makes it easier to accomplish a complex but still consistent user interface with good usability.

Qt supports most desktop operating systems – including Windows, Linux and Mac OS X – and a couple of mobile platforms as well: Maemo, MeeGo, Windows Mobile and Symbian. Recently it has been ported to Android as well, and there's an iOS port being developed, too.

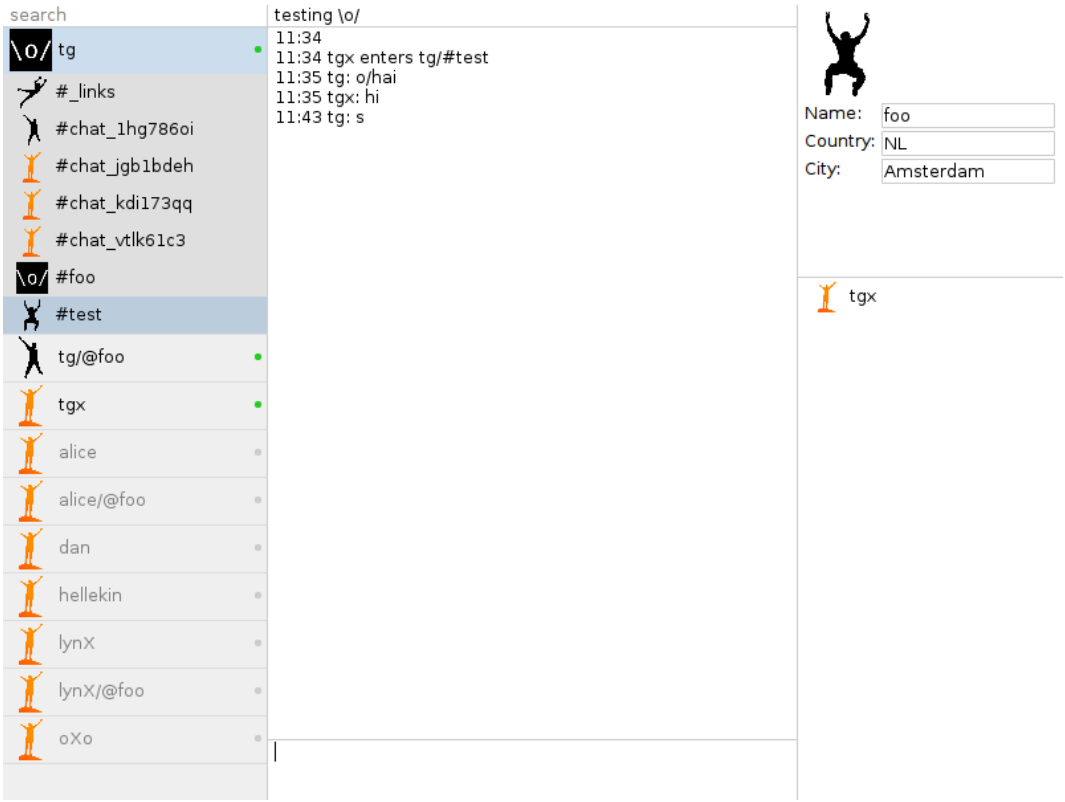

Figure 5.2: secushare, a GUI client

5.2 Web interface

We have plans for developing a web interface as well, which allows remote access of a node installed on a plug computer or server machine. This is useful in case the user does not have a device available that runs a full node with the whole software stack. The web interface will be a PSYC client written in JavaScript, communicating with psycd via WebSocket. This way we only need minimal enhancements on the server side, as the client is pretty much like a desktop client in this case, only the connection to psycd is implemented differently. Now that JavaScript typed arrays are available in most modern browsers, parsing of binary packets are possible now purely in JavaScript.

5.3 Mobile clients

As all components of GNUnet are written in C, it is possible to port it to smartphone platforms. Problem with this approach, however, is that continuous network traffic drains the battery really soon, so we'll have to take measures to reduce network traffic. If the mobile node connects only to one trusted node – e.g. hosted on a server or plug computer in the user's home – which forwards the necessary packets for the mobile node, this significantly reduces network traffic, as the mobile node does not have to take part in any routing scheme, which usually means continuous traffic, even if it's low volume.

Another approach is to only implement a client application for mobile devices which connects to a remote psycd on a trusted node over a TLS connection. This, however, requires users to set up a server or a plug computer at home and configure their firewall or NAT box to allow connections to the PSYC daemon. Advantage of the full node approach is that GNUnet already takes care of NAT traversal, it does not need to deal with (dynamic) DNS and TLS certificates.

5.4 Extensibility

Extensibility via custom applications is an important aspect of the system. We have two different approaches to achieve this.

5.4.1 Channel API

Channels can have an interface type defined in an `_interface` state variable. The default view is a chat interface, and we're planning to provide a few other built-in types in the secushare GUI client, e.g. a microblogging interface with status updates.

We intend to enable developers to write custom applications on top of channels, which will run in a sandboxed QML or HTML view inside the client, using a JavaScript API for sending and receiving packets for the channel. This approach does not expose any private user data to the applications, as they only have access to the channel they're running in, and nothing else.

5.4.2 Client API

For more complex tasks custom client applications have to be built using the libpsycclient C library. This approach allows full access to user data and messages for the application, thus users should be careful what client applications they install on their machine.

6. Conclusion and Future Work

The implementation of Secure Share contributes to the efforts of creating a privacy protecting peer-to-peer social network. The client API provided as a library allows for creating various client applications right away, while the lower layers of the system are worked out.

By implementing psycd as a service for the GNUnet framework, it allows us to benefit from GNUnet's modular architecture, which can be extended with new or improved routing schemes in future versions. The DV and mesh service are areas which need improvement. We need to have proper multicast message distribution in the network, and the mesh service is a promising candidate for that. It implements multicast groups, but group encryption still has to be implemented for this service. Thus improvements on the routing level are necessary to make the system really scalable and privacy protecting.

As the implementation is still in a prototype stage, further work is needed to enhance the functionality of the system. Areas that need more work are:

- improve the functionality and usability of the user interface, e.g. add dialogs for friendship establishment, and add different interfaces for different types of channels, e.g. status updates.
- implement dedicated groups independent of person entities
- user identities should be decoupled from node identities by assigning separate keys to users, so they have a master key and subkeys for their devices
- file transfer over PSYC, this requires splitting large packets into smaller fragments and reassembling them when receiving
- add UNIX socket support to psycd, as currently only TCP sockets are supported for clients

- TLS sockets could be added later as well to enable secure connection to a remote node in case a local installation is not available
- make the system work on mobile devices – this could be done either via setting up a GNUnet node on the device or via establishing a TLS connection to a remote node; this also requires developing a user interface specifically designed for mobile devices
- implement a web interface
- implement testing using the GNUnet testing library

Bibliography

[1] Ian Clarke et al. *Private Communication Through a Network of Trusted Connections: The Dark Freenet*. URL: https://freenetproject.org/papers/freenet-0.7.5-paper.pdf.

[2] George Danezis et al. "Drac: An architecture for anonymous low-volume communications". In: *Privacy Enhancing Technologies, volume 6205 of Lecture Notes in Computer Science*. Springer, 2010, pp. 202–219.

[3] Nathan S. Evans. "Methods for Secure Decentralized Routing in Open Networks". PhD thesis. Garching bei München: Technische Universität München, 2011, p. 234. ISBN: 3-937201-26-2. URL: https://gnunet.org/nate2011thesis.

[4] K. Graffi et al. "LifeSocial.KOM: A secure and P2P-based solution for online social networks". In: *Consumer Communications and Networking Conference (CCNC), 2011 IEEE*. IEEE. 2011, pp. 554–558.

[5] Brian Neil Levine and Clay Shields. "Hordes –- A Multicast Based Protocol for Anonymity". In: *Journal of Computer Security* 10.3 (2002), 213–240. ISSN: 0926-227X. URL: http://portal.acm.org/citation.cfm?id=603406.

[6] *Libpsyc Performance Benchmarks*. URL: http://lib.psyc.eu/bench.

[7] Carlo v. Loesch. *Protocol for Synchronous Conferencing*. 2007. URL: http://www.psyc.eu/whitepaper/white.en.html.

[8] Carlo v. Loesch, Gabor Adam Toth, and Mathias Baumann. "Scalability & Paranoia in a Decentralized Social Network". In: *Federated Social Web conference*. Berlin, Germany, 2011. URL: http://secushare.org/2011-FSW-Scalability-Paranoia.

Abbreviations

ABNF	Augmented Backus-Naur Form
DHT	Distributed Hash Table
DNS	Domain Name System
DV	Distance Vector (routing protocol)
F2F	Friend-to-Friend (network architecture)
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
OTR	Off-the-Record Messaging
P2P	Peer-to-Peer (network architecture)
PGP	Pretty Good Privacy
PSYC	Protocol for SYnchronous Conferencing
QML	Qt Modeling Language
SHA	Secure Hash Algorithm
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
URI	Uniform Resource Identifier
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

Appendix 1 - PSYC Syntax

Syntaxes in this section are described in Augmented Backus-Naur Form (ABNF).

PACKET SYNTAX

```
packet  = routing-header [ content-length content ] "|" LF
; the length of content is either implicit
; (scan until LF "|" LF)
; or explicitly reported in content-length.

routing-header  = *routing-modifier
entity-header   = *sync-operation *entity-modifier
content         = entity-header [ body LF ]
content-length  = [ length ] LF

routing-modifier= operator variable ( simple-arg / LF )
sync-operation  = ( "=" LF / "?" LF )
entity-modifier = operator variable entity-arg
entity-arg      = simple-arg / binary-arg / LF

body            = method [ LF data ]

operator        = "=" / ":" / "+" / "-" / "?" / "!" / "@"
simple-arg      = HTAB text-data LF
binary-arg      = SP length HTAB binary-data LF

length          = 1*DIGIT
binary-data     = *OCTET
                ; a length byte long byte sequence

method          = 1*kwchar
variable        = 1*kwchar
text-data       = *nonlchar

data            = *OCTET
                ; amount of bytes as given by length or until
                ; an (LF "|" LF) sequence has been encountered
```

```
nonlchar        = %x00-09 / %x0B-FF
                ; any byte except \n
kwchar          = %x30-39 / %x41-5A / %x61-7A / "_"
                ; alphanumeric or _
```