Design of a Social Messaging System Using Stateful Multicast

Gabor X Toth

University of Amsterdam
Faculty of Science
System and Network Engineering
Master thesis

Supervisor: Christian Grothoff, TU München

August 2013

Abstract

This work presents the design of a social messaging service for the GNUnet peer-to-peer framework that offers scalability, extensibility, and end-to-end encrypted communication. The scalability property is achieved through multicast message delivery, while extensibility is made possible by using PSYC (Protocol for SYnchronous Communication), which provides an extensible RPC (Remote Procedure Call) syntax that can evolve over time without having to upgrade the software on all nodes in the network. Another key feature provided by the PSYC layer are stateful multicast channels, which are used to store e.g. user profiles. End-to-end encrypted communication is provided by the mesh service of GNUnet, upon which the multicast channels are built. Pseudonymous users and social places in the system have cryptographical identities — identified by their public key — these are mapped to human memorable names using GNS (GNU Name System), where each pseudonym has a zone pointing to its places.

Acknowledgements

I would like to thank Christian Grothoff for the advice and discussions related to the design of the system presented here. I also thank Carlo v. Loesch for comments about the use of PSYC in the system.

Contents

1 Introduction **1**
1.1 Public Key Infrastructure 2
1.2 Identity management with GNS 3

2 Overview of the system **5**

3 Multicast service **9**
3.1 Multicast API overview 10
3.1.1 Origin 10
3.1.2 Member 10
3.1.3 Group 11
3.2 Starting the origin 11
3.3 Stopping the origin 12
3.4 Joining a group 12
3.4.1 Requesting join 12
3.4.2 Receiving the join request 14
3.4.3 Responding with a join decision 14
3.5 Parting a group 15
3.6 Multicast messages 15
3.7 Testing membership 17
3.8 Replaying multicast messages 18
3.9 Unicast requests 21

4 PSYC service **23**
4.1 PSYC API overview 24
4.1.1 Channel master 25
4.1.2 Channel slave 25
4.1.3 Channel 25
4.2 Starting the channel master 26
4.3 Stopping the channel master 27
4.4 Joining a channel 27
4.4.1 Requesting join 27
4.4.2 Responding to a join request 28

4.4.3 Notifying the channel about the new slave 29
4.4.4 Informing the requester about the decision 30
4.4.5 State synchronization 30
4.5 Parting a channel 31
4.6 Messages from the master 32
4.6.1 Sending messages 32
4.6.2 Receiving messages 33
4.7 Requests from slaves 34
4.8 History and state requests by applications 35
4.8.1 Requests for historic messages 35
4.8.2 State variable requests 36

5 PSYCstore service 37
5.1 PSYCstore API overview 37
5.2 Message store 37
5.2.1 Storing message fragments 38
5.2.2 Retrieving messages 39
5.3 Channel state store 40
5.3.1 Applying state modifiers 40
5.3.2 State hash updates 41
5.3.3 Retrieving state variables 42
5.4 Membership store 42
5.4.1 Storing membership 42
5.4.2 Testing membership 44

6 Social service 45
6.1 Social network model 45
6.2 Messaging in places 45
6.3 Applications 47
6.4 Social API overview 48
6.4.1 Home 48
6.4.2 Place 48
6.5 Using the social API 49
6.6 Entering a home 49
6.7 Advertising a home in GNS 50
6.8 Entering a place 51
6.8.1 Guest requests entry 51
6.8.2 Host answers the door 52
6.9 Leaving a place or home 53
6.10 Messages 54
6.10.1 Sending messages 54
6.10.2 Receiving messages 55
6.11 Learn the history of a place 57
6.12 Accessing objects in the place 57

7 Summary **59**

Bibliography **61**

A Sequence diagrams **63**
- A.1 History and replay . . . 64
  - A.1.1 Learning the history of a place . . . 64
  - A.1.2 Replaying multicast messages . . . 64
- A.2 Sending and receiving messages . . . 66
  - A.2.1 Sending a message to a home . . . 66
  - A.2.2 Receiving a message to a place . . . 66
- A.3 Entering a home . . . 68
- A.4 Entering a place . . . 68
  - A.4.1 Guest requests entry . . . 68
  - A.4.2 Host receives entry request . . . 70
  - A.4.3 Other guests receive entry notice . . . 72
- A.5 Leaving a place . . . 74
  - A.5.1 Guest sends leave request . . . 74
  - A.5.2 Host receives leave request . . . 74
  - A.5.3 Other guests receive leave notice . . . 76

1. Introduction

Messaging and social networking services used today are in most cases centralized and operated by large companies, which do not give adequate privacy guarantees to users. Often, federated systems are suggested as an alternative, but as argued in [1], federated systems are not better from centralized ones from a privacy perspective: they still require server operators who have access to all user data stored on their systems. Additionally, in case of federated systems users even have to trust a larger set of server operators with their personal data and communications. With centralized services, trust is at least limited to a single entity.

[1] describes the high-level design of Secure Share, an alternative social messaging system. It proposes an alternative, peer-to-peer approach, where messages are end-to-end encrypted while they traverse the network and reach their destination, and suggests combining GNUnet[1], a modular peer-to-peer framework, with PSYC[2], a messaging protocol, to achieve this.

In this document, we present an evolution of the Secure Share design, providing more details about the protocol and implementation strategy and detailing how these extensions will be integrated with the rest of the GNUnet framework. In this revised design, the system is separated into several GNUnet services for each layer, enabling the re-use of components by other P2P applications. The role of the PSYC protocol has changed as well: it is now only used as the messaging protocol between peers, but not for local client connections. For user interfaces, the service model of GNUnet is followed instead.

Recently, two components have been implemented in GNUnet that play a key role in the new design. One is the mesh service, which can establish tunnels between two peers in the network through multiple hops, and is used as a building block for the multicast message delivery service. The other is the GNU Name System[3] [2], which is used as a public key infrastructure in the

[1]https://gnunet.org/

[2]https://about.psyc.eu/

[3]Formerly called the GNU Alternative Domain System (GADS)

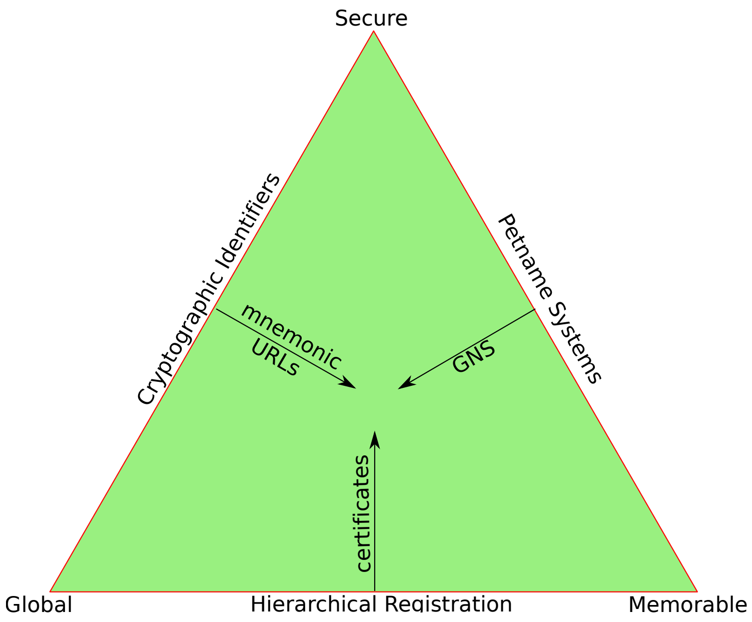

Figure 1.1.1: Zooko's triangle showing the relationship of cryptographically secure, globally unique and human memorable names. Examples of systems satisfying pairs of these properties are shown on the edges. This figure is from [2], p. 7.

system.

1.1 Public Key Infrastructure

The GNU Name System (GNS) is a secure, decentralized name system using memorable, but not globally unique names. According to the hypothesis of Zooko's Triangle [3, 4], achieving all three properties at the same time is not possible, Figure 1.1.1 illustrates this. Tor's .onion addresses are cryptographically secure, but not memorable. The Domain Name System (DNS) is a centralized system, where names are globally unique and memorable, but not secure. GNS offers an alternative to this, while maintaining backwards compatibility with DNS. It uses zones that can have DNS resource records (RR) in addition to RR types defined by GNS.

The GNS design uses cryptographical identifiers: each zone has an elliptic curve (ECC) private-public key pair, and it is identified by its public key. Resource records are signed with the private key of the zone. This allows users to manage their own zone in the system, which is used to implement a decentralized public key infrastructure (PKI). While GNS does not have globally unique names, it provides transitivity instead to make the system more usable: users can delegate a subdomain to another zone.

1.2 Identity management with GNS

Each entity in the proposed system will be identified by its elliptic curve public-private key pair, which is typically compactly represented using the hash of its public key.

GNS is used to provide memorable names for both pseudonymous users and social spaces for interactions, which we call places. As pseudonyms are identified by their ECC key, each pseudonym has implicitly also a GNS zone associated with it. A zone can contain pointers to various places operated or used by the owner of the pseudonym. For this purpose, we define a new record type called “PLACE” in GNS, which points to the public key of a place, and one or more peers that can be used to join the place. A GNS zone is published in the DHT, and its records are publicly accessible. A password can be used to encrypt certain records — including its label, type, and data — to ensure that knowledge about places is limited to authorized users.

The “PLACE” record of the empty label (+) has special significance: it points to a place which has no guests and is only used for sending requests to the pseudonym. It is used to establish an initial contact; after an entry request to this special place, the guest is typically be redirected to a place of the host’s choice. This way, a host can have several places for hosting different group of guests at the same time.

Table 1.2.1 shows an example of the zone of a pseudonym. If Alice publishes this zone, then Bob — who knows Alice — can use `music.alice.gnu` to enter one of Alice’s places, while Carol — who only knows Bob, but not Alice — would need to use `music.alice.bob.gnu` to enter the same place.

Readers familiar with earlier designs based on PSYC might want to note that in contrast to earlier designs using PSYC, this use of GNS eliminates the need for place names on the PSYC layer. As each place is purely identified by its public key, a memorable name is assigned to it using a GNS zone entry of a pseudonym.

Label	Type	Data
+	PLACE	$PlaceA_{pub}, H(PeerX_{pub})$
tech	PLACE	$PlaceB_{pub}, H(PeerX_{pub}), H(PeerY_{pub})$
music	PLACE	$PlaceC_{pub}, H(PeerZ_{pub})$

Table 1.2.1: Example GNS zone of a pseudonym

2. Overview of the system

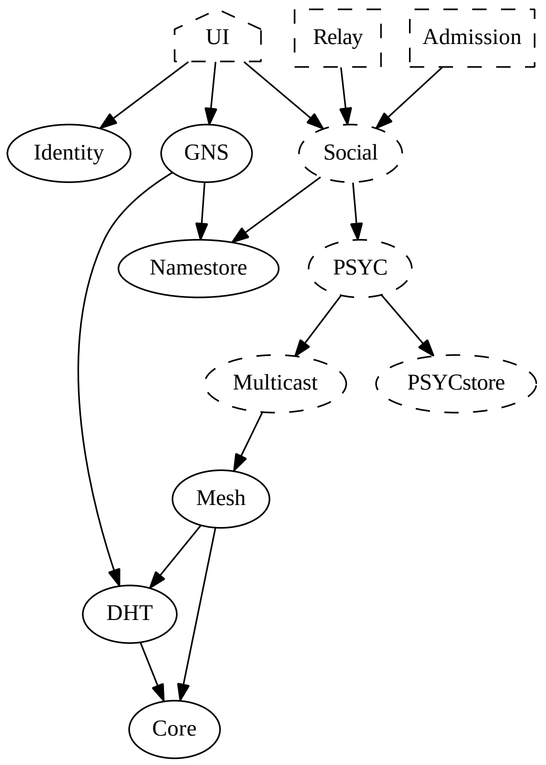

Figure 2.0.1: Components used from GNUnet in the social messaging system. *House: application, box: daemon, circle: service, arrow: dependency, dashed: component described here, to be implemented.*

The social messaging system we propose to create using GNUnet, a modular peer-to-peer framework. The framework consists of several services — each running as a separate process for fault isolation — which use message passing to communicate with each other. Each service provides abstractions other components can use by linking against a shared library which implements the respective API by communication with the service process. Components relevant to the social messaging system are depicted in Figure 2.0.1. The central components in the system are:

The Multicast service: implements reliable messaging in multicast groups. Takes care of managing group membership and message replay.

The PSYC service: parses the PSYC protocol syntax and notifies the social service about incoming method calls, also performs decentralized state operations.

The PSYCstore service: used by the PSYC service to store the decentralized state, messages, and membership information that belong to groups.

The Social service: models places and user identities, and allows applications to register method handlers for method calls they are interested in. Uses GNS for identity management and PKI.

Applications: user interfaces or background processes that handle certain PSYC method calls.

The various layers use different terms to describe related abstractions in an appropriate language for each layer. Table 2.0.1 summarizes the mapping between these different terms, which we will explain in more detail for each layer in the system in the following sections.

Multicast	PSYC	Social
Group	Channel	Place
Origin	Master	Host
Member	Slave	Guest
	State	Environment
Message	Method invocation	listen
		announce
		talk
	State operation	decorate
		look
	Story	History
join	join	enter
part	part	leave

Table 2.0.1: Relationships between the terms at the various layers. Note that as the abstractions change, the semantics of the terms do not always match perfectly; thus, the table describes which concepts on the lower layers are used to realize concepts at the higher layers.

Multicast layer: *members join* a *group* operated by the *origin* to exchange messages.

PSYC layer: the *master* controls a *channel* where *slaves* can *join* to learn about the channel's *state* and *story.* The master can perform *state operations* and trigger *method invocations* on the slaves.

Social layer: *guests* can *enter* a *place*, *talk* to the *host* of the place, *listen* to the host's *announcements*, and *look* at objects in the place's *environment*, which is *decorated* by the host.

We will now describe the main services of the messaging system: multicast, PSYC, PSYCstore and social. Ultimately, each service will run as a separate process and comes with a library that provides an API for accessing functionality of the service.

3. Multicast service

We call a set of peers a *group*. A *policy* is used to determine which peers are allowed to join a group. Each group has a designated member called the *origin* who is in control of the *policy* and the communication. Peers in a group that are not the origin are called *members*. While the origin is in charge of the policy, all members are trusted to enforce it and in particular not to share group communications with non-members. Using the multicast service, the origin can multicast messages to all group members, while group members can only unicast requests to the origin. Figure 3.0.1 illustrates the communication structure provided by a multicast group.

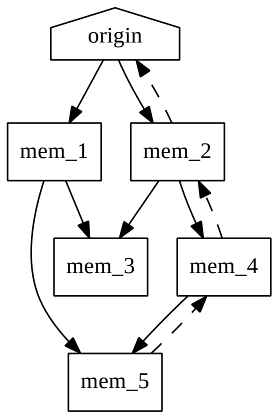

Figure 3.0.1: Participants and message flow in a multicast group. *Solid arrow: multicast message, dashed arrow: unicast request.*

The central functions of the multicast service thus are to organize peers into multicast groups, to enable membership management, and to provide reliable (multicast) message delivery to the peers in these groups. In the multicast system, group members receive message fragments from one or more members, and can relay messages to other members. Each message fragment is signed by the origin to enable the members to verify the integrity and authenticity of the messages.

The multicast service relies on the mesh service for reliable and confidential communication between peers. Mesh provides an end-to-end encrypted secure communication channel between two peers; other peers that may be involved in message passing are not able to access the data, as encrypted messages are secured using ECDHE (Elliptic Curve Diffie-Hellman Ephemeral) exchange and AES (Advanced Encryption Standard) encryption.

3.1 Multicast API overview

The multicast service exposes the following API, which is used by the PSYC service.

3.1.1 Origin

Functions available to the group's origin:

- **Start** (or restart) a multicast group.
- **Stop** a multicast group.
- **Transmit** a message to all members.
- **Join decision:** response to a join request.

Callbacks for the origin:

- **Request** received from a group member.
- **Own messages** fragmented and signed, to store them for replay.

3.1.2 Member

Functions available to group members:

- **Join** (or rejoin) a multicast group.
- **Part** a multicast group.
- **Transmit** a message to the origin.
- **Request replay** of a message from other group members.

Callbacks for group members:

- **Message** received for the group.

3.1.3 Group

Functions available to both the origin and members:

Membership test answer: respond to a membership test query.

Callbacks for the origin and members:

Join request received from a peer.

Membership test: the implementation needs to check whether a member was admitted to the group when the given message fragment was sent out. It is needed to ensure message fragments are relayed only to members.

3.2 Starting the origin

A group is fully functional while its origin is started. The origin is stopped when it goes offline, and can be restarted at a later point. After the origin is started, it can start transmitting multicast messages to group members, and it receives join requests from peers and unicast requests from members.

To start the origin, the following function of the multicast API is used:

```
struct GNUNET_MULTICAST_Origin *
GNUNET_MULTICAST_origin_start
   (const struct GNUNET_CONFIGURATION_Handle *cfg,
    const struct GNUNET_CRYPTO_EccPrivateKey *private_key,
    uint64_t last_fragment_id,
    GNUNET_MULTICAST_JoinCallback join_cb,
    GNUNET_MULITCAST_MembershipTestCallback test_cb,
    GNUNET_MULITCAST_ReplayCallback replay_cb,
    GNUNET_MULTICAST_RequestCallback request_cb,
    GNUNET_MULTICAST_MessageCallback message_cb,
    void *cls);
```

Group's private-public key pair: used to identify the group and sign message fragments sent to group members.

Last fragment ID: when restarting a group, the fragment ID to start numbering fragments from. As each member stores message fragments, this is necessary to make sure a rejoining member can replay missed message fragments.

Callbacks: functions to call for requesting replay, testing membership, as well as notifying about joins, unicast requests, and the origin's own message fragments it sends out. These are explained later in this chapter.

3.3 Stopping the origin

The following function of the multicast API is used to stop the origin:

```
void
GNUNET_MULTICAST_origin_stop
   (struct GNUNET_MULTICAST_Origin *origin);
```

When the origin is stopped, the group might still function partially: a peer can still join at another online member provided one is known, and the peer is already admitted or no admission is required. Replay requests can still be issued to other members. To shut down a group permanently, including all of its members, an application layer message has to be sent to all members by the origin.

Stopping an origin but leaving other members running improves asynchronous messaging capabilities of the group: if a member was offline for a while, it can reconnect to the group and request replay of missed messages, even if the origin is not online at the moment, provided some of the relays the member got during a previous session are still online.

3.4 Joining a group

3.4.1 Requesting join

To join a group as a member, the multicast API provides the following function:

```
struct GNUNET_MULTICAST_Member *
GNUNET_MULTICAST_member_join
 (const struct GNUNET_CONFIGURATION_Handle *cfg,
  const struct GNUNET_CRYPTO_EccPublicKey *group_key,
  const struct GNUNET_CRYPTO_EccPrivateKey *member_key,
  const struct GNUNET_PeerIdentity *origin,
  size_t relay_count,
  const struct GNUNET_PeerIdentity *relays,
  const struct GNUNET_MessageHeader *join_request,
  GNUNET_MULTICAST_JoinCallback join_cb,
  GNUNET_MULITCAST_MembershipTestCallback test_cb,
  GNUNET_MULITCAST_ReplayCallback replay_cb,
  GNUNET_MULTICAST_MessageCallback message_cb,
  void *cls);
```

Group's public key: identifies the group to join, and used to verify signatures of multicast messages.

Member's private-public key pair: used to identify the member and to sign unicast requests sent to the origin, including the join request.

Origin: the peer identity of the group's origin, to send the join request to if no relays are known or available.

Relays: peer identities of members of the group that are known to serve as relays, and can be used to send the join request to. This list can include relays known from a previous session, which allows rejoining a group later to replay missed messages, even if the origin is offline.

Join request: a message from the application initiating the join process. The PSYC layer uses this for initiating state synchronization, while applications can supply additional information needed for joining.

Callbacks: similar callbacks as for the origin, except for the request callback, which is only used for the origin.

When joining a group, the multicast service establishes a mesh tunnel to the relays if any given, or to the origin otherwise. After the tunnel is established, the connecting peer sends the join request provided by the application, with the following message header prepended to it, which contains the group ID, member's public key:

```
struct GNUNET_MULTICAST_JoinRequest {
  struct GNUNET_MessageHeader header;
  struct GNUNET_CRYPTO_EccSignature signature;
  struct GNUNET_CRYPTO_EccSignaturePurpose purpose;
  struct GNUNET_CRYPTO_EccPublicKey group_key;
  struct GNUNET_CRYPTO_EccPublicKey member_key;
  struct GNUNET_PeerIdentity member_peer;
};
```

Header: specifies the message type as a join request, and the size of the message.

Group's public key: associates the mesh tunnel with this multicast group, which is then used in both directions: for multicast messages from the origin and unicast requests to the origin.

Member's public key: identifies the joining member, and used to verify the signature of this request.

Member's peer ID: must match the peer the request is coming from to prevent replaying this join request from another peer, as the join request can be sent to other members as well, not just the origin.

Signature: the rest of the header fields and the message payload signed with the member's private key. This establishes a binding between the peer sending the request and the application layer identity specified by the member's public key.

3.4.2 Receiving the join request

Once the contacted peer — the origin or another member of the group — received the join request and the group policy requires admission control, the multicast service passes the joining member's identity and the message payload of the join request to the *join callback*, which allows the decision to be taken on higher layers of the system:

```
typedef void (*GNUNET_MULTICAST_JoinCallback)
   (void *cls,
    const struct GNUNET_EccPublicKey *member_key,
    const struct GNUNET_MessageHeader *join_request,
    struct GNUNET_MULTICAST_JoinHandle *jh);
```

3.4.3 Responding with a join decision

The multicast service should be informed about the decision using the following API function:

```
struct GNUNET_MULTICAST_ReplayHandle*
GNUNET_MULTICAST_join_decision
   (struct GNUNET_MULTICAST_JoinHandle *jh,
    int is_admitted,
    unsigned int relay_count,
    const struct GNUNET_PeerIdentity *relays,
    const struct GNUNET_MessageHeader *join_response);
```

Join handle: used to match the join decision to the corresponding join request.

Is admitted? Decision whether the member is admitted.

Relays: List of peer identities serving as relays for other members. If it does not contain the peer's own ID, it is treated as a redirect to other relays. This allows distributing the load among group members by building multilevel distribution trees for multicast messages in the group. A member tries to establish connections to multiple relays, which helps preventing a malicious or malfunctioning member blocking messages from part of the group.

Join response: application layer message to be sent back with the decision. It has relevance mostly in case of a negative decision, e.g. to give a reason for rejection, or redirect to another group.

The returned replay handle can be used to bring the joined member up to date, if a higher layer protocol requires it. The PSYC layer uses this to perform state synchronization after join, described later in Section 4.4.5.

3.5 Parting a group

When a member wants to part from a group, the following function is used:

```
void
GNUNET_MULTICAST_member_part
   (struct GNUNET_MULTICAST_Member *member);
```

Calling this function results in shutting down mesh tunnels to all connected group members, but the peer remains a member of the group and can reconnect later, unless an application layer leave request is sent to the origin before parting the group.

3.6 Multicast messages

For sending out a multicast message from the origin to the group, the multicast service provides the following function:

```
struct GNUNET_MULTICAST_OriginMessageHandle *
GNUNET_MULTICAST_origin_to_all
   (struct GNUNET_MULTICAST_Origin *origin,
    uint64_t message_id,
    uint64_t group_generation,
    GNUNET_MULTICAST_OriginTransmitNotify notify,
    void *notify_cls);
```

Message ID & group generation: used to set header fields for each fragment of the message.

Notify callback: function called to provide the next part of the message when buffer space is available. It is kept being called until it indicates the end of the message is reached. To be able to send messages asynchronously, a transmission might be suspended — when there's no data available yet — and resumed later.

The application calls this function to initiate a multicast message transmission to the group. The notify callback is called later, possibly multiple times. While calling this callback for more data, multicast splits the message into fragments less than 64 KB each, which is the maximum size of GNUnet messages[1].

After fragmenting the message, a multicast message header is prepended to each fragment with the following structure:

[1]The fragment payload will be slightly less than 64 KB, due to the message headers of each layer (multicast, mesh, core).

```
struct GNUNET_MULTICAST_MessageHeader {
  struct GNUNET_MessageHeader header;
  uint32_t hop_counter;
  struct GNUNET_CRYPTO_EccSignature signature;
  struct GNUNET_CRYPTO_EccSignaturePurpose purpose;
  uint64_t fragment_id;
  uint64_t fragment_offset;
  uint64_t message_id;
  uint64_t group_generation;
  enum GNUNET_MULTICAST_MessageFlags flags;
};
```

Header: contains the message type for multicast messages and size of the full message fragment, including the header and payload.

Hop counter: number of hops the message fragment has taken from the origin, updated at each hop thus not signed. It is used to determine shortest paths for unicast requests sent from a member to the origin.

Signature: the rest of header fields and the payload signed by the origin. Members use this to verify integrity and authenticity of the fragment.

Signature purpose: specifies the size of signed data and the purpose for signing. Needed for verifying the signature.

Fragment ID: monotonically increasing counter for identifying and ordering message fragments. It is also used to detect missed fragments. Starts at 0 with the first message fragment sent to the group, and the `uint64_t` type used allows a maximum of 2^{64} fragments in a group.

Fragment offset: byte offset of the current fragment, relative to the beginning of the message. The `uint64_t` used here limits the last fragment of a message to start at $2^{64} - 1$ bytes.

Message ID: opaque to multicast, used by applications to reassemble fragments of the same message. Allows for a theoretical maximum of 2^{64} messages, but the fragment ID counter would likely reach this limit earlier, which is incremented for every fragment of a message.

Group generation: used in groups with restricted history, incremented whenever a member leaves the group. It is used to detect changed membership, even if there are missed messages, as explained in Section 3.7.

Flags: indicate the first and last fragment of a message. By using a last fragment flag, the size of a message do not have to be prematurely known.

The multicast service does not wait for a full message to arrive before it starts sending fragments to the network: the transmission function uses a notify callback to ask for data for the next fragment, and is called until the there's more data for the message. Message transmission happens similarly in case of the higher layer services as well: the body of a larger message sent by an application is transmitted in smaller fragments between the services. This way larger messages can be sent asynchronously to group members, even with smaller messages in between the fragments of a larger message.

After the origin sent out a message fragment, directly connected members receive it, and relay it further towards the rest of the group, possibly through multiple hops.

Before a message fragment can be relayed, a membership test has to be performed for each directly connected member, to determine whether they are supposed to see that particular message fragment. The fragment is then only relayed to those members for whom the membership test returns a positive result. If a fragment was not relayed to a member due to the membership test not being able to confirm membership because of missed messages, this member would notice the missed fragment later and can request it replayed from this or another member of the group.

3.7 Testing membership

The multicast service calls the *membership test* callback of the API when it needs to make sure a given member has access to a particular message fragment when relaying fragments to other members.

```
typedef void (*GNUNET_MULTICAST_MembershipTestCallback)
   (void *cls,
    const struct GNUNET_EccPublicKey *member_key,
    uint64_t message_id,
    uint64_t group_generation,
    struct GNUNET_MULTICAST_MembershipTestHandle *mth);
```

The implementation of this callback then looks up locally stored membership information in order to answer whether the member in question was admitted when a message fragment with the given *message ID* and *group generation* was sent to the group. The *group generation* header field is a counter incremented whenever a member is removed from the group, and helps determining group membership even in case of missed fragments: as long as the group generation stays the same, no one has left the group, thus it is safe to relay further fragments.

The result of the membership test is provided with the following API function:

```
void
GNUNET_MULTICAST_membership_test_result
   (struct GNUNET_MULTICAST_MembershipTestHandle *mth,
    int result);
```

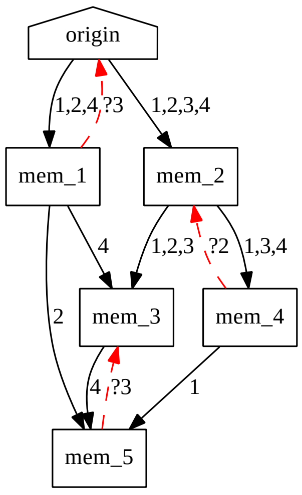

Figure 3.8.1: Message fragment distribution and replay in a multicast group. *The number on an edge indicates the fragment ID. A solid arrow is used for multicast messages, a dashed arrow represents replay requests.*

The *result* is either `GNUNET_YES`, `GNUNET_NO`, or `GNUNET_SYSERR` if there's not enough information stored to be able to answer the query, or another error occurred that prevents access to the membership store.

To reduce the number of membership tests that query the local database, the multicast service remembers the membership test result together with the group generation for each connected member. Then it is enough to compare the group generation of the to be relayed fragment with the last known value that passed the membership test for each member. When this simple test returns a negative result, because the group generation changed, a membership test is performed again for each directly connected member, to determine whether any of them left the group before relaying the fragment.

3.8 Replaying multicast messages

When a group member receives a fragment, it uses the *fragment ID* header field to detect any missed fragments. When a missing fragment is noticed, the multicast service requests replay of the fragment from another group member. Figure 3.8.1 illustrates message distribution and replay in a group. Refer to Section A.1.2 of the appendix for a sequence diagram of the API function calls and messages sent between the services during replay.

Each missed fragment has to be requested individually, which is necessary to avoid a member requesting a large number of fragments at once that could lead to denial of service. Fragments can arrive from multiple sources and out of order, for this reason it is also more efficient to request recently missed fragments one by one, as an already requested fragment could have arrived in the mean time from another source.

Replay of a fragment can be also requested through the multicast API explicitly, using one of the following functions.

One way to request a fragment is by *fragment ID*:

```
struct GNUNET_MULTICAST_MemberReplayHandle *
GNUNET_MULTICAST_member_replay_fragment
   (struct GNUNET_MULTICAST_Member *member,
    uint64_t fragment_id,
    uint64_t flags);
```

Another possibility is to specify a *message ID* and a *fragment offset*:

```
struct GNUNET_MULTICAST_MemberReplayHandle *
GNUNET_MULTICAST_member_replay_message
   (struct GNUNET_MULTICAST_Member *member,
    uint64_t message_id,
    uint64_t fragment_offset,
    uint64_t flags);
```

As applications deal with message IDs instead of fragment IDs, this latter form is useful when an application wants to retrieve all fragments of a particular message.

In both cases replay *flags* can be specified, which are opaque to multicast and passed to the *replay callback* of the multicast API that notifies about incoming replay requests.

When receiving a replay requests, the multicast service notifies about it using one of the following callbacks:

```
typedef void (*GNUNET_MULTICAST_ReplayFragmentCallback)
   (void *cls,
    const struct GNUNET_CRYPTO_EccPublicKey *member_key,
    uint64_t fragment_id,
    uint64_t flags,
    struct GNUNET_MULTICAST_ReplayHandle *rh);
```

```
typedef void (*GNUNET_MULTICAST_ReplayMessageCallback)
   (void *cls,
    const struct GNUNET_CRYPTO_EccPublicKey *member_key,
    uint64_t message_id,
    uint64_t fragment_offset,
    uint64_t flags,
    struct GNUNET_MULTICAST_ReplayHandle *rh);
```

This indicates the *member's public key* the request is coming from, either a *fragment ID* or a *message ID* and *fragment offset* that identifies the fragment, and the *flags* set for the query.

The implementation of the callback then makes sure the member has access to the requested fragment by performing a membership test, and either returns the fragment from the message store or an error code via the following API call:

```
void
GNUNET_MULTICAST_replay
   (struct GNUNET_MULTICAST_ReplayHandle *rh,
    const struct GNUNET_MessageHeader *message,
    enum GNUNET_MULTICAST_ReplayErrorCode error_code);
```

Replay handle: used to match the response with replay request.

Message: the message fragment retrieved from the message store, where it is stored exactly as it arrived from the network along with the signature of the origin, thus the authenticity of a replayed fragment can still be verified by the receiving member.

Error code: used to indicate any errors that can occur during replay.

- **OK:** everything is fine, the provided fragment is returned to the requester. If the result is not *OK*, a message with the error code is returned instead.
- **Not found:** the message fragment was not found in the message store, either it was already removed because it was too old, or the member has missed it and have never seen it.
- **Access denied:** the member does not have access to the requested fragment. Returned after a failed membership test.
- **Internal error:** an error occurred replay, e.g. the database was not available.

Another variant of the replay function is provided, which uses a *notify callback* to return the message. This can be used to replay a message generated by the responding member, which does not contain the signature of the master, and thus it is verified by other means on higher layers.

```
void
GNUNET_MULTICAST_replay2
   (struct GNUNET_MULTICAST_ReplayHandle *rh,
    GNUNET_MULTICAST_ReplayTransmitNotify notify,
    void *notify_cls);
```

A replay response can consist of multiple fragments in certain cases, for this reason the end of the replay session is explicitly indicated using the following function:

```
void
GNUNET_MULTICAST_replay_end
   (struct GNUNET_MULTICAST_ReplayHandle *rh);
```

3.9 Unicast requests

A member of the group can send a message with a given ID to the origin using the following API function:

```
struct GNUNET_MULTICAST_MemberRequestHandle *
GNUNET_MULTICAST_member_to_origin
   (struct GNUNET_MULTICAST_Member *member,
    uint64_t message_id,
    GNUNET_MULTICAST_MemberTransmitNotify notify,
    void *notify_cls);
```

The origin is then informed about an incoming request via the following callback of the API:

```
typedef void (*GNUNET_MULTICAST_RequestCallback)
   (void *cls,
    const struct GNUNET_EccPublicKey *member_key,
    const struct GNUNET_MULTICAST_RequestHeader *request,
    enum GNUNET_MULTICAST_MessageFlags flags);
```

An obvious way to implement such requests would be to establish a direct mesh tunnel from the member to the origin, but this would be harder scale to a large number of participants.

Another solution could be to use the reverse path of multicast messages for this purpose. In this case members would relay a message fragment towards the origin, choosing the upstream connection closest to the origin, which is determined using distance information present in the *hop counter* header field of incoming multicast messages.

A unicast request would than have a message header — similar to the multicast message header — prepended to an application layer message payload, with the following fields.

```
struct GNUNET_MULTICAST_RequestHeader
{
  struct GNUNET_MessageHeader header;
  struct GNUNET_CRYPTO_EccPublicKey member_key;
  struct GNUNET_CRYPTO_EccSignature signature;
  struct GNUNET_CRYPTO_EccSignaturePurpose purpose;
  uint64_t fragment_id;
  uint64_t fragment_offset;
  uint64_t message_id;
  enum GNUNET_MULTICAST_MessageFlags flags;
};
```

Member's public key: identifies the member the request is coming from.

Signature: the request signed with the member's private key. Used by the origin to verify the authenticity of the message, as it can be relayed by other group members.

Fragment ID, fragment offset, message ID, and flags: refer to identically named fields of multicast messages in Section 3.6. These fields allow unicast requests to be fragmented, similarly to multicast messages.

Such member-to-origin requests would still have to be encrypted to prevent relaying group members having access to the message content. For this purpose the member would encrypt each message fragment sent to the origin using a session key agreed upon after an ECDH (Elliptic Curve Diffie-Hellman) exchange with the origin.

Finally, an alternative to sending unicast requests from a member to the origin is using a separate multicast group with reversed roles, and connect the messages on the application layer by adding a reference to a previous message of another multicast group. This allows for a more decentralized communication model, which is suitable for e.g. (micro)blogging applications.

4. PSYC service

The PSYC service models PSYC channels, each using an underlying multicast group. The multicast group's origin corresponds to the channel master, while its members are called channel slaves on the PSYC layer. The channel master is the owner of the channel that can send messages to it, while a channel slave is a subscriber of the channel, who can receive multicast messages from and send unicast requests to the channel master.

Messages sent to a channel use the PSYC syntax. The PSYC service parses messages it receives from the underlying multicast group, and renders outgoing messages in the PSYC format before sending them out to other members of the group. Once a message is parsed, it notifies about the method call in the message via the PSYC API. The PSYC service does not generate messages by itself, and does not understand the semantics of messages, only the syntax.

Each PSYC channel has a decentralized channel state associated with it: a set of persistent key-value pairs replicated on each channel slave. The channel master is in charge of the state: it sends out state modifier operations to the channel. Channel slaves apply these changes to their local state database using the PSYCstore service. As each slave receives the same operations from the master, this results in each slave having the same channel state after receiving all state modifiers the master sent out.

A PSYC message consists of the following parts, in this order:

Routing header: replaced by the multicast message header when PSYC is used over GNUnet.

Length of the message: not needed when the PSYC syntax is used inside GNUnet messages, as the multicast header already indicates the size of each fragment and the last fragment of a message.

Entity header: contains transient variables and state modifiers set by an application. A state modifier defines an operation on the channel state, and consists of an operation, the name of a state variable to modify, and the argument(s) for the operation. The two most important operators are : (set) and = (assign). The former sets a transient variable only for the message it appears in, the latter assigns a value to a persistent state variable. Further operations on state variables are possible, depending on the type of variable. The variable types and the operations on them are going to be specified later on, as part of future work.

Method name to invoke on the recipient applications. This is the only compulsory part of the message.

Message body: the payload for the message sent by an application.

The following is an example of a message invoking a `_message_public_shout` method with a transient `_volume` variable, which is not saved in the channel state:

```
:_volume        100
_message_public_shout
Hello,
world!
```

The next message contains two state modifiers, each sets a variable for a profile field:

```
=_location_planet       Earth
=_location_continent    Europe
_notice_profile_location
```

The PSYC service uses the PSYCstore service to store message fragments sent to a channel, and to apply channel state operations found in messages. Storing message fragments is necessary in order to facilitate history queries from applications, and replay requests from multicast group members.

The PSYC service adds a message header to each PSYC message before it sends it through the multicast service. This header contains additional fields needed by the PSYC layer. One of these fields is the size of the message up to the end of the method name, which is necessary so that parsing of the message can start after all fragments containing modifiers arrived, but before the message body is complete. This way applications can already be notified of a possibly large message before its whole body arrives.

4.1 PSYC API overview

The PSYC API provides functionality for participants of a PSYC channel depending on their role. One set of functions are available only to the channel master, while another set only to channel slaves. There is also a third set of functions available to both the channel master and slaves.

4.1.1 Channel master

Functions available to the channel master:

- **Start** (or restart) a channel. Starts the underlying multicast group.
- **Stop** a channel. Stops the multicast group and terminates the channel master.
- **Transmit** a message to a channel through the multicast service.
- **Join decision:** respond to a join request, which is transmitted to the requesting peer through the multicast service.

Callbacks for the channel master:

- **Request received** from a channel slave.

4.1.2 Channel slave

Functions a channel slave has access to:

- **Join** (or rejoin) a remote channel. Sends a join request to the master through the multicast service.
- **Part** a remote channel. Disconnects from a multicast group and terminates the channel slave. An application layer leave request should be sent to the master before parting.
- **Transmit** a request to the master using the multicast service.

Callbacks used to notify a channel slave:

- **Message received** from the channel master.

4.1.3 Channel

Functions available to both the master and slaves are:

- **Add/remove a slave** to/from the membership database of the channel.
- **Get historic messages** for a specified message ID range.
- **Get a state variable** that best matches a given name.
- **Get all state variables** that matches a given name prefix.

4.2 Starting the channel master

The PSYC channel master corresponds to the group origin on the multicast layer. When starting a PSYC channel master, the underlying multicast group origin is started as well, as described in Section 3.2. The channel can be started and restarted using the following API function:

```
struct GNUNET_PSYC_Master *
GNUNET_PSYC_master_start
   (const struct GNUNET_CONFIGURATION_Handle *cfg,
    const struct GNUNET_CRYPTO_EccPrivateKey *channel_key,
    enum GNUNET_PSYC_Policy policy,
    GNUNET_PSYC_Method method,
    GNUNET_PSYC_JoinCallback join_cb,
    void *cls);
```

Channel's private-public key pair: used to identify the channel and to sign multicast messages sent to the channel.

Policy: admission control and history access settings. It is a combination of the following flags:

> **Admission control:** join requests have to be approved by an application running on the master.
>
> **Restricted history:** a past message can be replayed to only those slaves who were already admitted when the message was originally sent out. To enforce this, the PSYC service maintains a group generation counter incremented after a slave is removed from the channel.

The combination of these flags results in the following policies:

> **Anonymous channel:** no admission control and no restrictions on history. A channel slave can join directly at any other known slave, without the master having to admit the slave first. As the master does not necessarily know about each slave in this case, joins are not announced.
>
> **Private channel:** admission control with restricted history. A slave has to send a join request to the master first, and the master announces joins so that channel slaves know who the other currently admitted slaves are, which is needed to ensure only authorized slaves receive relayed and replayed messages.
>
> **Closed channel:** admission control without restricted history. It behaves like a private channel, but slaves have access to past messages as well.

Public channel: no admission control with restricted history. In this case joining slaves are announced to be able to apply history restrictions, for this reason a join request has to be sent to the master first, but slaves are auto-admitted.

Method callback: to inform the social service about method calls in requests from slaves.

Join callback: to inform the social service about join requests.

Before a channel master can be restarted, the PSYC service retrieves the state of counters maintained by the PSYC service from the PSYCstore, so that it can continue incrementing them from their last value. These counters are the message ID, group generation, and state delta. They are described later in this chapter.

4.3 Stopping the channel master

The channel master can be stopped using the following API function:

```
void
GNUNET_PSYC_master_stop (struct GNUNET_PSYC_Master *master);
```

When this function is called, the PSYC service stops the underlying multicast group origin, stops maintaining information related to the channel, and disconnects the caller from the PSYC service.

4.4 Joining a channel

When joining or rejoining a channel as a slave, the PSYC service initiates joining the underlying multicast group, which in turn involves sending out a join request to the origin or one of the given relays, as described in Section 3.4. Also refer to Section A.4 of the appendix for sequence diagrams of the join process.

4.4.1 Requesting join

The PSYC service provides the following API function for joining a channel:

```
struct GNUNET_PSYC_Slave *
GNUNET_PSYC_slave_join
   (const struct GNUNET_CONFIGURATION_Handle *cfg,
    const struct GNUNET_CRYPTO_EccPublicKey *channel_key,
    const struct GNUNET_CRYPTO_EccPrivateKey *slave_key,
    const struct GNUNET_PeerIdentity *origin,
    size_t relay_count,
    const struct GNUNET_PeerIdentity *relays,
    GNUNET_PSYC_Method method,
    GNUNET_PSYC_JoinCallback join_cb,
    void *cls,
    const char *method_name,
    const struct GNUNET_ENV_Environment *env,
    size_t data_size,
    const void *data);
```

Channel's public key: identifies the channel to join.

Slave's private-public key pair: the public key identifies the slave, while the private key signs unicast requests sent to the master.

Origin and relays: peer identifiers to join the multicast group at, and send requests to, as described in Section 3.4.

Method callback for handling method invocations in multicast messages sent to the group.

Join callback for handling other slaves joining.

Method name, environment, and data: a PSYC-formatted message is constructed from these parameters, which will serve as the join request sent to the master.

Upon calling this function, the PSYC service joins the multicast group using the provided information. It also adds a PSYC message header to the join message it constructs, which indicates the last known message ID that modified the state — this information is requested from the PSYCstore, and needed later for the state synchronization process, which happens right after admission.

4.4.2 Responding to a join request

After the origin or another group member received the join request, the *join callback* of the multicast service notifies the PSYC service about it, which queries the PSYCstore to determine whether the member is already admitted: if it is, a positive join decision is sent back to the requester.

In case the requester is not a member yet, the process continues differently depending on whether the request was sent to the master or a slave. In the latter case a negative join decision is sent back right away, as a slave is not in charge of the channel. In case of the master, however, the decision is taken by an application, and thus the join request is propagated to higher layers via the *join callback* of the PSYC API:

```
typedef int (*GNUNET_PSYC_JoinCallback)
   (void *cls,
    const struct GNUNET_EccPublicKey *slave_key,
    const char *method_name,
    size_t variable_count,
    const GNUNET_ENV_Modifier *variables,
    size_t data_size,
    const void *data,
    struct GNUNET_PSYC_JoinHandle *jh);
```

Public key identifying the slave requesting join.

Method name, variables, data: parts of the parsed PSYC message sent along with the join request, which contain additional information required for the join on the application layer.

Join handle: used to match the join request with the decision.

Once an application running in the master channel has made a decision about the request, there are three steps remaining for the master to complete the join process: notifying the channel about the new slave, informing the requester about the decision, and synchronizing the channel state of the requester.

4.4.3 Notifying the channel about the new slave

If the decision taken by the application is to admit the slave requesting join, then the slave needs to be added to the membership database of each participant of the channel: the master as well as all the slaves. This is necessary in order to be able to answer membership test queries later on.

Adding a slave to the membership database of the channel can be performed via the following function of the PSYC API:

```
void
GNUNET_PSYC_channel_slave_add
   (struct GNUNET_PSYC_Channel *channel,
    const struct GNUNET_EccPublicKey *slave_key,
    uint64_t announced_at,
    uint64_t effective_since);
```

Channel: either a master or slave.

Slave's public key: identifies the slave this membership change is about.

Announced at: ID of the message that announced the membership change. Needed for ordering of events.

Effective since: message ID since the membership change is in effect. This allows the master to give the slave access to past messages from an earlier point in time.

After this function is used by the channel master to update its membership database, the master still needs to transmit a message to notify the channel about the newly joined slave. Existing channel slaves then receive this message, the social service understands the method call notifying about the join, and calls this function to update its own membership database of the channel.

The membership database is updated using the PSYCstore API. The process of storing membership change events is described in Section 5.4.1.

4.4.4 Informing the requester about the decision

Finally, the requester needs to be informed about the join decision. In order to do this, the PSYC service is first notified about the decision using the following API function:

```
void
GNUNET_PSYC_join_decision
   (struct GNUNET_PSYC_JoinHandle *jh,
    int is_admitted,
    unsigned int relay_count,
    const struct GNUNET_PeerIdentity *relays,
    const char *method_name,
    const struct GNUNET_ENV_Environment *env,
    size_t data_size,
    const void *data);
```

Join handle that identifies the request this decision is about.

Is admitted? The decision taken: `GNUNET_YES` or `GNUNET_NO`.

Relays: other multicast group members that can be used as relays. The list of relays are distributed in application layer messages by the master.

Method name, environment with variables, data: parts of the PSYC message to be returned with the decision.

The join decision is then forwarded to the multicast service, which informs the requesting peer about the decision, as described in Section 3.4.

4.4.5 State synchronization

In case of a positive join decision, the PSYC service initiates a state synchronization process to bring the state of the joining slave up to date. The PSYC service chooses between two approaches based on the latest state message ID the joining slave sent along with join request. The first option is to send the latest full state for which a hash is known and replay any additional state modifying messages that were sent after that. The other option is to replay only the individual state modifying messages that the joining slave does not yet have. The cheaper method in terms of bandwidth can be determined based on the size of the full state and the state modifying messages stored in the PSYCstore.

When the full state is used for state synchronization, a message containing all state variables are generated by the responding slave, thus not signed by the master. In order for the joining slave to be able to verify the authenticity of the received state, the channel master periodically generates a hash of all state variables in lexical order, and adds it to a *state hash* PSYC header field. This hash is always added to multicast messages notifying about a join, as a new slave might not have access to earlier state messages due to history restrictions. The state hash allows the joining slave to verify the state it received, as each multicast message — including the one containing this hash — is signed by the master. If the hash of the received state passes the verification, it is stored in the PSYCstore, and further state modifiers can be applied to it.

4.5 Parting a channel

For slaves wanting to part a channel, the PSYC API provides the following function:

```
void
GNUNET_PSYC_slave_part
   (struct GNUNET_PSYC_Slave *slave);
```

When this function is called, the PSYC service parts the multicast group and disconnects the caller from the PSYC service. Calling this function does not result in any message being sent out to the network that would change the membership status of the slave. To remove a slave from the membership database of the channel, the following function of the PSYC API is used:

```
void
GNUNET_PSYC_channel_slave_remove
   (struct GNUNET_PSYC_Channel *channel,
    const struct GNUNET_EccPublicKey *slave_key,
    uint64_t announced_at);
```

This function does the opposite of `GNUNET_PSYC_channel_slave_add()` — described in Section 4.4.3 — and it is used in a similar way. It removes the given slave from the channel, and needs to be called by the master as well as every slave of the channel.

The master calls this function either upon a specific request of a slave, or after an unilateral decision to remove the slave from the channel. In either case, the master sends out a message to the channel informing the slaves about the membership change, so that the slaves can call this function, too, in order to update their own membership database. The group generation is incremented after this message has been sent to the channel, which makes sure the leaving slave will not have access to further messages, but it still gets to see this last message, either confirming its own leave request or informing about being kicked.

4.6 Messages from the master

4.6.1 Sending messages

The master can transmit a message to the channel using the following API function:

```
struct GNUNET_PSYC_MasterTransmitHandle *
GNUNET_PSYC_master_transmit
   (struct GNUNET_PSYC_Master *master,
    const char *method_name,
    const struct GNUNET_ENV_Environment *env,
    GNUNET_PSYC_MasterReadyNotify notify,
    void *notify_cls,
    enum GNUNET_PSYC_MasterTransmitFlags flags);
```

Master handle for the channel to send the message to.

Method name for the message.

Environment containing state modifiers and transient variables for the message.

Notify callback to notify the caller when it can provide the next part of the body, called possibly multiple times while it still returns data.

Flags for the message being transmitted:

> **Reset state:** whether this message resets the channel state, i.e. removes all previously stored state variables. This is achieved by setting the *state delta* PSYC header field to 0. This field is described in the next section.
>
> **Add state hash:** generate a hash of the full state and add it to the PSYC header of the message in a *state hash* field.

Increment group generation: whether the message contains a notification about a slave removed from the channel, in which case the group generation is incremented after sending this message.

Upon calling this function, the PSYC service renders a PSYC formatted message from the method name, modifiers, and message body. Then it uses the `GNUNET_MULTICAST_origin_to_all()` function — described in Section 3.6 — to send out a possibly fragmented message to the underlying multicast group. The method name and modifiers are sent out right away, while parts of the body are transmitted as data for it comes in.

4.6.2 Receiving messages

Channel slaves are notified about incoming multicast message fragments via the *message callback* of the multicast service. The PSYC service stores each incoming fragment using the PSYCstore service. Next, to determine how to proceed, it parses the first fragment of the message, which contains the length of the modifiers section and can contain a *state delta* PSYC header field.

The *state delta* is present in each message that contains state modifiers, and it is set to the number of messages since the last message that modified the state. The value 0 means a full state reset, in which case all previously stored state variables are discarded. The state delta allows a channel slave to detect a previously missed message that contained state modifiers, in which case it can not apply further operations on the channel state until all missing messages arrived, as state operations must be performed strictly in the order they were sent out by the master to keep the state consistent across the slaves. The missing messages are requested to be replayed via the multicast service, which then replays explicitly requested message IDs with higher priority.

Once there are no missing state messages, the PSYC service still has to wait until all fragments of the modifiers section of the current message arrived before it can start applying operations to the channel state.

After all fragments necessary to apply the outstanding state operations arrived, the PSYC service retrieves these from the PSYCstore, parses the modifiers found in them, and sends the parsed operator–name–value triplets back to the PSYCstore for applying them to the current channel state. This process is described in Section 5.3.1.

At this point, when all modifiers of a message are parsed and applied to the channel state, the PSYC service notifies about the incoming method call through the method callback of its API:

```
typedef int (*GNUNET_PSYC_Method)
   (void *cls,
    const struct GNUNET_CRYPTO_EccPublicKey *slave_key,
    uint64_t message_id,
    const char *method_name,
    size_t modifier_count,
    GNUNET_PSYC_Modifier *modifiers,
    uint64_t data_offset,
    size_t data_size,
    const void *data,
    enum GNUNET_PSYC_MessageFlags flags);
```

Slave's public key: `NULL` for multicast messages (only used in case of unicast requests from slaves, see Section 4.7).

Message ID: identifier of the message.

Method name: method name found in the message.

Modifiers: state modifiers and transient variables in the message.

Data offset: byte offset of the fragment of the data.

Data: fragment of the message body starting from the offset above.

Flags: indicate the first and last fragment of a message.

This function is called first with the method name, modifiers, and first part of the message body, then called again for each further fragment of the body as they arrive. For subsequent calls the modifiers and method name are not present, and the end of the message is indicated with the *last fragment* flag set.

In case of the master, requests from slaves are processed similarly, except that in case of the *request callback* used by the master, the slave's public key is set, which identifies the source of the message.

4.7 Requests from slaves

Slaves can send requests to the master using the following function of the API, which is very similar to the master's transmit function described in Section 4.6.1.

```
struct GNUNET_PSYC_SlaveTransmitHandle *
GNUNET_PSYC_slave_transmit
   (struct GNUNET_PSYC_Slave *slave,
    const char *method_name,
    const struct GNUNET_ENV_Environment *env,
    GNUNET_PSYC_SlaveReadyNotify notify,
    void *notify_cls,
    enum GNUNET_PSYC_SlaveTransmitFlags flags);
```

Slave handle for the channel whose master to send the request to.

Method name for the message.

Environment containing transient variables for the message.

Notify callback to notify the caller when it can provide the next part of the body, called possibly multiple times while it still returns data.

Flags for the message being transmitted. None defined yet, reserved for later use.

4.8 History and state requests by applications

An application running in the channel might need to have access to historic messages and state variables it does not keep track of. Both of these are stored by the PSYCstore service, and are accessible through the PSYC service.

4.8.1 Requests for historic messages

Historic messages of a master or slave channel can be requested using the following PSYC API call:

The messages are retrieved from the PSYCstore if available, otherwise a replay request is issued to another member through the multicast service.

```
struct GNUNET_PSYC_Story *
GNUNET_PSYC_channel_story_tell
   (struct GNUNET_PSYC_Channel *channel,
    uint64_t start_message_id,
    uint64_t end_message_id,
    GNUNET_PSYC_Method method,
    GNUNET_PSYC_FinishCallback finish_cb,
    void *cls);
```

The PSYC service retrieves the messages between the *start* and *end message ID* (inclusively) from the PSYCstore, and processes them similarly to new messages coming from the multicast service: it parses the fragments of each message, then calls the provided *method callback* to inform about the method call, transient variables, state modifiers, and the body of the message. For messages retrieved from the PSYCstore a *historic* flag is also set to make them distinguishable from new messages. If the PSYCstore does not have all the requested messages, replay requests are issued through the multicast service for the missing ones.

4.8.2 State variable requests

The PSYC API provides two ways for requesting state variables of a channel: either request all variables with a given name prefix, or request a single variable that best matches the given name.

For the former case the following function is provided:

```
uint64_t
GNUNET_PSYC_channel_state_get_all
   (struct GNUNET_PSYC_Channel *channel,
    const char *name_prefix,
    GNUNET_PSYC_StateCallback cb,
    void *cb_cls);
```

This function retrieves either exactly matching or more specific state variables with a matching *name prefix* from the PSYCstore and returns them through the provided callback. For instance, a request for `_a_b` would return both `_a_b` and `_a_b_c`, while an empty string as prefix would return all state variables in the channel.

The other option is requesting a variable best matching a given name:

```
struct GNUNET_PSYC_StateQuery *
GNUNET_PSYC_channel_state_get
   (struct GNUNET_PSYC_Channel *channel,
    const char *full_name,
    GNUNET_PSYC_StateCallback cb,
    void *cb_cls);
```

In this case the returned variable is either exactly matching or less specific than the requested one: e.g. when a request for `_a_b_c` is made, the first variable is returned that exists in the following order: `_a_b_c`, `_a_b`, `_a`.

5. PSYCstore service

Persistent storage needs of the system is provided by the PSYCstore service. It is used to store message fragments, state variables, and membership information of channels.

5.1 PSYCstore API overview

The PSYCstore provides the following functionality, exposed through its API:

Store a fragment of a message.

Retrieve a fragment by fragment ID.

Retrieve a message with all its fragments.

Apply state modifiers of a message.

Update state hash: update signed values of state variables to be able to serve state synchronization queries later.

Store membership: store join and part events in a channel to be able to perform membership tests later.

Membership test: check whether a channel slave has access to a given message.

Get counter values for a channel. Returns the latest values of the fragment ID, message ID, and group generation, which is needed when restarting a channel master.

5.2 Message store

Storing messages is necessary for two purposes: other members of the multicast group can request replay of missed messages, and applications can request message history.

Incoming message fragments for PSYC channels are stored in the same format as they arrive from the network in order to facilitate message replays. Each message fragment is signed by the channel master, slaves thus can verify integrity of replayed fragments.

5.2.1 Storing message fragments

A message fragment is stored using the following PSYCstore API call:

```
struct GNUNET_PSYCSTORE_OperationHandle *
GNUNET_PSYCSTORE_fragment_store
   (struct GNUNET_PSYCSTORE_Handle *h,
    const struct GNUNET_CRYPTO_EccPublicKey *channel_key,
    const struct GNUNET_MULTICAST_MessageHeader *message,
    GNUNET_PSYCSTORE_ResultCallback rcb,
    void *rcb_cls);
```

Handle for the PSYCstore.

Channel's public key: identifies the channel the message fragment was sent to.

Multicast message header followed by its payload.

State modifier flag: whether the message contains state modifiers. A *state delta* PSYC header field is present if the message contains state modifiers.

Result callback: to inform the caller whether the asynchronous storage operation succeeded: the PSYCstore library has to send the message fragment to the PSYCstore service, which stores it in a database on disk.

After receiving a message fragment, the PSYCstore service stores it along with certain fields extracted from the multicast message header for quick lookup. The following fields are stored for each fragment, shown with SQLite syntax:

```
CREATE TABLE messages (
  channel_id BLOB NOT NULL,
  fragment BLOB NOT NULL,
  fragment_id UNSIGNED INT NOT NULL,
  fragment_offset UNSIGNED INT NOT NULL,
  message_id UNSIGNED INT NOT NULL,
  group_generation UNSIGNED INT NOT NULL,
  multicast_flags UNSIGNED INT NOT NULL,
  psyc_flags UNSIGNED INT NOT NULL,
  PRIMARY KEY (channel_id, fragment_id),
  UNIQUE (channel_id, message_id, fragment_offset)
);
```

Channel ID: hash of the public key of the channel.

Message fragment: raw data of the multicast message fragment, as it arrived from the network.

Fragment ID from the multicast header.

Message ID from the multicast header.

Fragment offset from the multicast header.

Group generation from the multicast header. Used to determine its last value when restarting a channel.

Multicast flags: message flags from the multicast header, which indicate the first and last fragment.

PSYC flags: message flags from the PSYC header, necessary for channel state processing:

- **State modifier flag:** whether the message contains state modifiers.
- **State applied flag:** whether state modifiers in the massage has been applied to the stored state. Applying modifiers can be delayed if there was a missed message that modified the state (detected by PSYC using the state delta header field).
- **State hash flag:** set after the channel master sends out a hash of the channel state and after the signed variable values in the channel state store are updated. Together with the *state applied* flag, it is used to determine from which point incoming state modifiers can be applied to the channel state after rejoining a channel later. Also used during the state synchronization process to determine from which point to send state modifying messages to the joining slave.

5.2.2 Retrieving messages

Two functions are provided to retrieve message fragments from the PSYCstore. The first one retrieves a single fragment with the given *fragment ID*, which is used to serve replay requests coming from the multicast service:

```
struct GNUNET_PSYCSTORE_OperationHandle *
GNUNET_PSYCSTORE_fragment_get
   (struct GNUNET_PSYCSTORE_Handle *h,
    const struct GNUNET_CRYPTO_EccPublicKey *channel_key,
    uint64_t fragment_id,
    GNUNET_PSYCSTORE_FragmentResultCallback rcb,
    void *rcb_cls);
```

The second variant retrieves all fragments of a message with the given *message ID*, this is used to answer requests for historic messages by applications:

```
struct GNUNET_PSYCSTORE_OperationHandle *
GNUNET_PSYCSTORE_message_get
   (struct GNUNET_PSYCSTORE_Handle *h,
    const struct GNUNET_CRYPTO_EccPublicKey *channel_key,
    uint64_t message_id,
    GNUNET_PSYCSTORE_FragmentResultCallback rcb,
    void *rcb_cls);
```

In both cases the fragments are requested from the PSYCstore and returned asynchronously via the provided *fragment result callback*.

5.3 Channel state store

5.3.1 Applying state modifiers

The channel state store contains name-value pairs of state variables for each PSYC channel. The following fields are stored:

```
CREATE TABLE state (
  channel_id BLOB NOT NULL,
  name TEXT NOT NULL,
  value_current TEXT,
  value_signed TEXT,
  PRIMARY KEY (channel_id, name)
);
```

Channel ID: hash of the public key that identifies the channel the state variable belongs to.

Name of the variable.

Signed value of the variable.

Current value of the variable.

The channel state is updated by the master by adding state modifiers to messages. After the PSYC service has parsed modifiers in an incoming message, it uses the following function of the PSYCstore API to apply them to the stored channel state:

```
struct GNUNET_PSYCSTORE_OperationHandle *
GNUNET_PSYCSTORE_state_modify
   (struct GNUNET_PSYCSTORE_Handle *h,
    const struct GNUNET_CRYPTO_EccPublicKey *channel_key,
    uint64_t message_id,
    uint64_t state_delta,
    size_t modifier_count,
    const struct GNUNET_ENV_Modifier *modifiers,
    GNUNET_PSYCSTORE_ResultCallback rcb,
    void *rcb_cls);
```

Channel's public key: identifies the channel the state of which to modify.

Message ID the state modifiers appeared in.

State delta of the message: the number of message IDs since the last message that contained state operations. PSYCstore uses this to check whether it has applied all previous state modifiers by checking whether the message the state delta is referring to is present in the message store and has the *state applied* flag set. If this check passes, the modifiers are applied to the channel state store.

Modifiers: operation–name–value triplets that specify the operations to apply on state variables.

Calling this function results in updating the *current values* in the state store, the signed values remain unchanged, to be able to serve state synchronization requests.

5.3.2 State hash updates

Signed values in the state store are updated after the channel master sends out a hash of the full state. When a channel slave receives this hash, the PSYC service calls the following function of the PSYCstore API, which informs the PSYCstore that it can update the signed values in the state store for the given channel.

```
struct GNUNET_PSYCSTORE_OperationHandle *
GNUNET_PSYCSTORE_state_hash_update
   (struct GNUNET_PSYCSTORE_Handle *h,
    const struct GNUNET_CRYPTO_EccPublicKey *channel_key,
    uint64_t message_id,
    const struct GNUNET_HashCode *hash,
    GNUNET_PSYCSTORE_ResultCallback rcb,
    void *rcb_cls);
```

Channel's public key: identifies the channel the signed state of which to update.

Message ID: specifies when the state hash was sent out. This message is flagged in the message store with a *state hash* flag to indicate that earlier messages are not necessary to apply further state modifiers.

Hash of the full state: used to verify integrity of the state. If it does not match the hash of the locally stored state, an error is returned and state synchronization has to be retried.

Result callback: used to return either a success or failure result of the hash update operation.

5.3.3 Retrieving state variables

For retrieving state variables, two functions are provided that have a similar signature to the state functions described in Section 4.8.2.

Retrieving all variables with a given *name prefix* happens through:

```
struct GNUNET_PSYCSTORE_OperationHandle *
GNUNET_PSYCSTORE_state_get_all
   (struct GNUNET_PSYCSTORE_Handle *h,
    const struct GNUNET_CRYPTO_EccPublicKey *channel_key,
    const char *name_prefix,
    GNUNET_PSYCSTORE_StateCallback cb,
    void *cb_cls);
```

While the best matching variable can be retrieved with:

```
struct GNUNET_PSYCSTORE_OperationHandle *
GNUNET_PSYCSTORE_state_get
   (struct GNUNET_PSYCSTORE_Handle *h,
    const struct GNUNET_CRYPTO_EccPublicKey *channel_key,
    const char *full_name,
    GNUNET_PSYCSTORE_StateCallback cb,
    void *cb_cls);
```

5.4 Membership store

Storing membership information — join and part events of a channel — is necessary to enforce access control on messages being relayed or replayed, and also needed for admission control to determine whether a join request came from an already admitted channel slave.

5.4.1 Storing membership

Membership change events of a channel are stored using the following PSYC-store API function:

```
struct GNUNET_PSYCSTORE_OperationHandle *
GNUNET_PSYCSTORE_membership_store
   (struct GNUNET_PSYCSTORE_Handle *h,
    const struct GNUNET_CRYPTO_EccPublicKey *channel_key,
    const struct GNUNET_CRYPTO_EccPublicKey *slave_key,
    int did_join,
    uint64_t announced_at,
    uint64_t effective_since,
    uint64_t group_generation,
    GNUNET_PSYCSTORE_ResultCallback rcb,
    void *rcb_cls);
```

Channel's public key: identifies the channel the event happened in.

Slave's public key: identifies the channel slave the event is about.

Did join? whether this is a join or part event.

Announced at: ID of the message that announced this join or part event. Used for ordering of events.

Effective since: message ID from which point the slave should be considered joined. It is less than or equal to the message ID the event was announced at: this allows granting access to message replays starting from the specified ID. This is used to implement different history access policies: when slaves have only access to messages they have seen after join, this equals to the message ID the join was *announced at*, while in a channel without history restrictions this would be set to 0 for joins to allow access to all messages sent to the channel. In addition to implementing policies, this can also be used for fine-grained access control on a per-slave basis. For part events, the only allowed value for this is 0, which makes sure other group members do not allow a previous member back in the group anymore. A removed member thus only can only contact the origin to ask for rejoin.

Group generation: in case of a part event, the last group generation the slave has access to. This can be used to stop the removed slave from accessing further fragments of a not yet completed message, as message fragments sent to the channel will have an incremented group generation after a slave has been removed from the channel.

Result callback: informs the caller about the result of the asynchronous storage operation.

The hash of the two public keys, the event type, and the two message IDs are then stored in the following structure:

```
CREATE TABLE membership (
  channel_id BLOB NOT NULL,
  slave_id BLOB NOT NULL,
  did_join INT NOT NULL,
  announced_at UNSIGNED INT NOT NULL,
  effective_since UNSIGNED INT NOT NULL,
  group_generation UNSIGNED INT NOT NULL
);
CREATE INDEX ON membership (channel_id, slave_id);
```

5.4.2 Testing membership

The stored membership information is used later to determine whether a particular slave was a member of the channel when the given message ID was sent to the channel. This is performed using the *membership test* API call:

```
struct GNUNET_PSYCSTORE_OperationHandle *
GNUNET_PSYCSTORE_membership_test
   (struct GNUNET_PSYCSTORE_Handle *h,
    const struct GNUNET_CRYPTO_EccPublicKey *channel_key,
    const struct GNUNET_CRYPTO_EccPublicKey *slave_key,
    uint64_t message_id,
    uint64_t group_generation,
    GNUNET_PSYCSTORE_ResultCallback rcb,
    void *rcb_cls);
```

Channel's public key: identifies the channel in question.

Slave's public key: identifies the channel slave whose membership to determine.

Message ID to check whether the given slave has access to.

Group generation: makes sure the membership information stored is still valid.

Result callback: called with the result of the membership test, which is either `GNUNET_YES`, `GNUNET_NO`, or `GNUNET_SYSERR` if it can not be determined.

To answer a membership test query, the PSYCstore first determines if its database is up-to-date enough to answer the query. If the *group generation* parameter is larger than the value PSYCstore knows about, then membership cannot be determined for sure, as there was a part event not yet known, which might have removed the slave in question.

Next, it looks up the last announced event for the slave that has an *effective since* ID less than or equal to the given message ID to test: if the resulting event is a join, the membership of the slave is confirmed, otherwise if this is a part or no such event is found, a negative result is returned.

This is expressed in SQL as follows.

```
SELECT did_join FROM membership
WHERE channel_id = ? AND slave_id = ? AND effective_since <= ?
ORDER BY announced_at DESC LIMIT 1;
```

6. Social service

This chapter describes the social service and its API, which implements the social network model of the system. The social service receives incoming messages for places from the PSYC service, processes the method calls using the try-and-slice algorithm, and notifies applications that have matching method handlers registered. It also acts upon certain method names itself that are related to membership management of places.

6.1 Social network model

Basic concepts used in the social network model are pseudonymous users and social places where pseudonyms can enter. We define the following terms to describe these:

Ego: one of the user's own pseudonyms. A user can have multiple pseudonyms, or alter egos.

Nym: a pseudonym of another user in the system.

Home: one of the user's own *places*, hosted by one of the user's *egos*.

Place: a place hosted by a *nym*, where an *ego* or other *nyms* can enter.

Host: the pseudonym who is the owner of a *place* or *home*.

Guest: a pseudonym present in a *place* or *home* (other than the host).

Figure 6.1.1 shows relationships in the social network.

6.2 Messaging in places

A place is managed by a pseudonym hosting it, and uses a one-to-many message distribution model: the host can announce messages to all guests present, which is used in case of (micro)blogging and news announcements. For many-to-many messaging use cases — e.g. discussion forums or chat

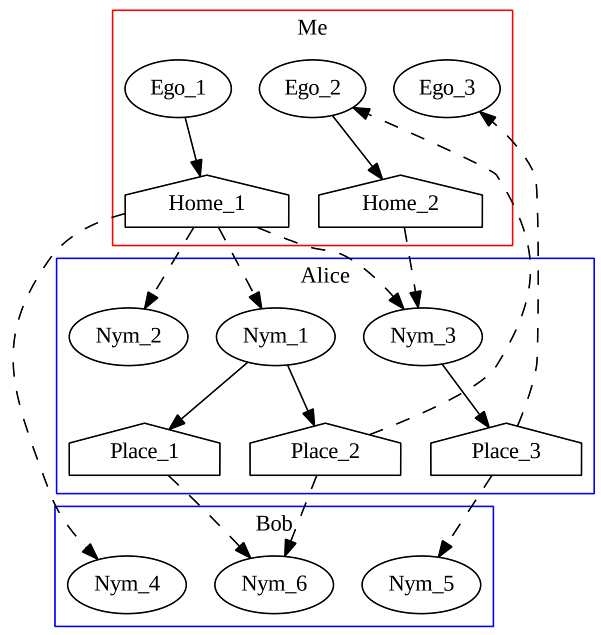

Figure 6.1.1: Relations in the social network from a user's perspective. *Solid line: host relationship; dashed line guest relationship. The arrows indicate the multicast message flow.*

rooms — guests publish content by asking the host to announce messages for them.

We distinguish between private and public places: in the former case the host controls admission to the place, and past messages are only visible to those who were present at the time they were sent, while public places do not have admission control and restrictions on accessing past message.

A place is modelled as PSYC channel, which in turn is implemented as a multicast group. This enables it scale to a large number of participants.

Each place has a persistent set of objects, which is used to model e.g. user profile fields, the list of guests present, or the topic of a chat room. The host can modify objects in the place by sending out operations to be performed on them to the guests. This is implemented using the decentralized state of the PSYC layer, which defines these operations.

Each message sent to a place has a method call associated with it, and optionally one or more transient variables or object modifying operations. The method calls are used as Remote Procedure Calls (RPC) between applications, each running on a different peer in the network. For this reason extensibility plays a key role here: the system should be able to evolve over time and gain new functionality, but in case of peer-to-peer systems it is not feasible to upgrade all the nodes at once, like it is possible for centralized systems. Due to this, backwards compatibility has to be maintained with peers running earlier versions of the software. To achieve this, messages use the PSYC syntax, which provides extensible method names that can be handled by nearest-matching method handlers, this is referred to as *try-and-slice* method processing.

The system uses a *push* model of distributing information: messages sent to a place are stored locally by each peer, which means that the message history and persistent objects in the place are available even when not connected to the network. This is a key property of the system contributing to its scalability and availability, as information modelled this way avoids costly request-response operations, and also frees the publisher from having to actively keep information online once it is published to the intended recipients.

This is in contrast with systems like LifeSocial [5] and MyZone [6], both of which employ a *pull* model, in which case information is retrieved on-demand from an online source: the former uses a DHT with encrypted profile entries, while the latter relies on mirrors to store user profiles, which have to be trusted to enforce access control on them.

6.3 Applications

Applications can be either interactive user interfaces, or bots running in the background and handling a certain task, such as relaying messages to guests in case of chat rooms, or automatically answering incoming entry requests.

Each application manages its own subscription list of homes and places, and receives method calls it registered handlers for. For instance a relay bot would only enter homes where chat messages need to be relayed, and handle `_message` methods. An admission bot would be used in homes to automate the entry process — e.g. auto-admit friends — and thus would only handle entry requests. User interfaces would typically enter multiple homes and places the user has subscribed to, and would implement method calls for displaying incoming messages.

Application creators can define their own method names they want to use, and implement various decentralized messaging applications based on the provided one-to-many distribution model. Various homes of a user can have different applications running in them, a few examples of what can be implemented this way:

(Micro)blogging: an application publishes content to subscribers. It needs a user interface for posting, and either a user interface or a bot for managing subscription requests.

Chat room: the host as well as all guests can send and receive messages. The host runs a relay bot that announces message requests coming from guests.

6.4 Social API overview

The social API provides access to functionality of the social service for applications. It defines operations on homes and places.

6.4.1 Home

Functions available for homes:

Enter home: an ego enters the home and starts hosting it.

Advertise home: publish a PLACE record in the GNS zone of the ego hosting the home.

Eject a guest out of the home.

Announce a message to all guests present.

Leave home.

Callbacks used in a home to notify about events:

Answer door: a guest wants to enter, decide either to *admit* or *reject entry.*

Farewell: notification about a guest leaving.

Message received from a guest, for registered method names.

6.4.2 Place

Functions provided for places are the following:

Enter place: an ego requests entry to a place.

Talk to the host.

Learn history: request historic messages of the place to be replayed, which are likely already available in the local PSYCstore.

Look at objects: examine the current value of an object.

Watch objects for change.

Leave place.

Callbacks used to notify about events in a place:

Method calls for methods with registered handlers.

Object changed in the place, when an object is watched.

Functionality related to history and objects can also be used by homes after converting a home handle to a place handle.

6.5 Using the social API

When an application starts up, it needs to load the private keys of the *egos* it is going to use in homes and places. Egos are managed by the identity service of GNUnet, which offers an API to create an ego (generate and store its private key), then later enumerate egos and retrieve their private keys.

Once the egos are loaded, an application enters the homes it wants to host and the places it wants to visit.

6.6 Entering a home

An application needs to enter a home before it can start sending and receiving method calls for it. The social API provides the following function to do so:

```
struct GNUNET_SOCIAL_Home *
GNUNET_SOCIAL_home_enter
   (const struct GNUNET_CONFIGURATION_Handle *cfg,
    const char *home_keyfile,
    enum GNUNET_PSYC_Policy policy,
    struct GNUNET_IDENTITY_Ego *ego,
    struct GNUNET_SOCIAL_Slicer *slicer,
    GNUNET_SOCIAL_AnswerDoorCallback listener_cb,
    GNUNET_SOCIAL_FarewellCallback farewell_cb,
    void *cls);
```

Home's private-public key pair: the public key identifies the place, while the private key is used to sign messages on the multicast layer. It is loaded from the provided key file.

Ego hosting the home. It is used to sign the ego's own messages sent to the home and to associate the home with the GNS zone of the ego.

Slicer for handling incoming messages from guests, it contains registered method handlers for try-and-slice processing, which is described in Section 6.10.2.

Policy: admission control and history access restrictions for the underlying PSYC channel. The available policies are discussed in Section 4.2.

Answer door callback: called when a guest requests entry to the home, and can be answered either by admitting the guest or rejecting entry.

Farewell callback: called to inform about a leaving guest.

Multiple applications can enter the same home simultaneously. When the first application enters the home, a PSYC channel, and subsequently a multicast group is created for it. These remain active until the home is left permanently. Section A.3 of the appendix shows the function calls and messages passed between components when a home is first entered. When a second application enters the same home, the social service can use the already started PSYC channel, and only needs to register the connecting application.

6.7 Advertising a home in GNS

Once an application has entered the home, it can publish a PLACE record in GNS using the following API call:

```
void
GNUNET_SOCIAL_home_advertise
   (struct GNUNET_SOCIAL_Home *home,
    const char *name,
    size_t peer_count,
    const struct GNUNET_PeerIdentity *peers,
    GNUNET_TIME_Relative expiration_time,
    const char *password);
```

Home: the PLACE record is published in the GNS zone of the ego hosting this home.

Name: the label used for the PLACE record.

Expiration time of the published record. A zero value can be used to remove an existing record.

Password: if provided, the published record is encrypted with this password — including its label, type, and data — providing a form of access control for private homes.

6.8 Entering a place

The place entry process consists of three phases: first a guest requests entry from the host of the place, then the host either decides to admit or reject entry, finally the host sends out a notification about the new guest to other guests present in the place. Refer to Section A.4 of the appendix for sequence diagrams of the entry process showing the full process across the layers.

6.8.1 Guest requests entry

Guests can request entry to a place by either providing its GNS address, or its public key and a list of peers used to join the underlying multicast group. The two variants of the place enter API function are the following:

```
struct GNUNET_SOCIAL_Place *
GNUNET_SOCIAL_place_enter
   (const struct GNUNET_CONFIGURATION_Handle *cfg,
    struct GNUNET_IDENTITY_Ego *ego,
    char *address,
    const char *method_name,
    const struct GNUNET_ENV_Environment *env,
    size_t data_size,
    const void *data,
    struct GNUNET_SOCIAL_Slicer *slicer);
```

```
struct GNUNET_SOCIAL_Place *
GNUNET_SOCIAL_place_enter2
   (const struct GNUNET_CONFIGURATION_Handle *cfg,
    struct GNUNET_IDENTITY_Ego *ego,
    struct GNUNET_CRYPTO_EccPublicKey *crypto_address,
    struct GNUNET_PeerIdentity *origin,
    size_t relay_count,
    struct GNUNET_PeerIdentity *relays,
    const char *method_name,
    struct GNUNET_SOCIAL_Slicer *slicer,
    const struct GNUNET_ENV_Environment *env,
    size_t data_size,
    const void *data);
```

Common function arguments:

Ego: pseudonym to use in the place. It is used by the multicast service to sign requests sent to the origin of the underlying multicast group — the host of the place on the social layer — thereby establishing a binding between the peer identity used on the multicast layer and the pseudonym used on the social layer.

Method name for the message sent along with the entry request.

Environment containing variables for the message.

Message body for the message.

Slicer with method name handlers the application wants to be notified of. Described in Section 6.10.2.

In the second variant, `place_enter2()`, the following arguments are used to specify the place to join:

Public key of the place to join.

Peer IDs of the origin and relays of the multicast group. Used by the multicast layer to send the join request to, as covered in Section 3.4.

In the first variant, `place_enter()`, the following argument is used instead:

GNS address of the place to join, which is looked up using the GNS service to retrieve the same information as given manually in the in `place_enter2()`. The GNS address can be specified in either the *place.nym.gads* form, if a human memorable name for the place is known, or in the *NYMPUBKEY.zkey* form, which is used to initiate contact with a nym in case only its public key is known.

After one of these functions are called, the social service sends a join request to the underlying PSYC channel, passing it the address details of the place, the public key of the ego, and the join message given by the application. The join process of the PSYC layer is described in Section 4.4.1.

6.8.2 Host answers the door

After the guest requested entry to the place, the host gets notified about it, unless the request was already approved by the PSYC layer below, which is the case when an already admitted guest reenters, or there is no admission control required by the underlying PSYC channel's policy.

The social API uses the following callback to notify connected applications about the guest requesting entry:

```
typedef void
(*GNUNET_SOCIAL_AnswerDoorCallback)
   (void *cls,
    struct GNUNET_SOCIAL_Nym *nym,
    const char *method_name,
    size_t variable_count,
    GNUNET_PSYC_Modifier *variables,
    size_t data_size,
    const void *data);
```

Nym: the identity of the guest requesting entry.

Method name, variables, data: parts of the join message sent by the guest.

One of the applications receiving the request — either a user interface or automated admission bot — would then answer the door either by admitting the guest, or rejecting entry.

To admit the guest, an application would call:

```
void
GNUNET_SOCIAL_home_admit
   (struct GNUNET_SOCIAL_Home *home,
   struct GNUNET_SOCIAL_Nym *nym);
```

Upon calling this function, the social service sends out a message to the place with a `_notice_place_enter` method call, informing about the public key of the *nym*, which is also the identifier of the channel slave on the PSYC layer. This message is used to update the local membership database of all slaves of the PSYC channel, in order to be able to enforce admission and history access restrictions — this is described in Section 4.4 in more detail.

An application can also decide to reject the entry request by calling the following function:

```
void
GNUNET_SOCIAL_home_reject_entry
   (struct GNUNET_SOCIAL_Home *home,
    struct GNUNET_SOCIAL_Nym *nym,
    const char *method_name,
    const struct GNUNET_ENV_Environment *env,
    size_t data_size,
    const void *data);
```

This function allows for returning a rejection message, which could e.g. specify the reason for rejection, or could redirect to another place.

6.9 Leaving a place or home

To leave a place or home, an application uses one of the following API functions:

```
void
GNUNET_SOCIAL_home_leave
   (struct GNUNET_SOCIAL_Home *home,
   int keep_active);
```

```
void
GNUNET_SOCIAL_place_leave
   (struct GNUNET_SOCIAL_Place *place,
    int keep_active);
```

In both cases the application gets disconnected from the social service, and other applications would still be able to use the place or home. The *keep active* flag has significance when the last application using the place or home leaves. If it is `GNUNET_YES`, the social service keeps the home or place active, still allowing the place to receive messages and store them in the PSYCstore, whereas if set to `GNUNET_NO`, it parts the underlying PSYC channel slave, or stops the channel master.

Later an application can enter the same home again, and continue announcing messages. When already admitted guests reconnect, they can enter without an application having to admit them again, as membership information is stored permanently in the PSYCstore.

A guest can ask the host to be removed from the membership database of the place entirely by sending a message with a method call requesting leave.

Section A.5 of the appendix shows sequence diagrams of the full place leave process across the layers.

6.10 Messages

This section discusses sending and receiving messages to homes and places. Refer to Section A.2 of the appendix for sequence diagrams of these operations.

6.10.1 Sending messages

The host of the home can announce messages to the guests present, while guests of places can only talk to the host directly. The functions provided by the API for these two message transmission operations are quite similar.

The host uses the following function to *announce* a message to guests present:

```
struct GNUNET_SOCIAL_Announcement *
GNUNET_SOCIAL_home_announce
   (struct GNUNET_SOCIAL_Home *home,
    const char *method_name,
    const struct GNUNET_ENV_Environment *env,
    GNUNET_SOCIAL_HomeTransmitNotify notify,
    void *notify_cls,
    GNUNET_SOCIAL_AnnouncementFlags flags);
```

While a guest can *talk* to the host using the following API call:

```
struct GNUNET_SOCIAL_TalkRequest *
GNUNET_SOCIAL_place_talk
   (struct GNUNET_SOCIAL_Place *place,
   const char *method_name,
   const struct GNUNET_ENV_Environment *env,
   GNUNET_SOCIAL_PlaceTransmitNotify notify,
   void *notify_cls,
   GNUNET_SOCIAL_TalkFlags flags);
```

In both cases the applications specify a *method name*, and an *environment*, which contains variables for the message. In case of a home, the environment can also contain operations on objects in the home.

When an application sends a message to a home, it adds a variable indicating the pseudonym the message is coming from. This is the ego for a host's own messages, and a nym for messages relayed from other guests.

After a period of inactivity without any applications sending a message to a home, the social service sends out a keep-alive message. This is necessary as a missing fragment can only be detected by a gap in the received fragment IDs on the multicast layer — the last fragment sent to the multicast group can not be noticed if missing.

6.10.2 Receiving messages

A slicer is used to register handlers for specific method names for the try-and-slice processing of incoming messages. It is used when a message is received for a home or place, or historic messages are replayed.

An application passes in a slicer when entering a home or a place, and can *add* and *remove* method handlers any time until the slicer is destroyed after leaving the home or place. The following two API functions are provided for adding and removing method handlers, respectively:

```
void
GNUNET_SOCIAL_slicer_add
   (struct GNUNET_SOCIAL_Slicer *slicer,
   const char *method_name,
   GNUNET_SOCIAL_Method method,
   void *method_cls);
```

```
void
GNUNET_SOCIAL_slicer_remove
   (struct GNUNET_SOCIAL_Slicer *slicer,
   const char *method_name,
   GNUNET_SOCIAL_Method method);
```

Upon an incoming method call, the try-and-slice processing of the method name is performed the following way: first the full method name is looked up in the list of registered method handlers in the slicer. If there's no match, the last keyword of the method name is removed and the shorter method name looked up. This is repeated until a matching method handler is found. A method handler with an empty method name matches everything. This allows handling more specific method names — that an earlier version of an application might not yet know about — with less specific method handlers registered in the slicer, which provides extensibility and backwards compatibility for method invocations.

Method handlers with a matching name registered in the slicer are then invoked, to notify the application about the incoming method call. A method handler is defined as follows:

```
typedef int
(*GNUNET_SOCIAL_Method)
   (void *cls,
    struct GNUNET_SOCIAL_Nym *nym,
    const char *full_method_name,
    uint64_t message_id,
    GNUNET_ENV_Environment *environment,
    uint64_t data_offset,
    size_t data_size,
    const void *data,
    enum GNUNET_PSYC_MessageFlags flags);
```

Nym: used only when the host of a home receives a method call from one of the guests. In this case it is the identity of the guest who sent the message. The fragments of this message were signed — and verified by the receiver — on the multicast layer with the key that belongs to this nym. For multicast messages it is not set, because those are signed by the host. For messages from a guest that are relayed by the host, the identity and signature of the sender can be added on the application layer in variables transmitted with the message.

Full method name: the full method name as specified in the message, which might be more specific than the method name this handler is registered for.

Message ID: uniquely identifies the message in the place.

Environment: contains transient variables and operations on persistent objects of the place. The values of transient variables can be used right away, while the result of an operation on an object is provided using a separate API described in Section 6.12.

Data offset: byte offset of the fragment of the data.

Data: fragment of the message body starting from the offset above.

Flags: indicate the first and last fragment of a message.

In case of a fragmented message, this method is called multiple times with further fragments of the data. As the data fragments do not contain the method name anymore, the message ID in each fragment is used to identify the method handler that this message has matched.

6.11 Learn the history of a place

When an application needs to access past messages of a place or home, it can request them through the following API call:

```
struct GNUNET_SOCIAL_HistoryLesson *
GNUNET_SOCIAL_place_get_history
   (struct GNUNET_SOCIAL_Place *place,
    uint64_t start_message_id,
    uint64_t end_message_id,
    const struct GNUNET_SOCIAL_Slicer *slicer,
    GNUNET_SOCIAL_FinishCallback finish_cb,
    void *finish_cb_cls);
```

This requests messages between the *start* and *end message ID* (inclusively) using the PSYC API — described in Section 4.8.1 — which retrieves the requested messages either from the PSYCstore, or requests replay of messages not in the message store from the multicast service. The retrieved messages are passed through the slicer of the place, with a *historic* flag set for messages retrieved from the PSYCstore.

This function can also be used to request history of a home after converting the home to a place handle. Section A.1.1 of the appendix shows a sequence diagram of the process.

6.12 Accessing objects in the place

An application can request certain objects in the place to be watched:

```
struct GNUNET_SOCIAL_WatchHandle *
GNUNET_SOCIAL_place_watch
   (struct GNUNET_SOCIAL_Place *place,
   const char *name_prefix,
   GNUNET_PSYC_StateCallback state_cb,
   void *state_cb_cls);
```

After calling this function, the values of objects with a matching *name prefix* are kept track of inside the library used by the application to access the social service. Whenever a method call that modifies a watched object is encountered, the modifier is applied to the in-memory value of the object, and the provided *state callback* is called to inform the application about the new value of the object.

To reduce memory usage, only the value of watched objects are kept track of. To retrieve the value of other objects in the place, the following two functions can be used, which retrieve the requested objects via the PSYC API described in Section 4.8.2.

To retrieve the values of all objects with a matching name prefix, an application would use the following function:

```
struct GNUNET_SOCIAL_LookHandle *
GNUNET_SOCIAL_place_look
   (struct GNUNET_SOCIAL_Place *place,
    const char *name_prefix,
    GNUNET_PSYC_StateCallback state_cb,
    void *state_cb_cls);
```

For retrieving the best matching (less specific) object, the following function call can be used

```
const void *
GNUNET_SOCIAL_place_look_at
   (struct GNUNET_SOCIAL_Place *place,
   const char *object_name,
   size_t *value_size);
```

7. Summary

The peer-to-peer messaging system presented here serves as the basis for building applications using the provided stateful multicast message distribution mechanism. Multicast message distribution allows the system to scale to a large number of participants, while the push model of disseminating information in combination with local storage increases the availability and reduces the network usage of the system compared to pull model. The local storage is used to store past messages and the decentralized state of a multicast group, a persistent set of key-value pairs, which can be used to implement e.g. user profiles.

Extensibility and backwards compatibility of the system is enabled by using the PSYC syntax for messages. An extensible RPC mechanism is provided using the PSYC syntax, where method calls are processed with the try-and-slice algorithm, which makes it possible to handle method names not yet known with more general method handlers.

The system offers end-to-end encrypted and authenticated communication between its participants. Members of a multicast group use an ECDHE exchange to establish ephemeral session keys and AES encryption to secure their connection to other group members. The system uses ECC keys to identify pseudonymous users and social places, and also to authenticate messages.

For mapping the cryptographical identifiers to human memorable names GNS is used, a decentralized name system. It allows users to manage their own zones, and offers a decentralized PKI that uses transitivity to make up for the lack of globally unique names.

Bibliography

[1] Gabor X Toth. *Secure Share. A framework for secure social interaction.* 2012. URL: http://tg-x.net/pub/secushare.pdf.

[2] Martin Schanzenbach. “Design and Implementation of a Censorship Resistant and Fully Decentralized Name System”. Master’s. Garching bei München: TU Munich, 2012, p. 116.

[3] *Zooko’s Triangle.* URL: https://en.wikipedia.org/wiki/Zooko's_triangle.

[4] Marc Stiegler. “An introduction to petname systems”. In: *Advances in Financial Cryptography* 2 (2005).

[5] 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.

[6] Alireza Mahdian et al. *Myzone: A next-generation online social network.* Tech. rep. Department of Computer Science, University of Colorado at Boulder, 2011.

Appendix A

Sequence diagrams

This appendix contains sequence diagrams that show API `function()` calls, and IPC `MESSAGE`s sent between the components.

Due to lack of space, the `GNUNET_`*`SERVICENAME_`* prefix is removed from the function names, where the *`SERVICENAME`* is determined by the library the arrow points to, or in case of callbacks from where the arrow originates from.

IPC messages are sent between a service and its library — which provides the API used to access the service — in this case the `GNUNET_MESSAGE_TYPE_`*`SERVICENAME_`* prefix is removed.

A.1 History and replay

A.1.1 Learning the history of a place

Figure A.1.1 shows the process of an application requesting historic messages for a place.

1. The application requests historic messages with a specific message ID range from the social service.
2. The social service relays this request to the PSYC service.
3. PSYC requests the messages from PSYCstore.
4. PSYCstore returns fragments found for the requested message IDs.
5. The fragments are parsed by PSYC and returned to the application, just like messages coming from the network, with an extra historic flag set.
6. If a requested message is not found in the PSYCstore, a replay request is issued for those message IDs.

A.1.2 Replaying multicast messages

Figure A.1.2 shows how a replay request from a multicast group member is handled.

1. The multicast service receives a replay request from another member.
2. The multicast API asks notifies the PSYC service about the replay request.
3. The PSYC service performs a membership test using the PSYCstore service, to determine if the member has access to the requested fragment.
4. If the membership test passes, the PSYC service retrieves the fragment from the PSYCstore, and passes it to the multicast service.
5. The multicast service responds to the replay request either with the requested fragment, or an error code.

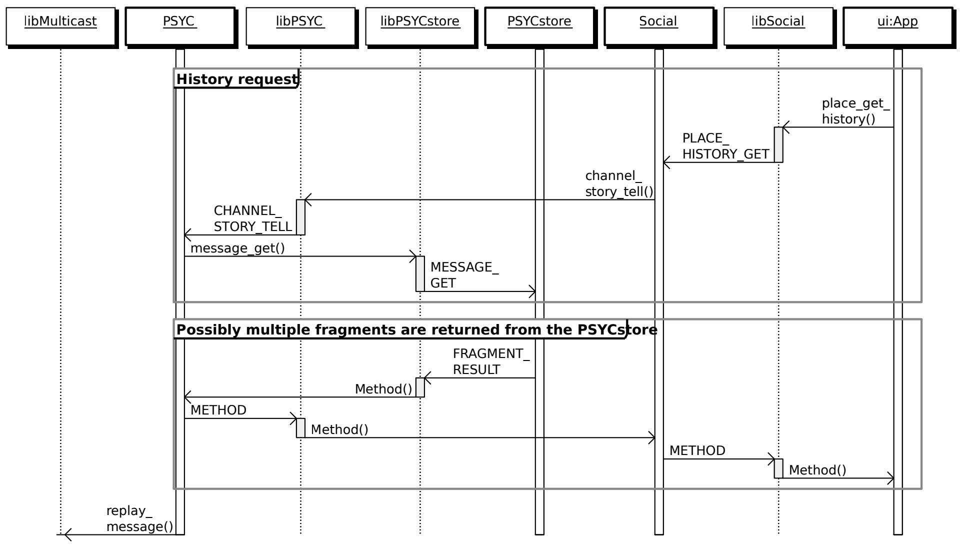

Figure A.1.1: An application requests historic messages of a place

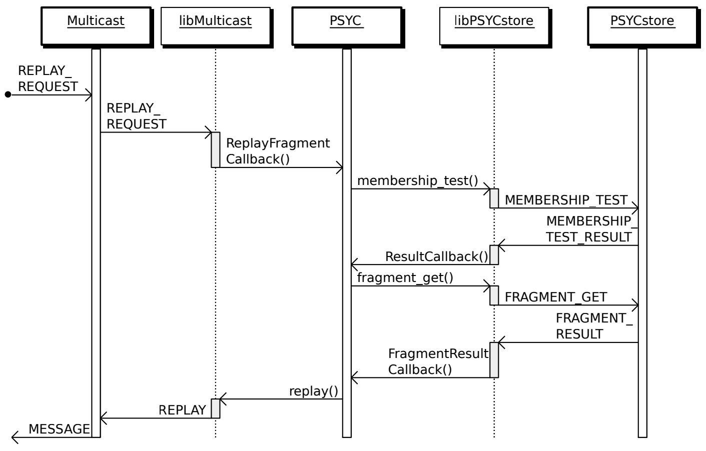

Figure A.1.2: Answering a multicast message replay request

A.2 Sending and receiving messages

A.2.1 Sending a message to a home

An application sends a message to a home the following way, as illustrated on Figure A.2.1.

1. The application announces the message to the home, by specifying its method name and optionally an environment that can set transient variables or modify objects of the home.
2. The application is notified when it can transmit the next part of the body, until the whole body is transferred. The first part of the message is sent to guests right away, with the rest of the body streamed to the network as pieces come in.
3. The message reaches the PSYC service through the social service.
4. The PSYC service constructs a PSYC-formatted message from the method name, modifiers, and first part of the body. It transmits this message using multicast, together with the message header fields determined by the PSYC service (message ID, group generation, state delta).
5. The PSYC service is notified when it can transmit the next part of the message.
6. Next, the message fragment containing the multicast message header is sent back to PSYC.
7. PSYC stores each fragment in the PSYCstore.
8. PSYC sends state modifiers to the PSYCstore to apply them to the current state.
9. The social service is notified about the method call, which then notifies applications with matching method handlers registered.

A.2.2 Receiving a message to a place

Messages sent to a place are distributed to all guests present, Figure A.2.2 shows the process of receiving a message.

1. Notify the PSYC service about an incoming message.
2. Store fragments and apply state modifiers using the PSYCstore.
3. Inform applications about the method call in the message.

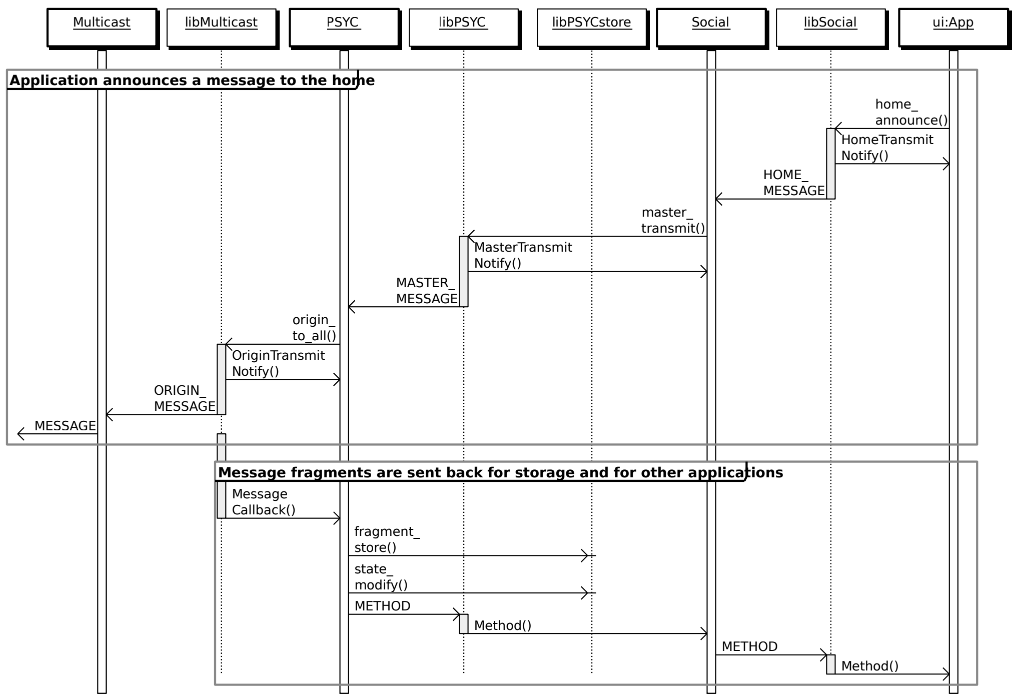

Figure A.2.1: Sending a message to a home

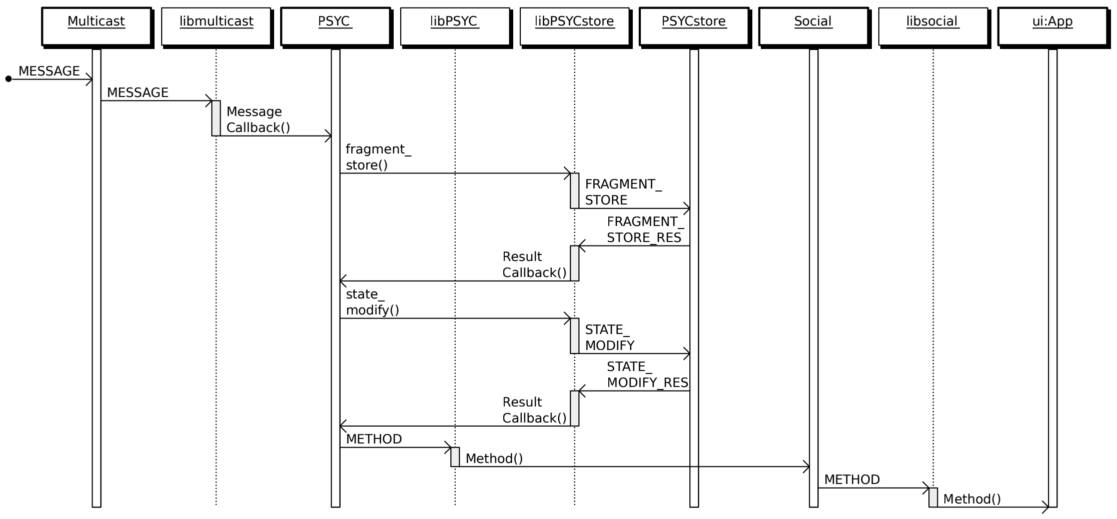

Figure A.2.2: Receiving a message to a place

A.3 Entering a home

Figure A.3.1 shows the process when the first application enters a home: the underlying PSYC channel and multicast group is started. When entering an already existing home, only the social service is contacted to register the application.

1. The first application enters the home.
2. The PSYC channel master is started.
3. PSYC requests the latest values of counters for this home, to continue numbering messages from in case of reentering an already existing home.
4. The multicast group's origin is started.
5. Finally, the application can advertise the home in GNS, if desired.

A.4 Entering a place

The place entry process consists of three phases: first a pseudonym requests entry from the host of the place, then the host either decides to admit or deny entry, finally the host sends out a notification about the new guest.

A.4.1 Guest requests entry

Sending a place entry request is illustrated on Figure A.4.1, and performed according to the following steps:

1. The application sends an entry request to the social service, which contains the address of the place, and any additional variables required for entry.
2. If a GNS address was provided, the social service looks it up using the GNS service.
3. Social requests the PSYC service to join the channel. A join message is passed along with this request with a `_request_place_enter` method call.
4. PSYC requests the counter values for this place from the PSYCstore, to determine the max. known state message ID for the channel.
5. PSYC requests the multicast service to join the multicast group.

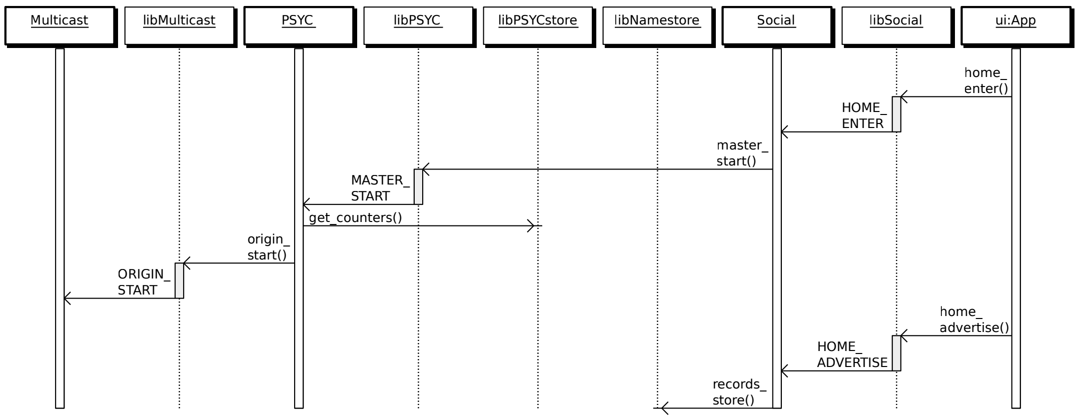

Figure A.3.1: Host enters a home and publishes it in GNS.

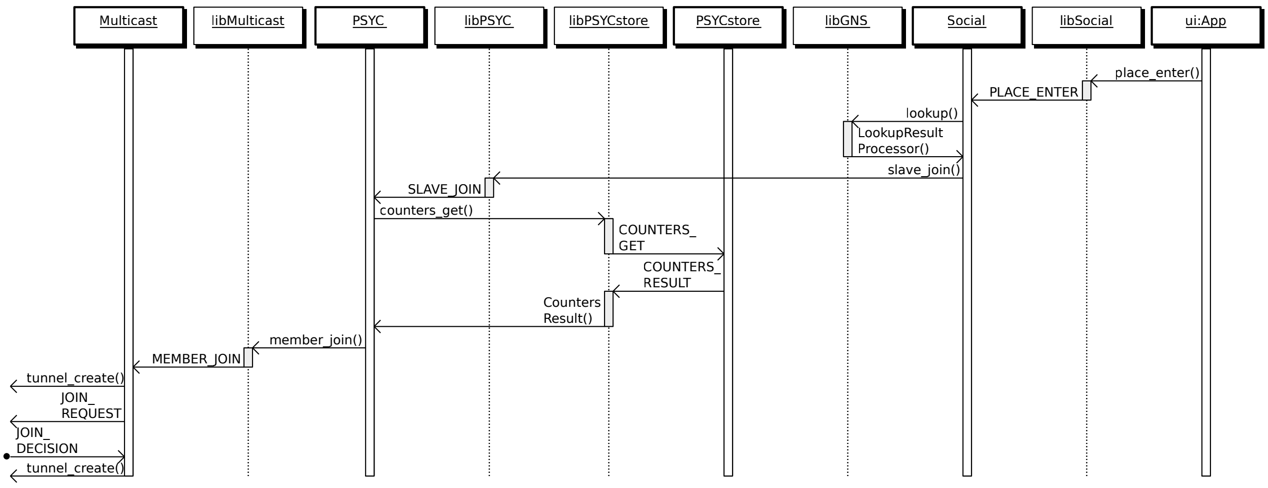

Figure A.4.1: Guest requests entry to a place.

6. Multicast tries to establish connection to the relays first, falling back to the origin if that fails, then sends the join request once the connection is established.
7. A join decision is returned in response, either admitting the peer or rejecting entry.
8. If the join decision contains other group members to connect to, a mesh connection is established to them, and the newly connected member starts receiving messages for the place.

A.4.2 Host receives entry request

Figure A.4.2 shows how an entry request is handled.

1. The origin of the multicast group receives an inbound mesh connection.
2. The first request is always a join request, and it reaches the application through join callbacks of the services in-between.
3. The application decides whether or not to admit the requesting nym.
4. If the nym is admitted, the social service sends out a message to the other guests with a `_notice_place_enter` method call, notifying them about the new guest. This message is stored in PSYCstore as well for later replay.
5. Next, the social service updates the membership information of the PSYC channel, which is saved in PSYCstore by marking the notification message sent in the previous step with a join flag. This information is used later for membership tests.
6. A join decision is sent back to the requesting peer, which might contain additional peer IDs of members to connect to.
7. The state of the joining member is brought up to date by sending the full state and/or state modifying messages the member does not yet have.
8. If the entry is rejected instead, a negative join decision is sent back to the requesting peer.

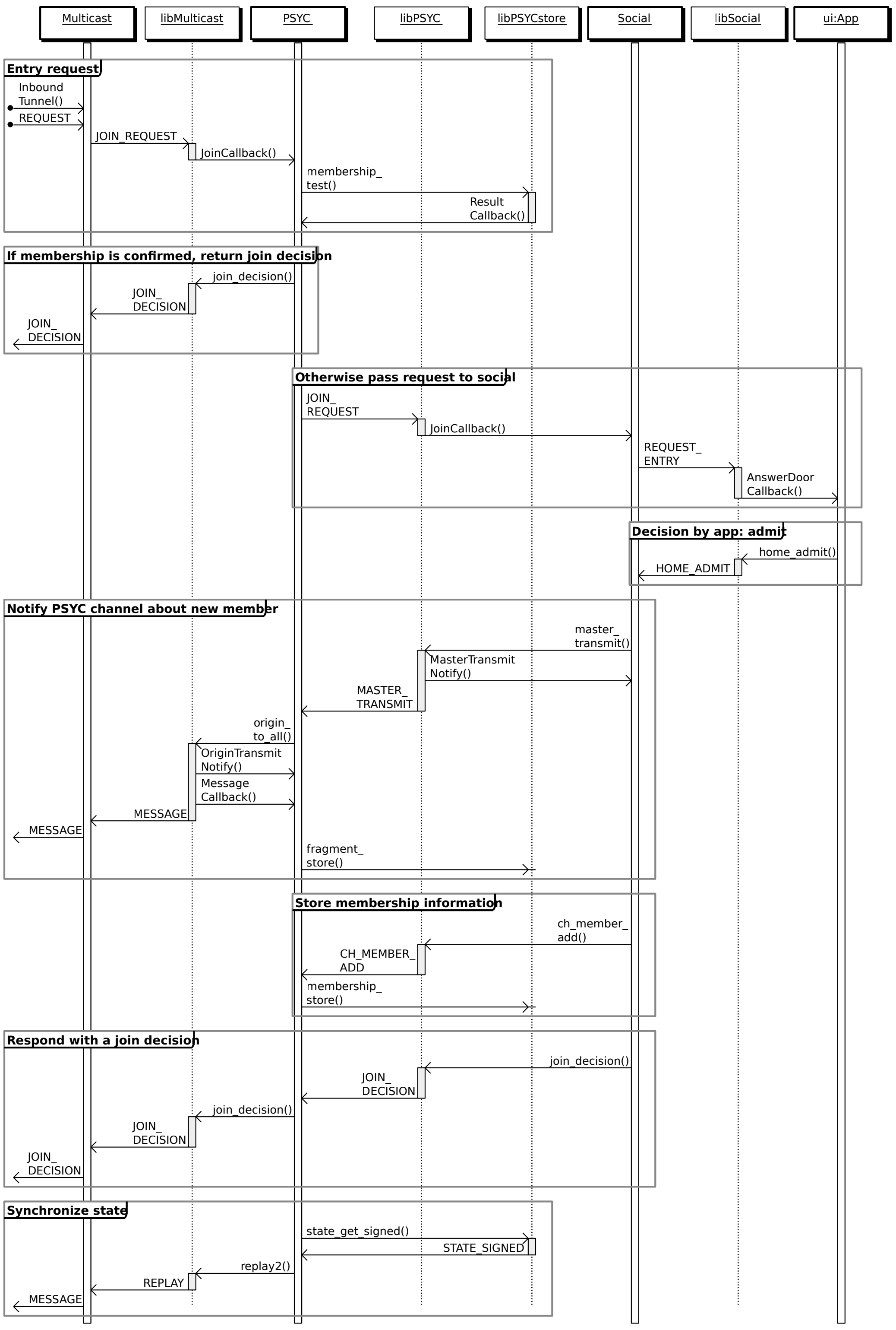

Figure A.4.2: Host receives an entry request to a home, then answers it.

A.4.3 Other guests receive entry notice

The message notifying about the new member is distributed to all members of the multicast group. Figure A.4.3 illustrates what happens when a member receives this message.

1. After the message arrived, it is stored using the PSYCstore, and the social service is notified about the `_notice_place_enter` method call.
2. The social service reacts to this method call by updating membership information of the PSYC channel, just like the host of the place did before. This way any member can answer membership test queries needed for joining and replaying messages.

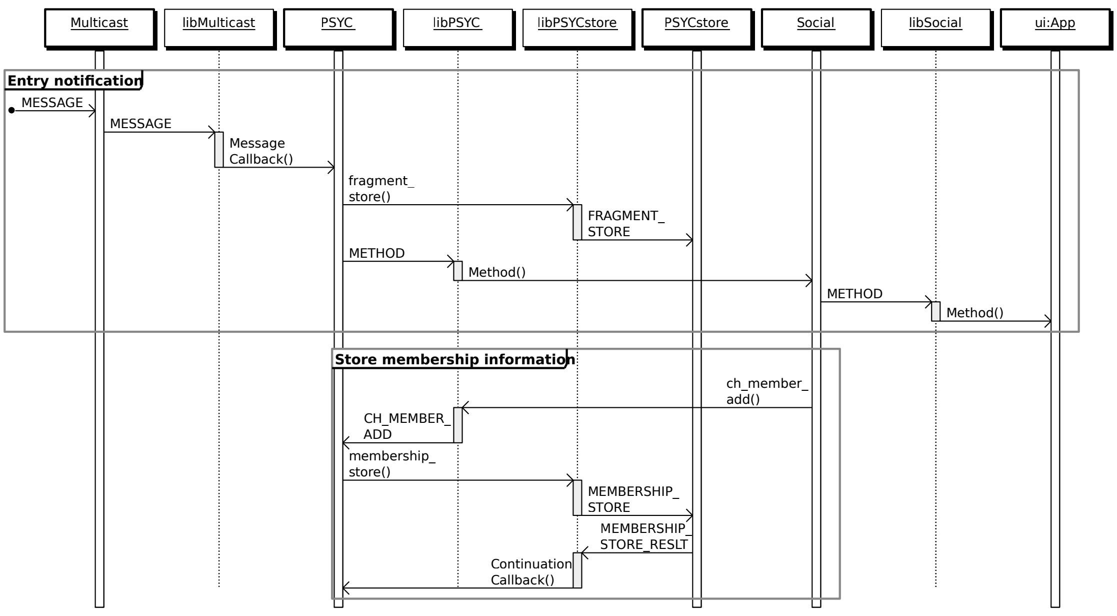

Figure A.4.3: Host announces the new guest.

A.5 Leaving a place

Leaving a place permanently entails the following: the guest intending to leave sends a leave request to the host of the place. When the host receives the request, it notifies all guests present about the leave event.

A.5.1 Guest sends leave request

Figure A.5.1 illustrates the process of a guest sending a leave request.

1. The application requests leave and disconnects.
2. The social service generates a `_request_place_leave` request and passes it to the PSYC service for sending it to the channel master
3. The request is transmitted through the multicast service to the origin of the multicast group.
4. Next, the social service parts the PSYC channel, the PSYC service parts the multicast group, and finally the multicast service destroys the mesh tunnel. If only this step is performed without sending the leave request, the guest can reconnect later without the host having to approve entry again, as channel membership doe not change in this case.

A.5.2 Host receives leave request

On Figure A.5.2 the process of a host receiving a leave request is shown.

1. The host of the place receives the part request. The social service recognizes the `_request_place_leave` method call, and also notifies the application about the leaving nym, in addition to the regular method callbacks.
2. The social service sends out a notification about the leaving nym to the PSYC channel. The message contains a `_notice_place_leave` method call, and the leaving nym in a variable.
3. Social tells the PSYC service about the leaving member.
4. PSYC saves the membership change with PSYCstore.

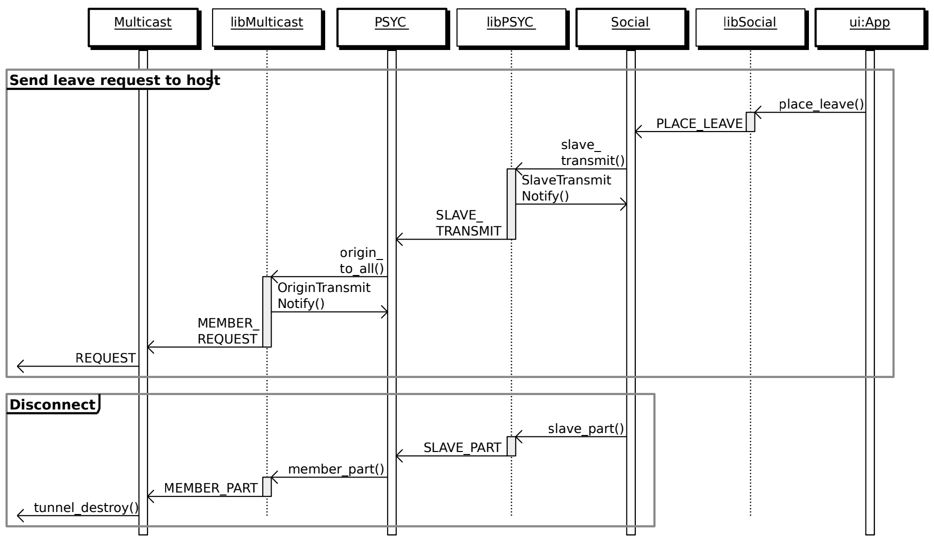

Figure A.5.1: Guest sends a leave request and parts.

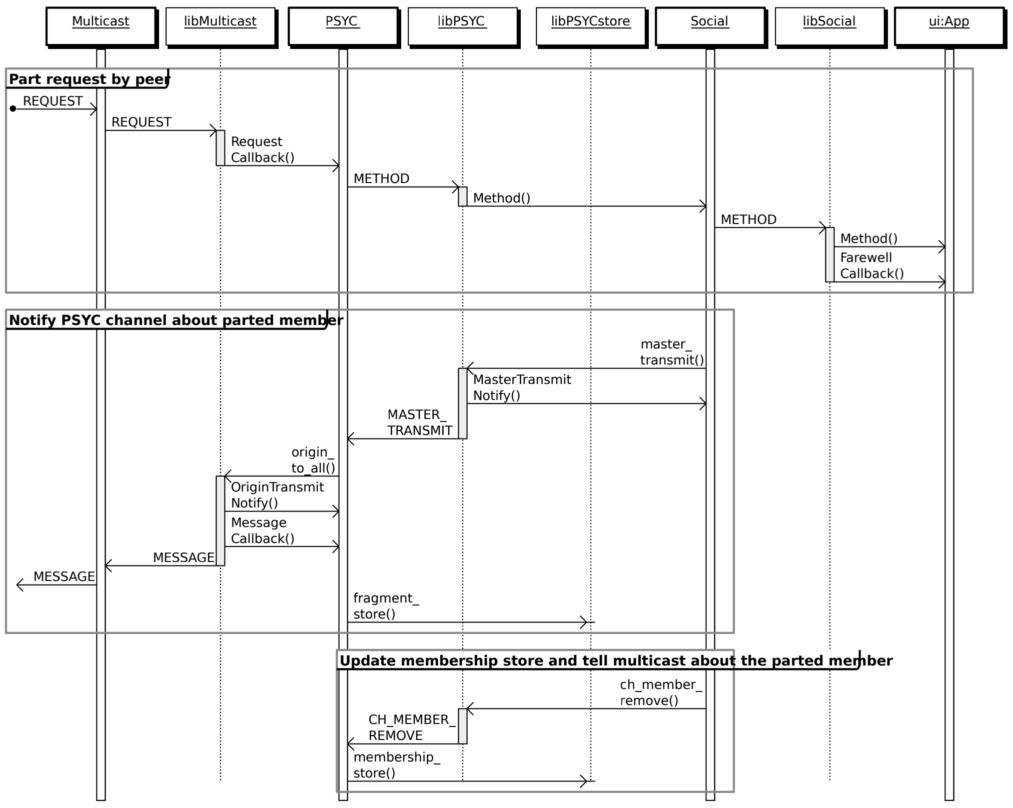

Figure A.5.2: Host receives a leave request and notifies guests present.

A.5.3 Other guests receive leave notice

Other guests present in the place receive the leave notice, as shown on Figure A.5.3.

1. A guest receives the multicast message containing the leave notice, and the social service is notified about the method call.
2. The social service reacts to the `_notice_place_leave` method call itself, and also notifies applications with registered method handlers.
3. Guests handle this method call similarly to the host: social tells the PSYC service about the leaving member, then PSYC updates the membership information in the PSYCstore.

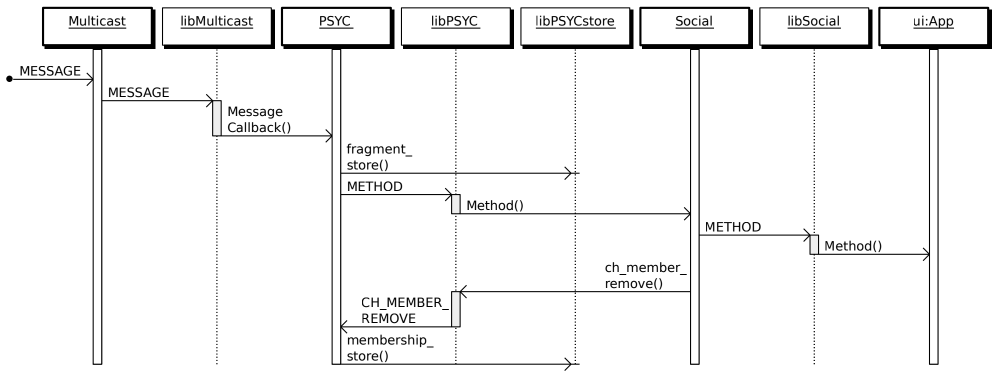

Figure A.5.3: Other guests receive a notice about the leaving guest.