(n+1)sec protocol specification

eQualit.ie

February 2, 2017

1 Introduction

The (n+1)sec system is a protocol for end-to-end encrypted and authenticated textual group chat communications. It aims to provide a platform and paradigm for synchronous text chat similar to those used in commonly used chat systems such as IRC [5] and Jabber multi user conversations [6], augmented with the cryptographic tools to guarantee communication security without relying on any third parties for these security assurances.

The (n+1)sec system, in conscious imitation of the celebrated off-the-record or *OTR* chat cryptography framework [2],

2 Protocol Overview

The (n+1)sec system is a protocol through which users of text chat systems, such as IRC [5] or Jabber multi-user-chat [6], can hold cryptographically secured multi-party text chat sessions. Using an approach similar to OTR [2], (n+1)sec clients exchange encoded text messages via a general-purpose group chat room, such as an IRC channel or a Jabber multi-user chat room, and thereby construct end-to-end encrypted chat sessions that use the general-purpose chat room as a carrier.

Chat sessions in (n+1)sec make use of end-to-end encryption to ensure the security of chat communications, even when the chat session is held in a public or otherwise presumed-insecure chat room. This cryptography is used to authenticate the identity of the members of the chat session, as well as to encrypt the chat communications in such a way that only the members of a chat session have access to the chat contents.

Chat in (n+1)sec takes place in so-called *conversations*. An (n+1)sec Conversation is a chat session, behaviorally similar to IRC channels and the like, that exists as a distributed agreement between a group of people in a carrier chat room to talk to each other, and whose communication takes place via (n+1)sec messages exchanged in a single carrier chat room. A Conversation at any point has a set of *participants*, which is part of the state of a Conversation that all participants agree on at any point. The different Participants in a Conversation have all authenticated each other's identities, as identified by a public key, and —some caveats notwithstanding— chat messages in a Conversation are encrypted in such a way that only its constituent Participants can decrypt.

All communications related to a Conversation are necessarily accompanied by a cryptographic signature based on the public key of the sender. Furthermore, all chat messages in a Conversation —making up the payload of that Conversation— are encrypted using a symmetric cryptographic key negotiated between, and known only by, the Conversation's Participants. This symmetric key is negotiated each time the list of Participants of a Conversation changes (that is, each time a Participant either joins or leaves the Conversation), as well as at regular intervals to reduce the risk of any particular key being compromised.

2.1 Rooms

The (n+1)sec protocol makes use of

Section 5

2.2 Conversations

This includes the description of how ephemeral public keys are used to identify conversation participants.

2.3 Conversation state machine

on a protocol level, np1sec conversations are represented by state machines represented identically across all clients that take part in the conversation. The np1sec protocol specifies in detail how messages and chat events are to affect this idealized state machine, which implementations need to follow to the letter to stay compatible.

2.4 Joining a conversation

2.5 Key agreement

2.6 Chat messages

3 Chat Model

maybe we can skip this?

4 Cryptography

The (n+1)sec protocol makes use of several different cryptographic techniques to implement the security properties outlined in the previous section. In addition to standard cryptographic constructions such as symmetric ciphers and public-key signatures, (n+1)sec uses versions of the *Triple Diffie-Hellman* deniable authentication scheme, and the Abdalla-Chevalier-Manulis-Pointcheval GKE+P authenticated group key exchange protocol.

In the remainder of this section, we introduce the nonstandard cryptographic techniques used in the (n+1)sec protocol. Section 4.3 describes and motivates the choice of standard cryptographic primitives used in the (n+1)sec protocol.

4.1 Triple Diffie-Hellman authenticated key exchange

In several places, the (n+1)sec protocol makes use of the Triple Diffie-Hellman deniable authenticated key exchange protocol [4]. Like the traditional Diffie-Hellman key exchange protocol, the Triple Diffie-Hellman protocol allows two parties to exchange a shared secret over a communication channel on which an eavesdropper may be present, which can be used as the basis of an ephemeral symmetric key which achieves perfect forward secrecy.

Unlike traditional Diffie-Hellman, the Triple Diffie-Hellman protocol provides authentication of the protocol participants, which makes the protocol secure in the face of men in the middle or other active attackers. This authentication component of the Triple Diffie-Hellman protocol is a form of *deniable* authentication: the messages exchanged in the protocol do not contain any artifacts that an attacker could use as cryptographic proof that communication between two partners occurred, which simpler schemes such as traditional Diffie-Hellman augmented with cryptographic signatures would.

The Triple Diffie-Hellman protocol yields a secret that is known only to the parties holding the private keys of the protocol participants. By communicating proof that the two parties know this secret, without necessarily using this secret as the base of a symmetric key, the Triple Diffie-Hellman protocol can also be used as a deniable authentication protocol, allowing two parties to authenticate each other under the allowance of deniability.

4.1.1 Notation

In the remainder of this section, we use G to denote a finite cyclic group of prime order N with generator g, and H to denote a cryptographic hash function. The Triple Diffie-Hellman protocol can be defined in terms of these parameter components, and the description below will use these components abstractly. The specific algorithms used to implement these cryptographic primitives in the (n+1)sec protocol are described in Section 4.3.

4.1.2 Protocol

The Triple Diffie-Hellman protocol takes place between two parties, denoted Alice and Bob, who each possess a private key and know each other's corresponding public key; these public keys are used as the basis of authentication. Alice and Bob's private keys take the form of natural numbers A and B, respectively, with $1 \leq A, B < N$, where N is the order of the group G. The corresponding public keys are the group values g^A and g^B, respectively.

When starting a Triple Diffie-Hellman session, Alice and Bob both generate ephemeral private keys, denoted a and b with $1 \leq a, b < N$, that are used for the duration of the session, and used to derive a shared secret that forms the basis of a symmetric key. As in traditional Diffie-Hellman, these ephemeral keys are not saved to long-term storage, and erased after the completion of a session, and this forms the basis of forward secrecy: the session secret is based (among others) on these ephemeral keys, which means that compromise of the long-term keys A and B will not jeopardize past session keys. Alice and Bob announce their respective ephemeral public keys, g^a and g^b.

Alice and Bob then compute the terms g^{Ab}, g^{aB}, and g^{ab}. These terms can be computed by Alice as $g^{Ab} = (g^b)^A$, $g^{aB} = (g^B)^a$, and $g^{ab} = (g^b)^a$, respectively; symmetrically, Bob can compute them as $g^{Ab} = (g^A)^b$, $g^{aB} = (g^a)^B$, and $g^{ab} = (g^a)^b$. Using these terms as input to a key derivation function, described below, yields a shared secret $S = f(g^{Ab}, g^{aB}, g^{ab})$.

A key property of the Triple Diffie-Hellman protocol is that the key derivation function uses the terms g^{Ab}, g^{aB}, and g^{ab}, but not g^{AB}. Because all terms

used by the key derivation function are based on at least one ephemeral key, an attacker who holds both ephemeral private keys – but neither of the long term private keys – is also in a position to compute the secret S. Indeed, an attacker knowing a, b, g^A, and g^B can compute $g^{Ab} = (g^A)^b$, $g^{aB} = (g^B)^a$, and $g^{ab} = ((g)^a)^b$.

This property has two critical consequences that (n+1)sec relies upon. A person who knows the public keys of Alice and Bob can generate their own ephemeral private keys a' and b', compute the assorted secret S, and thereby forge a convincing exchange between Alice and Bob that is indistinguishable for a third party to a genuine exchange between Alice and Bob. Because of this forgeability, a genuine exchange between Alice and Bob is useless to a third party as a cryptographic proof of communication between Alice and Bob, which forms the basis of deniable authentication (described below). Furthermore, the (n+1)sec protocol makes use of the possibility of computing a shared Triple Diffie-Hellman secret based on both ephemeral private keys as part of a denial-of-service recovery procedure, described in Section 4.2.2.

4.1.3 Secret computation

The Triple Diffie-Hellman protocol computes a secret based on the public keys and private keys of the two participants of the protocol. It does this by computing the terms g^{Ab}, g^{aB}, and g^{ab}, and using these terms as input to a key derivation function.

As described in Section 4.3, (n+1)sec uses the Ed25519 Twisted Edwards curve [3] as its cryptographic group G. The terms g^{Ab}, g^{aB}, and g^{ab} are therefore points (x, y) on that curve. To compute the shared secret $S = f(g^{Ab}, g^{aB}, g^{ab})$, (n+1)sec uses the following procedure:

1. Compute g^{Ab}, g^{aB}, g^{ab}.
2. For each of g^{Ab}, g^{aB}, g^{ab}, encode the point as a 32-byte stream encoding the x-coordinate in little endian encoding.
3. Sort the x-coordinate byte streams of the three points in lexicographical order, denoted x_1, x_2, x_3, with $x_1 \leq x_2 \leq x_3$.
4. Compute $S = H(x_1 \mathbin{+\!\!+} x_2 \mathbin{+\!\!+} x_3)$, where $\mathbin{+\!\!+}$ expresses concatenation.

The shared secret S for keys A, a, B, b is also denoted $TDH(g^A, g^a, g^B, g^b)$.

4.1.4 Authentication

The Triple Diffie-Hellman protocol exchanges a secret shared by two participants each identified by a pair of public keys. When one party then sends a message indicating knowledge of the secret, such as a message encrypted using the secret-derived symmetric key or a hash of the secret, this proves to the other party that the sender possesses the private keys on which the secret is based. Using this scheme, the Triple Diffie-Hellman protocol can be used as a deniable authentication system.

If Alice and Bob exchange a secret S, based on Alice's public keys g^A and g^a and Bob's public keys g^B and g^b, and Bob then sends a message $m(S)$ derived from S that could not have been derived from any message sent by Alice, this proves to Alice that Bob possesses both the private keys B and b. Bob can then use his ephemeral private key b to generate *deniable signatures* of further messages. When Alice receives such a message signed using the b private key, she can verify that the message was sent by Bob (for she knows that Bob possesses both B and b). But Alice cannot prove this fact to a third party, for as far as the third party is concerned Alice could possess both a and b, and have forged the b-based signature as well as $m(S)$.

The (n+1)sec protocol implements this scheme using the following authentication challenge protocol:

Participants. User A with username U_A has announced long term public key g^A and emphemeral public key g^a. User B with username U_B has announced g^B and g^b.

Round 1. User A sends user B an *authentication challenge* nonce N.

Round 2. User B sends user A the *authentication confirmation* $T = H(U_B \mathbin{+\!\!+} N \mathbin{+\!\!+} TDH(g^A, g^a, g^B, g^b))$.

Computation. User A verifies the correctness of the authentication confirmation. If it is correct, A knows that B possesses the private key b.

The (n+1)sec protocol uses the authentication challenge protocol in several places to provide mutual deniable authentication between users, in such a way that authentication of B to A and authentication of A to B typically run in parallel. This takes the form of a message from a user with username U_A, long term public key g^A, and ephemeral public key g^a, to a user with username U_B, long term public key g^B, and ephemeral public key g^b, containing an authentication challenge N; followed by a message from B to A containing the authentication confirmation $T = H(U_B \mathbin{+\!\!+} N \mathbin{+\!\!+} TDH(g^A, g^a, g^B, g^b))$. When A receives that message with a correct value of T, A can then mark B as being authenticated for the ephemeral public key g^b, and accept messages from B signed against that public key.

4.2 Group Key Exchange

Chat messages in (n+1)sec sent to a conversation are encrypted using a symmetric key known to all participants of the conversation. The participants of a conversation negotiate such a key each time a new participant joins the conversation, or a previous participant leaves the conversation.

To exchange such a symmetric key among a group of participants, the (n+1)sec protocol makes use of a *group key exchange* protocol. A group key protocol is an extension of the traditional Diffie-Hellman key exchange protocol; where the Diffie-Hellman protocol allows two parties to exchange a shared secret over an untrusted communication channel to be used to derive a symmetric key, a group key exchange protocol can achieve the same thing for larger collections of users. The (n+1)sec protocol invokes an instance of a group key exchange protocol each time a conversation requires a new shared symmetric key.

A complication of group key exchange protocols that has no equivalent in the two-party Diffie-Hellman key exchange protocol lies in the possibility of *denial of service* attacks. A participant in a group key exchange protocol might send messages containing invalid contribitions to the key exchange, in a way that other participants cannot readily detect, causing the key exchange mechanism to fail. While most group key exchange protocols will verify at the end that a key was exchanged successfully, and notice the problem if one of the participants to the exchange acted maliciously, most key exchange protocols cannot then determine which participant was responsible for the failure. Should the participants of the conversation then simply try again to exchange a key, the malicious participant could keep disrupting the negotiation of a key indefinitely, freezing conversation progress without anyone being able to determine the party responsible for this breakdown.

The (n+1)sec protocol makes use of a version of the Abdalla-Chevalier-Manulis-Pointcheval GKE+P authenticated group key exchange [1]. The version of this protocol used by (n+1)sec is modified in such a way that both the participants of a key exchange, and nonparticipating observants of a key exchange, can always determine the party responsible for any failures of the key exchange; these participants can then choose not to include the responsible party in further key exchange attempts, and solve the problem thereby.

4.2.1 Cryptography

The (n+1)sec group key exchange protocol takes place when a group of conversation participants decide that they need a new shared key between themselves. This happens only when all participants have already authenticated each other, and each participant has a signing key accepted as genuine by all participants.

All communications involved in the group key exchange protocol are signed using the author's such signing key; this aspect of the protocol ensures authentication of all participants involved in a group key exchange procedure.

In this context, the group key exchange protocol has the following conceptual steps:

1. All participants in the key exchange are ordered in a circle. Participants are denoted U_0 to U_{n-1}, with indices taken modulo n: $U_n = U_0$, $U_{-1} = U_{n-1}$. Participant U_i has public key $k_i = g^{m_i}$.
2. Each participant U_i generates a temporary private key x_i, used for the duration of the key exchange, and broadcasts the associated public key $y_i = g^{x_i}$.
3. Each participant U_i computes a Triple Diffie-Hellman secret shared with his left and right neighbours in the circle: $d_{i-1,i} = TDH(k_{i-1}, y_{i-1}, k_i, y_i)$; $d_{i,i+1} = TDH(k_i, y_i, k_{i+1}, y_{i+1})$.
4. Each participant U_i computes and broadcasts the XOR sum of their two Triple Diffie-Hellman secrets: $z_i = d_{i-1,i} \oplus d_{i,i+1}$.
5. By combining the linear combinations z of the secrets d, each participant computes the secrets $d_{j,j+1}$ for all $0 \leq j < n$. Each participant U_i computes these values by computing $d_{i+1,i+2} := z_{i+1} \oplus d_{i,i+1}$, $d_{i+2,i+3} := z_{i+2} \oplus d_{i+1,i+2}$, ...
6. Each participant U_i computes the shared secret $S = d_{0,1} \mathbin{+\!\!+} d_{1,2} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} d_{n-2,n-1} \mathbin{+\!\!+} d_{n-1,0}$. This secret is used as input to a key derivation function.

This protocol derives its security from the fact that the Triple Diffie-Hellman secrets $d_{j,j+1}$ can be recovered by the participants of the exchange, but not by any eavesdropper. The different values of z_j form a system of linear equations in the variables $d_{j,j+1}$; the protocol publishes the information that $d_{0,1} \oplus d_{1,2} = z_1$, $d_{1,2} \oplus d_{2,3} = z_2$, and so on. Of these n equations, $n-1$ are independent; the last one can be derived from the first $n-1$, for $z_0 = d_{n-1,0} \oplus d_{0,1} = \bigoplus_{i=1,n-1} z_i$. This makes the values of z a system of $n-1$ independent linear equations in n variables, which provides no usable information to an eavesdropper. Only by knowing the value of at least one secret d are participants able to compute the values of all other secrets $d_{j,j+1}$, for this gives the equation system a unique solution.

When the cryptographic protocol above is finished, the participants confirm to each other that they have all computed the same value of S. If this is not the case, this indicates that at least one of the participants U_i is malicious, and has broadcast a value of z_i that is not derived correctly from the public key

y_i they announced. In this situation, the participants of the protocol can all reveal their private key x_i; this allows all participants (and nonparticipating observers) to compute the correct values y_j and z_j that all participants should have announced. By comparing these expected values to the values actually broadcast, participants and observers can identify the malicious participant, and start a new group key exchange without that participant.

4.2.2 Protocol

The (n+1)sec protocol implements a concrete version of the abstract protocol described in the previous section. This protocol gets invoked when the set of chat-eligible members of a conversation changes, for reasons such as members joining and leaving the conversation. It also gets invoked when an existing group key has been in use for some time; by replacing a group key after a finite time limit, this limits the risk of the compromise of a symmetric group key, and thereby ensures that the compromise of any short-term keys compromises only a small fragment of long-running conversations. The details of the invocation of the group key exchange protocol are described from Section 6 onwards.

The group key exchange protocol implemented by (n+1)sec consists of a maximum of four phases. After completing the two cryptographic phases described in Section 4.2.1, a third phase takes place in which the participants of the exchange verify that they have all computed the same shared secret. If this is indeed the case, the exchange finishes with a successful completion. If not, a fourth phase starts in which all participants reveal their temporary private keys, facilitating denial-of-service recovery.

Concretely, (n+1)sec implements the following protocol:

Participants. The protocol initiates for a set of participants v, each having a username U_v and a long term public key k_v known by all participants. Participants are sorted in lexicographical order by username, and denoted U_0 to U_{n-1}.

Round 1. Each participant U_i generates a random private key x_i and public key $y_i = g^{x_i}$, and broadcasts y_i.

Round 2. Each participant U_i proceeds as follows:

- compute $\texttt{groupid} := H(U_0 \mathbin{+\!\!+} k_0 \mathbin{+\!\!+} y_0 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} U_{n-1} \mathbin{+\!\!+} k_{n-1} \mathbin{+\!\!+} y_{n-1})$;
- compute $d_{i-1,i} = H(TDH(k_{i-1}, y_{i-1}, k_i, y_i) \mathbin{+\!\!+} \texttt{groupid})$ and $d_{i,i+1} = H(TDH(k_i, y_i, k_{i+1}, y_{i+1}) \mathbin{+\!\!+} \texttt{groupid})$;
- compute $z_i = d_{i-1,i} \oplus d_{i,i+1}$;
- broadcast $(z_i, \texttt{groupid})$.

Round 3. Each participant U_i proceeds as follows:

- verify that all participants have sent the correct `groupid`. If not, abort, and mark all participants that sent an invalid `groupid` as

malicious;
- compute $d_{j,j+1}$ for all $0 \leq j < n$, by computing $d_{j,j+1} := z_j \oplus d_{j-1,j}$;
- compute $S := H(d_{0,1} \mathbin{+\!\!+} d_{1,2} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} d_{n-2,n-1} \mathbin{+\!\!+} d_{n-1,0})$;
- broadcast $H(S \mathbin{+\!\!+} \texttt{groupid})$.

Round 4. Each participant U_i proceeds as follows:
- verify that all participants have sent the correct $H(S \mathbin{+\!\!+} \texttt{groupid})$. If so, accept S as the exchanged key, and finish.
- if not, broadcast the private key x_i.

Aftermath. If participants disagreed on the value of $H(S \mathbin{+\!\!+} \texttt{groupid})$, and the private keys x_j were broadcast, each participant computes the correct values of y_j, z_j, and $H(S \mathbin{+\!\!+} \texttt{groupid})$ for each participant; marks all participants that broadcast invalid values as malicious; and aborts.

When this protocol completes, the participants have either accepted a group key, or have aborted having marked a nonempty set of participants as malicious. Participants can then start exchanging messages encrypted using this group key; or, in the case of unsuccessful abortion, can remove the malicious participants from the conversation and start a new invocation of the group key exchange protocol for the reduced set of participants. The details of this response are described in Sections 6 and 7.

4.3 Cryptographic primitives

The (n+1)sec system makes use of certain cryptographic primitives to achieve its security properties. In particilar, the (n+1)sec protocol uses a cryptographic hash function; a digital signature scheme; and a symmetric cipher. Additionally, the (n+1)sec protocol makes use of a secure finite cyclic group in which the Diffie-Hellman protocol can be performed.

The (n+1)sec protocol uses the following algorithms as implementations of these abstract primitives:

Cryptographic hash. (n+1)sec uses the SHA256 cryptographic hash algorithm as its hash function and random oracle. SHA256 provides a sufficiently secure hash primitive for the level of security provided by (n+1)sec and is widely implemented.

Signature scheme. (n+1)sec uses the Ed25519 elliptic curve signature scheme as its signature primitive. The Ed25519 scheme has been chosen as the signature primitive due to its efficiency and more secure implementability over other elliptic-curve digital signature algorithms.

Cyclic group. (n+1)sec uses the Curve25519 elliptic curve as its finite cyclic group in which to perform Diffie-Hellman. This choice makes it possible to use public keys for both signatures and Diffie-Hellman computations interchangeably.

Symmetric cipher. When encrypting messages sent to a conversation using a symmetric key known to the members of that conversation, (n+1)sec uses the AES-256 algorithm in Galois/Counter Mode (GCM) as its symmetric cipher. The (n+1)sec protocol follows the suggestion by the original OTR protocol [2] of using counter mode.

5 Carrier Chatrooms

exact carrier chatroom model

carrier limitations (based on high level API)

invitable user list

6 The Conversation State Machine

The key abstraction of the (n+1)sec protocol is the *conversation*, which is the context in which chats take place. An (n+1)sec conversation is an agreement between a group of users to exchange encrypted chat in a setting that behaves like a conventional chatroom, by exchanging (n+1)sec messages in a single carrier chat room. As such, a conversation has such things as a participant list, an ordered sequence of chat messages and related events, and indications of when participants join and leave the conversation. For the purposes of security, conversations also keep track of what users are able to receive which chat messages at each point.

Conversations are a *distributed* notion. When a group of users participating in a conversation, this fact is not recorded on the chat server, or in any other way regulated by the carrier chat infrastructure; instead, conversations exist only in the memory of the clients of their participants, and they rely on distributed coordination to keep the abstraction intact. For such a distributed coordinated abstraction to behave like a conventional chatroom, it is critical that its participants are at all times in agreement about such things as the member list, the chat transcript, and the state and history of the conversation in general.

Achieving and maintaining this agreement is a challenging affair. Because communication over the channel provioded by the carrier chat room is not instantaneous —messages take a nonzero time to arrive at the carrier chat room after being sent by a user, and then take another nonzero time before being received by all other users— multiple processes involved in maintaining a conversation might happen simultaneously in an interleaved fashion. For example, a new user might attempt to join a conversation at the unfortunate time when a group key

exchange procedure (as described in Section 4.2) is in progress. For another example, a conversation participant might stop responding due to connectivity problems while the process to accept a new user into the conversation is in progress, which should not cause the user-acceptance procedure to freeze indefinitely while the troubled participant fails to participate. Resolving these potential conflicts in a way that reliably results in agreement between conversation participants about what happened and in what order is a famously difficult problem. This problem becomes more difficult still if a conversation might contain malicious participants, that seek to sow confusion and prevent participants from reaching agreement, and thereby make coherent conversation impossible.

The (n+1)sec protocol approaches this problem using the following structure:

- Conversations have a formally specified state, referred to as the *conversation state machine*, of which all conversation participants maintain a copy. Different participants of a conversation maintain identical copies of this state machine at all times.
- The (n+1)sec protocol specifies for each chat message sent in a carrier chat room how this affects any conversations taking place in that chat room. Clients of conversation participants implement that specification precisely.
- Because different users in a carrier chat room receive the same messages in that room in the same order, different conversation participants perform the same modifications to their copies of the conversation state machine. Thus, when all participants in a conversation have finished processing all carrier chat room messages up to a certain point, they all maintain identical copies of the conversation state machine.
- All messages that a conversation participant can send that affect a conversation are designed in such a way that the visible interpretation of the message remains intact, no matter what intervening events may have occurred between the time that the sender sent the message and the time other participants receive it. For example, if a new group key exchange procedure finishes while a participant is sending a chat message, the design of both the chat message process and the key exchange process is such that the chat message is still received and interpreted as desired, and only by the expected recipients.

The remainder of this section describes the detailed procedure involved in participating in a conversation and maintaining the assorted state machine; the procedure of joining a conversation; and the exact content and semantics of the conversation state machine. An overview of the messages of (n+1)sec, and their specified consequences for the state machines of any conversations that might be affected, is described in Section 7.

6.1 Participating in a conversation

The (n+1)sec conversation state machine specification defines an *ideal abstract computation*. It defines how an ideal chunk of conceptual state evolves in response to a sequence of messages in a carrier chat room, and the semantics of these changes in state. As such, the (n+1)sec conversation state machine specification can be interpreted as a definition of a function from an initial state, and a sequence of carrier chat room messages, to a resulting state. With an interpretation of this state and modifications of this state as events in a conversation, this specification defines an ideal abstract interpretation of a sequence of carrier room messages as a sequence of conversation events.

An (n+1)sec client can *passively participate* in a conversation by implementing this abstract computation. By acquiring a copy of the current conversation state machine contents (described in Section 6.2) and then interpreting carrier chat room messages according to the specification of the conversation state machine, the client can keep exact track of the state machine contents, as well as most events happening in the conversation. A passive participant of a conversation can not decrypt chat messages sent to the conversation, as these chat messages are encrypted using a key that the passive participant does not know; but the passive participant can keep track of the list of members of the conversation and their status, as well as all other details of the conversation other than the chat message content. In particular, a passive participant can maintain a complete copy of the state machine contents specified by the abstract computation, despite not being able to decrypt any chat messages. The joining procedure, described in Section 6.2, relies on this ability, for a user who is invited to but not yet part of a conversation should be able to keep track of the conversation's list of members when deciding whether or not to join the conversation.

An (n+1)sec client can *actively participate* in a conversation by keeping track of the conversation state machine as above, while also sending messages to the conversation when the conversation allows or requires them to. When a user is invited to a conversation, that conversation's state machine changes in such a way that it will respond to messages by the invited user. By responding to that situation, the invited user can —all going well— eventually become a proper member, participate in key exchanges, send chat messages to the conversation, and decrypt messages sent by others.

By implementing a conversation state machine's abstract computation and maintaining identical copies of the conversation state machine, participants in a conversation can remain in agreement about the status and events of a conversation, and thereby achieve the distributed abstraction of a chatroom that behaves in a conventional way. This agreement, however, relies on the different participants implementing the abstract computation *exactly*; if different participants of a conversation interpret a message in such a way that they end up representing a *slightly* different version of the conversation state machine, the distributed

abstraction of a coherent chatroom will unravel in short order, making further coherent chat impossible. For example, if a carrier chat room message by user U is interpreted by one participant A as valid and processed, while being interpreted by another participant B as invalid and ignored, user B may eventually remove user U from the conversation because of a perceived failure to participate in some process of which the ignored message was part. If A then judges that U is still a member in good standing, while B decides that U is no longer a member of the conversation and initiates a group key exchange procedure that does not include U, coherent chat in this conversation cannot continue.

To avoid this contingency, members of a conversation regularly publish a checksum of the contents of the conversation state machine, as maintained by them. When receiving such a message, all other members compare this checksum against the checksum of their own copy; when these values are not equal, this indicates that the views of the conversation between the different members have diverged.

When such a divergence is detected, those members that notice that other members have divergent conversation state machine contents remove the diverging users from the contents of their copies of the conversation state machine, and announce that fact. After receiving that announcement, the victims of this divergement in turn remove the announcers from their own copies of the conversation state machine, and announce this as well. After several such messages, the incoherent conversation has split itself into two or more parts, with each resulting conversation consisting of all members of the original conversation that still have consistent state machines among themselves. In effect, the incoherent conversation has then split itself into two or more internally coherent successor conversations.

6.2 Joining a conversation

Conversations in (n+1)sec can be joined only after being invited into the conversation by one of the existing participants of the conversation. When this happens, the invited user performs a procedure to acquire a copy of the conversation state machine, and becomes a passive participant of the conversation. When this procedure finishes, the invited user can determine and keep track of the members of the conversation, and decide whether or not to accept the invitation based on that information. If the invited user wishes to accept the invitation and join the conversation as an active participant, they can announce this fact by sending a message to the conversation, and thereby start the process that —all going well— will lead to them becoming a proper participant with the ability to take part in chat.

To allow the invitee to acquire an up-to-date copy of the conversation state machine, the invitee and the inviter follow the following protocol:

1. The inviter sends a message containing an invitation for the invitee to join the conversation.

2. The inviter waits until they receive this invitation message back from the carrier chat room. When they do, the inviter sends a message containing the contents of the conversation state machine *after processing the invitation message*.

3. The invitee records all carrier chat room messages received after the invitation message, until they receive the message containing the conversation state machine contents.

4. When the invitee receives the message containing the conversation state machine contents, they construct a local copy of the state machine based on these contents. They then process all recorded messages, from but excluding the invitation message, up to and including the conversation state machine contents message, in the context of the newly constructed state machine copy.

5. After having processed the message containing the conversation state machine contents, the invitee has a copy of the same conversation state machine as the existing members of the conversation, and in particular the inviter. The invitee can now process further carrier chat room messages normally, and implement a passive participant of the conversation.

After having become a passive participant through this protocol, the invitee can send a message accepting the invitation and become an active participant thereby. The new member of the conversation is at that point still several steps removed from the status where they can participate in chat; the details of the procedure involved for a new member to become a chatting participant are described in Section 6.3.1.

6.3 State machine contents

Users participating in an (n+1)sec conversation maintain a copy of the state of the conversation state machine. The *state* of the conversation state machine is represented by a valuation of the structure described below.

Structure conversation state machine	
members	set of member
key-exchanges	list of key-exchange
latest-session-id	hash
timeouts	relation of member
events	list of event
status-checksum	hash

The remainder of this section describes the semantics and detailed structure of each of these fields.

6.3.1 Members

An (n+1)sec conversation at any point in time has a set of *participants*, which are those users in the conversation's carrier chat room that are involved or potentially involved with chat in the conversation. Every time the set of participants of a conversation changes —when a new participant joins a conversation, or when an existing participant leaves the conversation— the participants of the conversation perform a key exchange procedure (as outlined in Section 4.2), which will yield a key shared between the conversation participants that can be used to encrypt chat messages with. A newly joined participant of a conversation may not be in the possession of a shared key yet if the key exchange procedure is still in progress, but it is an invariant of conversation that every participant is either in possession of such a session key, or is involved in a key exchange process that aims to accomplish this.

Participants of a conversation are identified by their username and long term public key, which together define their identity. Participants also have a *conversation public key*, which is a temporary key used to sign messages addressed to a conversation. This conversation public key is used as the ephemeral public key used in the Triple Diffie-Hellman deniable authentication protocol, as described in Section 4.1, which is used by the different participants in a conversation to authenticate each other. By signing conversation messages using an ephemeral public key in this way, conversation participants can verify the authenticity of messages without violating deniability. Users participating in multiple conversations must use different conversation public keys for different conversations, which allows users of a carrier chat room to identify the conversation addressed by a particular carrier chat room message, as detailed in Section 7.4.

Besides participants, a conversation may have zero or more *invitee* users. A conversation's invitees, if any, are those users in its carrier chat room that have been invited by one the conversation's participants to join the conversation, and that have either not yet replied to this invitation, or have yet to complete the procedure that ultimately grants participantship.

Several steps are involved between the invitation of a new user into a conversation, and the promotion of that user to a proper participant. New users follow the following process:

Step 1. A participant of the conversation sends an `INVITE` message (Section **??**) to a user in the carrier chat room. This message contains the long term public key of the invitee, as well as their username.

Step 2. The invitee obtains a copy of the conversation state machine, as described in Section 6.2.

Step 3. The invitee sends an `INVITE_ACCEPTANCE` message (Section 7.4.5) to the conversation. This message contains the fresh conversation public key the invitee will use for this conversation.

Step 4. The invitee identifies themself to the participants in the conversation, and vice versa, as outlined in Sections 7.4.6 and 7.4.7.

Step 5. The participant that invited the invitee sends an `AUTHENTICATE_INVITE` message (Section 7.4.8) to the conversation, indicating that they have successfully authenticated the invitee as having the expected identity.

Step 6. The invitee sends a `JOIN` message (Section 7.4.10) to the conversation. This promotes the invitee to a participant.

Aftermath. After the invitee gets promoted to a participant, a key exchange process begins to negotiate a key known to the new participant. When this completes, the new participant can participate in chat in the conversation.

A conversation's participants and its invitees together are denoted as that conversation's *members*. The members of a conversation are those users in its carrier chat room whose messages can potentially affect the conversation's state machine. A conversation state machine contains the complete set of its members, along with relevant metadata.

Structure participant	
username	string
long-term-public-key	publickey
conversation-public-key	publickey

Unidentified invitees are represented in the conversation state machine by the unidentified-invitee structure.

Structure unidentified-invitee	
username	string
long-term-public-key	publickey
inviter	participant reference

Structure identified-invitee	
username	string
long-term-public-key	publickey
conversation-public-key	publickey
inviter	participant reference

Structure authenticated-invitee	
username	string
long-term-public-key	publickey
conversation-public-key	publickey
inviter	participant reference

6.3.2 Key exchanges

6.3.3 Latest session id

6.3.4 Timeouts

6.3.5 Events

6.3.6 State machine checksum

Terminology:

member: user whose nickname exists in a the state machine

four stages of membership:

unidentified invitee: member without conversation pubkey

unauthenticated identified invitee: member with conversation pubkey but no authentication flag

authenticated invitee: member with conversation pubkey that has been sponsored by a participant

participant: member who has fully joined, can participate in key negotiations, can invite others

6.4 Carrier chat room events

7 Messages

7.1 Message structure

opcode / sender / payload struct

7.2 Encoding

tons of icky details here

7.3 Room messages

Room messages are (n+1)sec messages that do not address any particular conversations. They are used to announce a client as being (n+1)sec capable; to announce a client's cryptographic identity in the form of a public key; and for different (n+1)sec clients in a room to confirm each other's identities.

7.3.1 QUIT

The `QUIT` (= 0x01) message causes the sender to effectively leave the room as far as the (n+1)sec protocol is concerned. A user that sent a `QUIT` message is considered in the same state as a user who has not announced (n+1)sec capability; the user has thus left the room from the point of view of the (n+1)sec abstraction, even if the user has not left the carrier chat room.

`QUIT` [0x01]	
cookie	nonce

A `QUIT` message declares the *sender* to have effectively left the (n+1)sec room. An implementation should handle it in the same way as the user leaving the carrier chat room.

When receiving a `QUIT` message, the client SHOULD retract all cryptographic identities of the *sender*, the same way it would do if the *sender* were to leave the carrier chat room instead. The client MUST remove the *sender* from all conversations of which they are a member, the same way as if the *sender* had left the carrier chat room instead, as described in Section **??**.

The *cookie* field carries no significance for clients receiving a `QUIT` message and should be ignored. The field is included to allow an implementation to recognize when it receives its own `QUIT` message, and thus leave the (n+1)sec room in a predictable state when connecting to a carrier chat room in which they are already present.

7.3.2 HELLO

The `HELLO` (= 0x02) message announces the sender's (n+1)sec capability, as well as their cryptographic identity. It is sent either by a client when it enters a

room to announce its presence to other (n+1)sec users in the room, or in reply to such a message by existing users in the room to announce their presence to the newcomer.

`HELLO` [0x02]	
long-term-public-key	publickey
room-public-key	publickey
solicit-replies	boolean

A `HELLO` message sent to a room by a user *sender* indicates that *sender* is an (n+1)sec-capable chat client, claiming to possess the private key corresponding to *long-term-public-key*. Based on this declaration, the user receiving the `HELLO` message is able to invite the *sender* to any current and future conversations, using the announced *long-term-public-key*.

A `HELLO` message sent by a particular user only indicates that this user *claims* to possess the private key corresponding to the *long-term-public-key*. A client can confirm this cryptographic identity by performing an authentication process, as described in Section **??**, using the *room-public-key* as the *sender*'s ephemeral public key for authentication purposes. This authentication process is implemented using the `ROOM_AUTHENTICATION_REQUEST` and `ROOM_AUTHENTICATION` messages. A client SHOULD NOT trust the authenticity of the *sender*'s identity without having successfully completed such an authentication process, and —from a user interface perspective— should probably avoid depicting the *sender* to hold *long-term-public-key* in any way until the authentication process is successfully completed.

To avoid attacks in which a malicious user could waste unlimited resources by sending a large amount of `HELLO` messages claiming different identities, an implementation MAY limit the amount of active invitable identities for a particular *sender* to 1, or limit it based on some other characteristic. If so, a `HELLO` message claiming an identity the implementation does not currently consider active MAY be interpreted by the implementation as an implicit retraction of any earlier claimed identities by this *sender*. If an implementation does interpret a `HELLO` message as a retraction of earlier identities in this way, it may only retract these identities for the sake of representing the visible users in the room, as described in Section 5. In particular, the `HELLO` message MUST NOT have any effect on any conversations of which the *sender* is a member.

The *solicit-replies* flag, if set, indicates that the sender requests all (n+1)sec-capable clients in the room to identify itself; this is generally the case after a user has just joined a room, and wants to be able to invite the people in it to conversations. If this flag is set, any clients that wish to identify themselves can reply with a `HELLO` message of their own, repeating their already-established identity to the new user. However, to avoid bandwidth amplification attacks, implementations SHOULD avoid replying to a message with *solicit-replies* set

from a *sender* to which it has already identified itself. To avoid infinite sequences of `HELLO` replies, a reply message to a message with *solicit-replies* set MUST NOT itself have *solicit-replies* set.

7.3.3 ROOM_AUTHENTICATION_REQUEST

The `ROOM_AUTHENTICATION_REQUEST` (= 0x03) message is used to request a client to authenticate itself based on the cryptographic identity announced in an earlier `HELLO` message. It contains the authentication challenge that forms the base of this authentication process.

`ROOM_AUTHENTICATION_REQUEST` [0x03]	
my-long-term-public-key	publickey
my-room-public-key	publickey
username	string
long-term-public-key	publickey
room-public-key	publickey
authentication-challenge	nonce

A `ROOM_AUTHENTICATION_REQUEST` message is a request to the client that uses the identity consisting of the (*username*, *long-term-public-key*, *room-public-key*) triple to authenticate itself to the user using the (*sender*, *my-long-term-public-key*, *my-room-public-key*) identity. It declares that *sender* will consider *username* authenticated for this identity after receiving a valid authentication confirmation for this pair of identities, described in Section **??**, using *authentication-challenge* as the authentication challenge.

A `ROOM_AUTHENTICATION_REQUEST` message is addressed to the user with username *username* and private keys matching the *long-term-public-key* and *room-public-key*. Any client that does not have this complete identity —including clients that use this username but use different keys— SHOULD ignore the message. The client that does have this identity, if any, can authenticate itself to the *sender* by sending a `ROOM_AUTHENTICATION` message containing the authentication confirmation described above.

7.3.4 ROOM_AUTHENTICATION

The `ROOM_AUTHENTICATION` (= 0x04) message provides confirmation of a cryptographic identity to a single recipient. Assuming the authentication confirmation is valid, this allows the recipient to confirm that the sender holds the private keys they claimed to possess in a `HELLO` message.

ROOM_AUTHENTICATION [0x04]	
my-long-term-public-key	publickey
my-room-public-key	publickey
username	string
long-term-public-key	publickey
room-public-key	publickey
authentication-confirmation	authentication-token

A ROOM_AUTHENTICATION message is a confirmation to the client using the identity consisting of the (*username*, *long-term-public-key*, *room-public-key*) triple that the *sender* holds the private keys corresponding to *my-long-term-public-key* and *my-room-public-key*. The *authentication-confirmation* should contain the authentication token described in Section **??**; if it does, this proves that the sender does indeed hold these private keys.

A ROOM_AUTHENTICATION message is addressed to the user with username *username* and private keys matching the *long-term-public-key* and *room-public-key*. Any client that does not have this complete identity —including clients that use this username but use different keys— SHOULD ignore the message. The client that does have this identity, if any, should consider the authentication valid if and only if

- that client previously sent a ROOM_AUTHENTICATION_REQUEST to the user with identity (*username*, *long-term-public-key*, *room-public-key*), and
- the *authentication-confirmation* field equals the expected authentication token computed from the *authentication-challenge* sent in the accompanying ROOM_AUTHENTICATION_REQUEST message, as specified in Section **??**.

If both requirements hold, this proves that the *sender* holds the private key corresponding to *my-long-term-public-key*.

7.4 Conversation messages

The majority of messages in (n+1)sec are addressed to, and relevant for, a particular conversation. These *conversation messages* contain information addressing the recipient conversation, and affect only the status of that specific conversation. Conversation messages have a shared structure expressing the relation between messages and conversations, and are handled in a similar way.

Conversation message general structure	
conversation-public-key	publickey
message-signature	signature
message-body	octet-stream

Conversation messages are sent by the identified members of a particular conversation, and addressed to all members of that conversation. To denote the conversation to which a conversation message is addressed, conversation messages carry a *conversation-public-key* field, which contains the conversation public key of the identified conversation member that sent the message. The conversation addressed by a conversation message, then, is that conversation (if any) that contains an identified member with username *sender* and conversation public key *conversation-public-key*. Conversation messages also carry a *message-signature*, which is a signature of the message-specific body of the message signed using the private key corresponding to *conversation-public-key*.

The members of a conversation, as described in Section 6, share a representation of the abstract *conversation state machine* that defines the state of the conversation. Coordination between members of a conversation relies on the different members maintaining consensus about the exact state of this state machine; if members of a conversation somehow come to disagree about the state of this state machine, the conversation can no longer be maintained, as described in Section 6.1. It is therefore critical that different members of a conversation process conversation messages in such a way that the abstract conversation state machine is affected in precisely identical ways between their clients.

This section specifies for each conversation message what effect the message has on the abstract conversation state machine of each conversation to which it applies. In order to not break compatibility, implementation MUST implement these specifications to the letter. Some aspects of a client's state for a particular conversation, such as the determination of which other members are authenticated, are not contained in the conversation state machine; for those topics, the specification has a force limited to a behavior that the client SHOULD implement.

Conversation messages contain a signature of the message body, which is used to verify authenticity of messages sent by conversation members. This signature is computed as a cryptographic signature of the message's *message-body* octet stream, as described in Section **??**, using the private key corresponding to the message's *conversation-public-key*. On receiving any conversation message, clients should verify this signature, by confirming that the message's *message-signature* is a valid signature of the message's *message-body* for the *conversation-public-key*. If this signature is not valid, implementations MUST NOT modify the conversation state machine of any conversations. Implementations SHOULD ignore messages with invalid signatures entirely, though they MAY raise some form of impersonation warning instead.

If a conversation message has a valid signature matching its *conversation-public-key*, the message is addressed to any conversations, existing in the carrier chat room in which the message was sent, that contain an identified member with username *sender* and conversation public key *conversation-public-key*. Conver-

sations matching these conditions are said to be *addressed by* the conversation message. On receiving a conversation message with a valid signature, implementations MUST NOT modify the conversation state machine of any conversations that are not addressed by the message. Implementations MUST modify the conversation state machine of any conversations addressed by the message for which the implementation is maintaining a representation, by implementing the rules specified below.

There is one exception that applies to this specification of conversations addressed by conversation messages. The `INVITE_ACCEPTANCE` message, described in Section 7.4.5, is used by unidentified members of a conversation to upgrade their status to an identified member, and is addressed to and processed by certain conversations beyond those that include the *sender* as an identified member. These `INVITE_ACCEPTANCE` messages carry a *message-signature* signed based on its *conversation-public-key* as normal, but are addressed to a larger collection of conversations. The details of this anomaly are specified in Section 7.4.5.

When an implementation receives a conversation message with a valid signature addressed to one or more conversations of which the implementation is maintaining a representation, the implementation must digest the message into those conversations' *status-checksum*, as described in Section **??**. For each of those conversations to which the message is addressed, the implementation must then perform the message-specific processing specified in the remainder of this section.

7.4.1 Event messages

7.4.2 INVITE

The `INVITE` (= 0x11) message is sent by a conversation participant to invite a new user to the conversation, identified by a (username, long term public key) pair. An `INVITE` message informs the invited user of the existence of the conversation, and allows the invited user to start tracking the status of the conversation.

`INVITE` [0x11]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	username	string
	long-term-public-key	publickey

An `INVITE` message

7.4.3 CONVERSATION_STATUS

CONVERSATION_STATUS [0x12]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	username	string
	long-term-public-key	publickey
	conversation-state-machine	state-machine

7.4.4 CONVERSATION_CONFIRMATION

CONVERSATION_CONFIRMATION [0x13]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	username	string
	long-term-public-key	publickey
	conversation-status-hash	hash

7.4.5 INVITE_ACCEPTANCE

INVITE_ACCEPTANCE [0x14]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	my-long-term-public-key	publickey
	inviter-username	string
	inviter-long-term-public-key	publickey
	inviter-conversation-public-key	publickey

Addressed-conversation exception documentation here.

7.4.6 CONVERSATION_AUTHENTICATION_REQUEST

The CONVERSATION_AUTHENTICATION_REQUEST (= 0x15) message is used to request a conversation member to authenticate itself for the cryptographic identity consisting of its long term public key and conversation public key. It is the in-conversation analogue of the ROOM_AUTHENTICATION_REQUEST message. It contains the authentication challenge that forms the base of this authentication process.

CONVERSATION_AUTHENTICATION_REQUEST [0x15]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	username	string
	authentication-challenge	nonce

A CONVERSATION_AUTHENTICATION_REQUEST message is a request to the identified member *username* to authenticate itself to the *sender*, for *username*'s and *sender*'s long term public keys and conversation public keys described by the conversation state machine. It declares that *sender* will consider *username* authenticated for this identity after receiving a valid authentication confirmation for this pair of identities, described in Section **??**, using *authentication-challenge* as the authentication challenge. The *username* identified member, upon receiving a CONVERSATION_AUTHENTICATION_REQUEST message, can authenticate itself to the *sender* by sending a CONVERSATION_AUTHENTICATION message containing this authentication confirmation.

The CONVERSATION_AUTHENTICATION_REQUEST message has no effect on the conversation state machine besides the *status-checksum*.

7.4.7 CONVERSATION_AUTHENTICATION

The CONVERSATION_AUTHENTICATION (= 0x16) message provides confirmation of a cryptographic identity to a member of a conversation. It is the in-conversation analogue of the ROOM_AUTHENTICATION message. Assuming the authentication confirmation is valid, this allows the recipient member to confirm that the sender holds the private keys corresponding to their public keys described by the conversation state machine.

CONVERSATION_AUTHENTICATION [0x16]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	username	string
	authentication-confirmation	authentication-token

TODO: This, and the above message, probably needs to be formalized.

7.4.8 AUTHENTICATE_INVITE

The AUTHENTICATE_INVITE (= 0x17) message is sent by a conversation participant to promote an identified, unauthenticated invitee to an authenticated

invitee. Afterwards, the authenticated invitee is able to join the conversation as a participant.

AUTHENTICATE_INVITE [0x17]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	username	string
	long-term-public-key	publickey
	conversation-public-key	publickey

An AUTHENTICATE_INVITE message indicates that the *sender* has confirmed that the member *username* holds the identity consisting of the (*username*, *long-term-public-key*, *conversation-public-key*) triple, and that *sender* is willing to allow this user entrance into the conversation. Once this allowance has been granted, the invited member is able to join the conversation with a JOIN message.

If a conversation contains an unauthenticated identified invitee invited by a certain participant, that participant SHOULD send an AUTHENTICATE_INVITE message after receiving a CONVERSATION_AUTHENTICATION message that successfully confirms the identity of the invitee. Any other participants may also send such an AUTHENTICATE_INVITE message, and thereby become the inviter of the invited member.

If the conversation state machine addressed by an AUTHENTICATE_INVITE message contains an unauthenticated identified member with identity triple (*username*, *long-term-public-key*, *conversation-public-key*), and the *sender* is a participant of the conversation, then the unauthenticated member gets promoted to an authenticated member. The newly-authenticated member's inviter becomes the message's *sender*. If not, the AUTHENTICATE_INVITE message has no effect on the conversation state machine besides the *status-checksum*.

7.4.9 CANCEL_INVITE

The CANCEL_INVITE (= 0x18) message is sent by a conversation participant to retract any invitations for a given (username, long term public key) pair.

CANCEL_INVITE [0x18]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	username	string
	long-term-public-key	publickey

A CANCEL_INVITE message indicates that the *sender* wants to retract an active

invitation for a given user. If no active invitations for the described member exist, or the member described by the `CANCEL_INVITE` message has since become a participant of the conversation, the message has no effect.

A `CANCEL_INVITE` message removes from the conversation state machine any invited members —be they unidentified invitees, unauthenticated identified invitees, or authenticated invitees— with username *username*, long term public key *long-term-public-key*, and inviter *sender*. Participants of the conversation, and invitations by inviters other than *sender*, are not affected.

7.4.10 JOIN

The `JOIN` (= 0x19) message is sent by an authenticated invitee to upgrade their own status to a participant of the conversation. This triggers the creation of a new key exchange process.

TODO how to render this?

`JOIN` [0x19]	
conversation-public-key	publickey
conversation-signature	signature
conversation-message-body	

7.4.11 LEAVE

TODO how to render this?

7.4.12 CONSISTENCY_STATUS

TODO how to render this?

7.4.13 CONSISTENCY_CHECK

`CONSISTENCY_CHECK` [0x23]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	conversation-status-hash	hash

7.4.14 TIMEOUT

TIMEOUT [0x24]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	username	string
	set-timeout	boolean

7.4.15 KEY_EXCHANGE_PUBLIC_KEY

KEY_EXCHANGE_PUBLIC_KEY [0x31]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	key-id	hash
	session-public-key	publickey

7.4.16 KEY_EXCHANGE_SECRET_SHARE

KEY_EXCHANGE_SECRET_SHARE [0x32]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	key-id	hash
	group-hash	hash
	secret-share	secretshare

7.4.17 KEY_EXCHANGE_ACCEPTANCE

KEY_EXCHANGE_ACCEPTANCE [0x33]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	key-id	hash
	key-hash	hash

7.4.18 KEY_EXCHANGE_REVEAL

KEY_EXCHANGE_REVEAL [0x34]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	key-id	hash
	session-private-key	privatekey

7.4.19 KEY_ACTIVATION

KEY_ACTIVATION [0x41]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	key-id	hash

7.4.20 KEY_RATCHET

KEY_RATCHET [0x42]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	key-id	hash

7.4.21 CHAT

References

[1] M. Abdalla, C. Chevalier, M. Manulis, and D. Pointcheval. *Flexible Group Key Exchange with On-demand Computation of Subgroup Keys*, pages 351–368. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[2] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.

[3] A. Langley, M. Hamburg, and S. Turner. Elliptic Curves for Security. RFC 7748 (Informational), Jan. 2016.

[4] M. Marlinspike. Simplifying OTR deniability. https://whispersystems.org/blog/simplifying-otr-deniability/, 2013.

[5] J. Oikarinen and D. Reed. Internet Relay Chat Protocol. RFC 1459 (Experimental), May 1993. Updated by RFCs 2810, 2811, 2812, 2813, 7194.

[6] P. Saint-Andre. Multi-User Chat. XMPP Standards Foundation XEP 0045, 1999.