(n+1)sec protocol specification — draft

eQualit.ie

February 6, 2017

1 Introduction

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

2.2 Conversations

2.3 Conversation state machine

2.4 Joining a conversation

2.5 Key agreement

2.6 Chat messages

3 Chat Model

3.1 Rooms

3.2 Conversations

3.3 Participants

3.4 Authentication

3.5 Transcript

3.6 Threat model

3.7 Cryptographic assurances

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 corre-

sponding 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 com-

puting 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 $S = H(x_1 + x_2 + x_3)$, where $+$ 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* $T = H(U_B + N + TDH(g^A, g^a, g^B, g^b))$.

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 $T = H(U_B + N + TDH(g^A, g^a, g^B, g^b))$. 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 partic-

ipants 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 partici-

pant 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

5.1 Carrier model

5.2 Carrier limitations

5.3 User presence

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 conversa-

tion 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 re-

sponse 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 *as it is after processing the invitation message*.

3. The invitee records all carrier chat room events —carrier chat room messages, as well as notifications of users joining or leaving the carrier chat room— 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 and events, 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 their 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-key-exchange-id	hash
event-queue	list of event
timeout-matrix	relation over member^2
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. An existing participant of the conversation sends an `INVITE` message (Section 7.4.2) 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.

For invitees of a conversation, the conversation state machine keeps track of the participant that invited them, denoted the *inviter* of the invitee. If the participant that invited an invitee leaves the conversation while the joining process is still in progress —the invitee has not yet become a participant themselves— then the invitation disappears, and the invitee is forcibly removed from the conversation. This measure stops conversations from containing "orphaned" invitees that are not there by request of any of the conversation's current participants. Of course, if an invitee gets dropped from a conversation because of the departure of their inviter, other participants of the conversation may invite the invitee again, and restart the invitation process.

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 the conversation's members, along with their progress in the joining process. This takes the form of a set of structures representing members in different states of progress in the joining process.

A conversation's participants are each represented in the conversation state machine by a participant structure:

Structure participant	
username	string
long-term-public-key	publickey
conversation-public-key	publickey
in-chat	boolean

A conversation state machine stores, for each participant, their *username*, *long-term-public-key*, and *conversation-public-key*. It also stores a flag *in-chat* indicating whether the conversation has yet negotiated a chat key to which the participant has access.

An *unidentified invitee* of a conversation is an invitee member that has been

invited to the conversation using an `INVITE` message, but that has not yet replied with an `INVITE_ACCEPTANCE` message. 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

An unidentified invitee of a conversation has a *username* and *long-term-public-key*, which is the long term identity that has been invited into the conversation. An unidentified invitee does not yet have an empheral public key, for that key is not announced by the invitee until their `INVITE_ACCEPTANCE` message. The conversation state machine also stores, for each unidentified invitee, the *inviter* participant that invited them.

An *unauthenticated identified invitee* of a conversation is an invitee member that has been invited to the conversation with an `INVITE` message and who has replied with an `INVITE_ACCEPTANCE` message, but who has not yet been authenticated by their inviter via an `AUTHENTICATE_INVITE` message. Unauthenticated identified invitees are represented in the conversation state machine by the identified-invitee structure:

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

Unauthenticated identified invitees have a *conversation-public-key* as well as a *username* and *long-term-public-key*. This conversation public key is announced by the invitee in their `INVITE_ACCEPTANCE` message.

An *authenticated invitee* of a conversation is an invitee member that has been authenticated by their inviter with an `INVITE_ACCEPTANCE` message. Authenticated invitees are represented in the conversation state machine by the authenticated-invitee structure:

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

An authenticated invitee is stored the same way in a conversation state machine as an unauthenticated identified invitee; their only difference is the status of their authentication process. Authenticated invitees can become independent participants by sending a `JOIN` message to the conversation.

A conversation state machine contains a field *members*, which is the set of all the members of the conversation. This field is an unordered collection of the conversation's members, each represented as either a participant, unidentified-invitee, identified-invitee, or authenticated-invitee structure.

The identified members of a conversation —that is, those members that are either unauthenticated identified invitees, authenticated invitees, or participants— are uniquely identified by their username. That is to say, a user of a carrier chat room may participate in a conversation with *at most one* long term public key and conversation public key; if a conversation state machine contains an identified member with a given username, it may not contain any other members (identified or not) with that same username. However, if a conversation state machine does not contain any identified members with a given username, it may contain multiple different unidentified invitees with the same username; this allows multiple participants in a conversation to attempt to invite a user to a conversation using different long term public keys. As soon as that carrier chat room user actively participates in the conversation by replying with an `INVITE_ACCEPTANCE` message, and thereby become an unauthenticated identified invitee, the remaining unidentified invitee structures are removed from the conversation state machine, as described in Section 7.4.5.

6.3.2 Key exchanges

The (n+1)sec protocol makes use of a group key exchange protocol, described in Section 4.2, to distribute a shared symmetric key between all participants of a conversation, which is used by that conversation's participants to encrypt chat messages. An instance of this protocol initiates every time the set of participants in a conversation changes —that is, whenever a new participant gets promoted into the conversation, and whenever a participant leaves the conversation— as well as in certain other conditions (see Section 7.4.20). Each instance of the protocol initiated attempts to exchange a key between all users who are a participant of the conversation at the time the key exchange protocol gets initiated.

The group key exchange protocol is a process that requires nontrivial time to complete. When a key exchange process is initiated, those participants that are already in possession of a chat key will continue to send chat messages encrypted using this old key; only when the key exchange process finishes do the participants involved switch to using the key exchanged thereby. This nontrivial duration of a key exchange protocol leads to the distinction between participants that are already part of a conversation's chat, and participants that need to complete a key exchange procedure before attaining this status.

Another consequence of this key usage protocol lies in what happens when a participant leaves a conversation. When a participant leaves a conversation, a

key exchange process starts to negotiate a key to which this former participant does not have access. But until that key exchange process completes and the remaining participants have started using the resulting key, the former participant is still in a position to decrypt conversation chat. The remaining participants, after all, are still sending chat encrypted using the old key that the former participant possesses, and will do so until the new key is negotiated. If the former participant of the conversation still has access to the carrier chat room messages — by having an accomplice in the carrier chat room, or by having control over the carrier chat room server, or even simply by having left the conversation but not having left the carrier chat room — then the former participant can still decrypt conversation chat, until the conversation has finished replacing the old key. Implementations of the (n+1)sec protocol SHOULD therefore make a distinction in the user interface between the point in time where a participant has notionally left the conversation, and the point in time where the conversation has reliably switched to a key whose messages the former participant cannot decrypt.

Because a nontrivial amount of time is involved between the initiation of a key exchange and its completion, a key exchange can begin while an earlier key exchange procedure is still in progress. For example, when two new members of a conversation are promoted to participants shortly after each other, a key exchange that includes the first new participant will start when the first participant gets promoted; if the second participant then gets promoted before this key exchange completes, a second key exchange including both the first and second new participants will start while the first key exchange is still running. As a consequence, multiple key exchanges can be going on in the context of a given conversation at any time.

When a key exchange procedure finishes successfully, the conversation participants involved in the key exchange will switch to using the chat key produced by the key exchange for all further chat. To coordinate this, each participant will send a `KEY_ACTIVATION` message announcing the chat key they will use from that point onward, and will encrypt all chat messages sent after sending that `KEY_ACTIVATION` message using the newly activated key. The successful completion of a key exchange will also cancel all remaining active key exchanges that started before the just-finished key exchange; this ensures that the progression of chat keys used over time follows a monotone sequence, and no keys ever get activated that have since been superceded by more recent keys.

Key exchange procedures do not always finish successfully. Active key exchange processes may sometimes get *cancelled*; this happens when one of the participants of a key exchange leaves the conversation while the key exchange is ongoing, as well as when a key exchange is superseded by the successful completion of a later key exchange. Moreover, key exchanges can be *aborted unsuccessfully* if one of its participants maliciously contributes in such a way that key negotiation fails, as outlined in Section 4.2.2. When this happens, the key exchange

procedure can identify those participants that did not cooperate properly with the key exchange protocol; those participants that contributed maliciously are removed from the conversation, and a new key exchange procedure gets started among the remaining non-malicious participants.

A conversation's state machine contains a record of all key exchanges that are in progress in the conversation at any given point in time. For each such key exchange, the conversation state machine stores all information about the key exchange that users not party to the key exchange ha access to; roughly speaking, this means that the conversation state machine stores a history of all public contributions to the key exchange procedure by its participants. The key exchange protocol is designed in such a way that this information is sufficient for users that passively participate in the conversation to determine whether a key exchange finished successfully, and if not, which participants contributed maliciously and are therefore afterwards removed from the conversation.

The information about a key exchange contained in a conversation's state machine is *not* sufficient for a passively participating user to compute the chat key negotiated by the key exchange process. This is of course intentional; a key exchange process is a procedure to negotiate a key known only to the participants of the key exchange, and not to anyone else. As a consequence, (n+1)sec clients that participate in a key exchange need to store more information than the contents of the conversation state machine. In particular, as long as a key exchange is ongoing, its participants need to store the secrets on which their contributions are based, which form the missing parts that allow these participants to compute the chat key negotiated by the key exchange process. However, these secrets do not form part of the conversation state machine of the conversation of which the key exchange is part, which consists only of those parts of the state of a conversation that users passively participating in a conversation can keep track of.

The conversation state machine represents each active key exchange process with the following structure:

Structure key-exchange	
key-exchange-id	hash
stage	Public-Key, Secret-Share, Acceptance, Reveal
participants	set of key-exchange-participant

Key exchange processes represented in a conversation state machine are identified by their *key-exchange-id*. Messages addressed to a conversation that contribute to a key exchange procedure contain the *key-exchange-id* of the key exchange they contribute to as part of their message structure.

Key exchange structures contain a *stage* field, describing which of the four rounds of the key exchange protocol described in Section 4.2.2 the key exchange procedure is currently executing:

PUBLIC-KEY. A key exchange is in the PUBLIC-KEY stage, or Round 1, when some of its participants U_j have yet to announce their public key y_j.

SECRET-SHARE. A key exchange is in the SECRET-SHARE stage, or Round 2, when all participants U_j have announced their public key y_j, but not all participants have yet announced their linear combination of secrets, or *secret share*, z_j.

ACCEPTANCE. A key exchange is in the ACCEPTANCE stage, or Round 3, when all participants U_j have announced their public key y_j and secret share z_j, but not all participants have yet announced their checksum of the computed key $H(S + \texttt{groupid})$.

REVEAL. A key exchange is in the REVEAL stage, or Round 4, when all participants U_j have announced their public key y_j and secret share z_j and have announced their checksum of the computed key $H(S + \texttt{groupid})$, the different values of $H(S + \texttt{groupid})$ do not agree with each other, and not all participants have yet announced their private key x_j. This stage is only used for key exchanges that do not complete successfully.

Key exchange structures contain a set of their participants, as well as the contributions of each of those participants. This takes the form of the *participants* field, which contains a key-exchange-participant structure for each participant of the key exchange:

Structure key-exchange-participant	
participant	participant reference
session-public-key	optional publickey
secret-share	optional hash
key-digest	optional hash
session-private-key	optional privatekey

The participants of a key exchange are those members of the conversation it occurs in that had the status of participant at the time the key exchange got initiated, each identified by the *participant* field. The key-exchange-participant structure contains four fields representing the participant's contributions in each of the four stages of the key exchange process:

session-public-key. A participant U_j's *session-public-key* field contains their public key y_j announced in the PUBLIC-KEY stage, if any. This field is always present in the SECRET-SHARE and later stages, and present for zero or more participants in the PUBLIC-KEY stage.

secret-share. A participant U_j's *secret-share* field contains their secret share z_j announced in the SECRET-SHARE stage, if any. This field is always present in the ACCEPTANCE and later stages, always absent in the PUBLIC-KEY stage, and present for zero or more participants in the SECRET-SHARE stage.

key-digest. A participant U_j's *key-digest* field contains their announced value of the key digest $H(S + \texttt{groupid})$ announced in the ACCEPTANCE stage,

if any. This field is always present in the REVEAL stage, always absent in the SECRET-SHARE and earlier stages, and present for zero or more participants in the ACCEPTANCE stage.

session-private-key. A participant U_j's *session-private-key* field contains their private key x_j corresponding to their public key $y_j = g^{x_j}$, announced in the REVEAL stage, if any. This field is always absent in the ACCEPTANCE and earlier stages, and present for zero or more participants in the REVEAL stage.

A conversation state machine stores a list *key-exchanges* of all active key exchange processes, each represented by a key-exchange structure, in the order in which they were initiated. This ordering of the recorded key exchanges is relevant when a key exchange process finishes; for when a key exchange process completes successfully, all earlier key exchange processes in the conversation get cancelled, and removed from the conversation state machine. Active key exchange processes also get cancelled, and thereby removed from the conversation state machine, when one of the participants involved in the key exchange leaves the conversation.

New key exchange processes get initiated, and added to the conversation state machine, when a new participant joins the conversation; when a participant leaves the conversation; and when the currently active chat key gets replaced by a fresh key to limit the duration for which any particular chat key is in use. These procedures are described in more detail in Sections 7.4.5, 6.4.3, and 7.4.20, respectively.

Besides storing a lost of active key exchanges, a conversation state machine also stores the *key-exchange-id* of the most recent key exchange that completed successfully. This piece of information is stored in the conversation state machine's *latest-key-exchange-id* field. This tracked piece of information is used in determining whether the currently used chat key has been in use for long enough that it should be replaced with a fresh key, as described in Section 7.4.20. The procedure of keeping the *latest-key-exchange-id* field up-to-date is described in Section 7.4.17.

6.3.3 Events

There are situations in the proceedings of an (n+1)sec conversation in which the processes of the conversation require specific contributions from the conversation's members. For example, active key exchanges require their participants to contribute their parts in each of the key exchange states before the key exchange can proceed to the next stage. For another example, when a participant of a conversation invites a new user, existing identified members of the conversation are required to send a message confirming that they are part of the conversation,

which allows the new invitee to confirm the set of members of the conversation. In either case, the normal function of the conversation —or parts of it— cannot proceed until all relevent members of the conversation have contributed their part.

In these situations, a member that does not cooperate by sending their contribution the conversation processes —perhaps because they maliciously want to sabotage the conversation, or maybe simply because they are suffering network connectivity problems— can hold up useful activity in the conversation indefinitely. To avoid (n+1)sec conversations from being entirely at the mercy of malicious saboteurs and unfortunate network contingencies alike, conversations need to enforce participation in situations where a member contribution is mandatory — by removing nonparticipating members from the conversation, if need be.

To coordinate this enforcement in an orderly fashion, it is important that a conversation's state machine keeps close track of the contributions of its members that the conversation is still waiting on. This record of pending member contributions can then be used as the base of a timeout mechanism, removing from a conversation those members that do not contribute for a sufficiently long time. This timeout mechanism is described in more detail in Section 6.3.4.

The (n+1)sec protocol codifies this concept by storing in each conversation state machine a queue of *pending events*. An *event* is a record in a conversation state machine of a particular future contribution the conversation expects of some subset of its members. Events are added to this queue whenever an activity takes place that requires some members of the conversation to respond; for example, when a key exchange progresses from the PUBLIC-KEY stage to the SECRET-SHARE stage as its last remaining participant sends in their public key, an event is created, denoting that all participants of that key exchange process are to send in a message containing their secret share. Events are removed for a given member once the contribution described by the event is received by the conversation. If an event for a member stays in the event queue for too long a time, this triggers a timeout procedure for the offending member.

A conversation state machine stores a queue of pending events *event-queue* as a sequence of event structures:

Structure event	
members	set of member reference
contribution	event-contribution

Each pending event in a conversation is represented in the conversation state machine as an event structure. An event specifies, in its *contribution* field, a specific type of message its conversation expects to receive from a set of members; the detailed structure of this *contribution* field is described below. The *members* field of an event specifies the set of identified members from which

such a message is expected. This set shrinks each time one of the members in the set sends their contribution message to the conversation; when all members have sent their contribution and the *members* set is empty, the event is removed from the *event-queue*.

To simplify nonparticipation timeout tracking, members of a conversation must send their event contributions in the order in which the events were created. To facilitate this, a conversation state machine's *event-queue* field is structured as a *list*, rather than a set. Whenever a conversation member sends a message that could function as an event contribution, it is verified that this message matches the *first* event in the conversation's event queue for which the sender is part of the event's *members* set. If this is not the case —if the conversation does not contain an event matching the message, or the event matching the message is preceeded by a different event that includes the message's sender, or the conversation does not contain any events that include the message's sender— then this is considered an error, and handled appropriately. The details of this mechanism are described in Section 7.4.1.

An event's *contribution* can refer to one of several different possible structures, each representing a different type of contribution. The different possible contribution structures are outlined below.

A conversation-status-contribution event is created when a conversation participant sends an `INVITE` message. It records the expectation that the sender of that invitation sends a `CONVERSATION_STATUS` message containing an encoding of the conversation state machine for the benefit of the invited user.

Structure conversation-status-contribution	
invitee-username	string
invitee-long-term-public-key	publickey
state-machine-hash	hash

A conversation-status-contribution contains the *invitee-username* and *invitee-long-term-public-key* of the user invited by the `INVITE` message, as well as the hash of the encoding of the conversation state machine that is to be sent to the invited user. A conversation-status-contribution event is satisfied after receiving a `CONVERSATION_STATUS` message, with message fields *username* and *long-term-public-key* equal to the event's *invitee-username* and *invitee-long-term-public-key* fields, and with a *conversation-state-machine* whose hash is equal to the *state-machine-hash*. The details of this expectation are described in Section 7.4.3.

A conversation-confirmation-contribution event is created when a conversation participant sends an `INVITE` message. It records the expectation that the members of the conversation, including the sender of the `INVITE` message, send a `CONVERSATION_CONFIRMATION` message confirming their presence in the conversation for the benefit of the invited user.

Structure conversation-confirmation-contribution	
invitee-username	string
invitee-long-term-public-key	publickey
status-checksum	hash

A conversation-confirmation-contribution contains the *invitee-username* and *invitee-long-term-public-key* of the user invited by the `INVITE` message, as well as the value of the conversation state machine's *status-checksum* after processing the `INVITE` message. A conversation-confirmation-contribution event is satisfied after receiving a `CONVERSATION_CONFIRMATION` message, with message fields *username* and *long-term-public-key* equal to the event's *invitee-username* and *invitee-long-term-public-key* fields, and with a *conversation-status-checksum* message field equal to the *status-checksum* field. The details of this expectation are described in Section 7.4.4.

A consistency-check-contribution event is created when a conversation member sends a `CONSISTENCY_STATUS` keepalive message. It records the expectation that the sender of that message sends a `CONSISTENCY_CHECK` message, containing the conversation's *status-checksum* field as it was after receiving the `CONSISTENCY_STATUS` message.

Structure consistency-check-contribution	
status-checksum	hash

A conversation-check-contribution contains the conversation state machine's *status-checksum* as it was after processing the `CONSISTENCY_STATUS` message. A conversation-check-contribution event is satisfied after receiving a `CONSISTENCY_CHECK` message, whose message field *conversation-status-checksum* is equal to the *status-checksum* field. The details of this expectation are described in Section 7.4.13.

A key-exchange-contribution event is created when a new key exchange process is instantiated, or when an existing key exchange process reaches a new stage. It records the expectation that the participants of that key exchange send their key exchange contribution for the key exchange's active stage.

Structure key-exchange-contribution	
key-exchange-id	hash
stage	Public-Key, Secret-Share, Acceptance, Reveal

A key-exchange-contribution contains the *key-exchange-id* of the key exchange referred to, as well as the key exchange *stage* for which a contribution is expected. Depending on the value of the *stage* field, a key-exchange-contribution event is satisfied after respectively receiving either a `KEY_EXCHANGE_PUBLIC_KEY`, `KEY_EXCHANGE_SECRET_SHARE`, `KEY_EXCHANGE_ACCEPTANCE`, or `KEY_EXCHANGE_REVEAL` message, with a *key-exchange-id* message field set to *key-exchange-id*. The details of this expectation are described in Sections 7.4.15 through 7.4.18. A key-exchange-contribution event need not refer to a key exchange present in the conversation state machine's *key-exchanges* list; when a key exchange process is

cancelled, the representing key-exchange is removed from the conversation's *key-exchanges*, but the assorted key-exchange-contribution event remains in place. Thus, a key-exchange-contribution event may refer to a historic key exchange process that has since been cancelled.

A key-activation-contribution event is created when a key exchange process finishes successfully. It records the expectation that the participants of that key exchange send a `KEY_ACTIVATION` message, indicating that they will start using the newly negoatiated key for all further chat messages.

Structure key-activation-contribution	
key-exchange-id	hash
participants	set of participant reference

A key-activation-contribution contains the *key-exchange-id* of the successfully completed key exchange, as well as the set *participants* of all participants of the conversation that finished this key exchange. A key-activation-contribution event is satisfied after receiving a `KEY_ACTIVATION` message, whose message field *key-id* equals the *key-exchange-id* field. The details of this expectation are described in Section 7.4.19.

6.3.4 Timeouts

Conversations in (n+1)sec implement a timeout procedure. One reason for this, as described in the previous section, is that a member of a conversation could otherwise hold up activity in a conversation indefinitely by failing to participate in a critical process happening inside a conversation. This could happen either intentionally as part of a malicious attack, or unintentionally as a side effect of network connectivity problems. In either case, the conversation has no other option but to eventually remove the nonparticipating member, and continue without them.

Another reason why timeouts are a necessary component of (n+1)sec conversations lies in the desideratum that the members of a conversation are aware of the exact list of users taking part in a chat session. If a participant of a chat session stops receiving chat messages from a certain point onwards because of connectivity problems, the other participants of the chat should notice this absence; it should not be possible for a situation to arise in which a certain participant of a conversation has long since been disconnected from the network, while the other participants of the chat remain under the impression that this absentee is still present. To avoid this situation, members of a conversation send regular keepalive messages indicating that they are still part of the conversation, and members that stop sending these keepalives will eventually get removed from the conversation.

Unfortunately, timeouts are not a trivial notion to implement in a distributed

setting. Conversations cannot rely on the carrier chat room infrastructure to perform the bookkeeping of whom and when to remove from a conversation due to inactivity. Both the possibility of a malicious member remaining connected to the carrier chat room infrastructure while not participating in the conversation, and the possibility of a malicious carrier chat room silently disconnecting a chat participant, are situations in which a conversation's participants need to pass a judgement of timeout status independent from the carrier chat room.

Instead, the members of a conversation need to determine in a purely distributed fashion which members have been nonresponsive long enough to warrant a forcible removal from the conversation. Reaching distributed consensus on this matter is challenging; a situation might easily arise in which a message sent by member A arrives just in time for member B to consider A fast enough to avoid a timeout, but just too late for member C to consider the same.

The (n+1)sec protocol approaches this problem by having each member of a conversation do their own independent tracking of which other members are and are not meeting the conditions to warrant a timeout removal. Members send a message to the conversation when they consider some other member to have crossed the timeout treshold, and these judgements are recorded in the conversation state machine. The decision of whether or not a member is actually removed from a conversation for timeout reasons is then specified as a judgement to be computed from the opinions stored in the conversation state machine.

To implement this scheme, a conversation state machine contains a field *timeout-matrix*, which is a matrix of bits. For each participant P of the conversation, and for each identified member M of the conversation, the *timeout-matrix* field stores a boolean $t_{P,M}$, indicating whether participant P considers member M to have met the condition for a timeout. Participants can send `TIMEOUT` messages, as described in Section 7.4.14, to the conversation to modify their fragments of this *timeout-matrix*. Whenever this matrix reaches one of certain conditions —described below— the conversation removes those members that have been deemed timed out; this situation can arise after receiving a `TIMEOUT` message, or after a conversation member leaves the conversion (voluntarily or otherwise).

If a conversation ever acquires a subset of participants V, such that for all participants $P \in V$ and all participants $Q \notin V$, $t_{P,Q} = \mathsf{true}$ —that is, if there is ever a set of participants that together have declared all other participants timed out— a *symmetric split* between the two groups of participants happens. All participants P in V remove from their version of the conversation state machine all participants $Q \notin V$, and declared them timed out; and symmetrically, all participants $Q \notin V$ remove from their version of the conversation state machine all participants $P \in V$. Afterwards, the two resulting conversations go their own separate independent ways, using the same procedure as the one described in Section 6.1.

The (n+1)sec protocol uses this symmetric notion of timeouts as a defense against malicious participants of a conversation, who might otherwise try to destroy a conversation by declaring all its members as being timed out. The symmetric notion makes this impossible; if a malicious participant P, or even a collection of colluding malicious participants V, maliciously declare a set of other participants timed out, all they will accomplish is a situation where the non-malicious fragment of the conversation remain in a conversation of which the malicious participants V are not part.

If a symmetric split occurs in a conversation, all users passively participating in the conversation will reach the same conclusion of a split happening. Theoretically, any such user could afterwards continue to passively participate in both of the resulting conversations. In practice, any member of the original conversation would only be interested in keeping track of the resulting component that contains themself. Invitees of the original conversation would only remain involved with the resulting component that contains their inviter, for they are no longer part of the component from which their inviter was removed.

If a conversation ever reaches the point where an invitee member M is declared timed out by all participants —that is, if $t_{P,M} = \mathsf{true}$ for all participants P— the invitee is simply removed from the conversation asymmetrically. This is a considerably simpler condition, and unambiguous for all concerned.

The *timeout-matrix*-based architecture of the timeout system used in (n+1)sec is designed in such a way that coherent timeout judgements enforced by the conversation state machine do not rely on the sensibility of timeout judgements of independent conversation members. Indeed different versions of timeout behavior may be suitable for different types of carrier chat systems, as well as other contextual constraints.

Nevertheless, this specification recommends a particular form of timeout judgements that are expected to work well for most carrier chat contexts. Implementations SHOULD follow these recommendations when there is no specific reason to deviate from them.

A conversation participant P should declare a timeout on member M when one of the following conditions arises:

- A particular event for the member M has been pending for more than 60 seconds, or
- The member M has not sent a `CONSISTENCY_STATUS` message for more than 120 seconds, or
- The member M is a participant of the conversation, and should have declared a timeout on a third member N more than 60 seconds ago.

A conversation participant P should retract earlier-declared timeouts on member M when all of these conditions have stopped being true.

6.3.5 State machine checksum

To confirm that the different members of a conversation have maintained consensus about the status of the conversation state machine, each conversation maintains a running digest of all events that caused, or potentially caused, a modification of the conversation state machine. Each time a conversation state machine processes an event that might result in a modification of the status of the conversation state machine, this digest is updated based on both an encoding of the event, and an encoding of the present state of the conversation state machine. This ensures that, if two members of a conversation have computed the same value of this digest, they have both (a) processed the same events, and (b) computed the same conversation state machine contents after doing so. By regularly announcing their computed values of this digest, as described in Sections 7.4.12 and 7.4.13, the members of a conversation can verify that they have successfully maintained consensus about the proceedings of the conversation.

A conversation state machine stores a field *status-checksum* representing this running digest. When a conversation is created, this field is initialized to an arbitrary value. Any time an event happens in the carrier chat room hosting the conversation, this field is updated, according to the protocol specified in Sections 6.4 and 7.4.

6.4 State machine operations

The state machine of a conversation changes its state in response to certain events that happen in the carrier chat room hosting the conversation. The two types in particular that modify a conversation state machine consist of *messages* sent in a carrier chat room, and users of the carrier chat room *leaving* the room. Both events can potentially influence the status of a conversation's state machine, depending on the relation the involved user has with the conversation.

6.4.1 Carrier chat messages

Messages sent in a carrier chat room can be classified into three broad categories: *non-(n+1)sec messages*, *room messages*, and *conversation messages*. Non-(n+1)sec messages are those messages that do not decode as a valid (n+1)sec message, as described in Section 7.2; these messages do not affect the status of any conversation state machines.

Room messages are (n+1)sec messages that are addressed at a carrier chat room in general, rather than any specific conversation, described in Section 7.3. These messages are generally informative about the (n+1)sec capable users present in the carrier chat room, and do not affect any conversation state machines. The exception to this generalization is the `QUIT` message, described in Section 7.3.1, which announces a retraction of the sender's (n+1)sec capability. When a user sends a `QUIT` message in a carrier chat room, this is treated identically as if the user had left the carrier chat room instead, according to the specification outlined in Section 6.4.2.

Conversation messages, which form the majority of (n+1)sec messages, are messages that are addressed to a specific conversation. Conversation messages carry a field used to specify the exact conversation to which they are addressed, as specified in detail in Section 7.4. Conversation messages modify the conversation state machine of the conversation to which they are addressed, if any.

When a conversation message is sent to a carrier chat room, the conversation state machine of the conversation addressed by the message —if any— is modified in the following way:

- The conversation's *status-checksum* field digests the conversation message. This digest is computed by replacing the *status-checksum* field with the term

  $$H(\texttt{state-machine} \mathbin{+\!\!+} \textit{sender} \mathbin{+\!\!+} \textit{opcode} \mathbin{+\!\!+} \textit{message-body})$$

  , where `state-machine` is the encoding of the previous status of the state machine, as defined in Section 7.2; *sender* is the username of the sender of the conversation message, as a byte string; *opcode* is the message opcode, as defined in Section 7.1; and *message-body* is the encoded content of the conversation message, as defined in Section 7.4.

- The conversation message is processed according to the specifications described in Section 7.4.

6.4.2 Users leaving the carrier chat room

When a user leaves a carrier chat room, this has consequences for the conversations of which that user used to be a member. The user is removed as a member from any conversations in the carrier chat room that included them; conversations for which this is the case further record this departure in the conversation's *status-checksum*.

When a user leaves a carrier chat room, this affects all conversations in that room that include a member with a username equal to the username of the

departed user. For conversations that do not include such a member, those conversations' state machines are not affected. Conversations that include such a member are modified in the following way:

- The conversation's *status-checksum* field digests the leaving-user event. This digest is computed by replacing the *status-checksum* field with the term

  $$H(\texttt{state-machine} \mathbin{+\!\!+} \textit{sender} \mathbin{+\!\!+} \texttt{'\textbackslash 0'} \mathbin{+\!\!+} \texttt{"left"})$$

  , where `state-machine` is the encoding of the previous status of the state machine, as defined in Section 7.2, and *sender* is the username of the sender of the conversation message, as a byte string.

- Any members of the conversation with username equal to the username of the departed user are removed from the conversation, according to the specifications described in Section 6.4.3.

6.4.3 Members leaving a conversation

There are several different situations in which a member gets removed from a conversation. Examples of these situations include members leaving the carrier chat room; members getting timed out of a conversation after failing to reply for a sufficiently long time; participants cancelling the invitations of invitees; and malicious participants sabotaging a key exchange process.

When this happens, the departing member is removed from the conversation state machine. Because members are represented in many different locations in the conversation state machine structure, this is a fairly involved operation. The remainder of this section summarizes the modifications that happen to a conversation state machine status in this situation.

When a member is removed from a conversation, that member is removed from the *members* set of all pending events in the conversation state machine's *event-queue*. If this removes the last member from an event's *members* set, the affected event is removed from the event queue, as described in Section 7.4.1. Is this removes the last member from a key-activation-contribution event, this declares *in-chat* all participants in that event's *participants* set, as described in Section 7.4.19. The member removed from the conversation is also removed from the *participants* sets of all key-activation-contribution events, if any.

If the removed member was a participant of the conversation, this causes the cancellation of all active key exchange procedures of which the removed participant was a part. These key exchanges are removed from the conversation state machine, and a new key exchange procedure gets initiated, as described below. The key-exchange-contribution events referring to these key exchanges are not removed, but remain intact without referring to any key exchange procedures.

Removing a participant from a conversation can lead to the removal of multiple other members of the conversation as well. If a participant is removed from a conversation, all invitees in the conversation whose inviter is the removed participant are removed as well. Furthermore, the removal of a participant from a conversation can trigger a timeout procedure, as described in Section 6.3.4; it is possible for a conversation state machine's *timeout-matrix* to cause a symmetric or asymmetric split to happen after removal of a participant, where none happened with the participant in place.

When a participant is removed from a conversation, this triggers the invocation of a new key exchange procedure among the remaining participants. To avoid creating multiple identical parallel key exchanges when multiple participants are removed at the same time, however, this process is structured in such a way that only a single new key exchange is created in this scenario.

To implement this property, a conversation state machine does not create a new key exchange procedure whenever a participant gets removed from the conversation. Instead, at most one new key exchange procedure is created for each carrier chat room event processed by the conversation state machine. If, after completely processing a carrier chat room event —either a chat message sent in the carrier chat room, or a user leaving the room— this processing has resulted in the removal of one or more conversation participants, the conversation state machine creates a new key exchange.

When a key exchange is created in this way, a new key-exchange is added to the conversation state machine. This key exchange has *key-exchange-id* equal to the conversation state machine's *status-checksum* field after digesting the carrier chat room event; a *stage* of PUBLIC-KEY; and a set of *participants* that contains one key-exchange-participant for each remaining participant in the conversation, each with all contributions unset.

When this key exchange is created, a key-exchange-contribution event is added to the conversation state machine's event queue, with *key-exchange-id* equal to the *key-exchange-id* of the newly created key exchange, and *stage* equal to PUBLIC-KEY. All participants of the conversation MUST send a `KEY_EXCHANGE_PUBLIC_KEY` message for this key exchange, containing their public-key contribution for the key exchange process.

7 Messages

7.1 Message structure

7.2 Message encoding

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 6.4.2.

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 procedure, as described in Section 4.1.4, using the *room-public-key* as the *sender*'s ephemeral public key for authentication purposes. This authentication procedure 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 procedure, and — from a user interface perspective— should probably avoid depicting the *sender* to hold *long-term-public-key* in any way until the authentication procedure 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 themselves; 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 SHOULD 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 4.1.4, 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	hash

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 confirmation described in Section 4.1.4; 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 confirmation computed from the *authentication-challenge* sent in the accompanying `ROOM_AUTHENTICATION_REQUEST` message, as specified in Section 4.1.4.

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 over the octet-stream

$$opcode \mathbin{+\!\!+} message\text{-}body$$

, 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 sig-

natures 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*. Conversations 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 6.4.1. 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

Some types of conversation messages have the status of *event messages*. Event messages are those conversation messages that are sent to a conversation to satisfy a pending event, as described in Section 6.3.3. Event messages are sent as a mandatory contribution to a conversation, whenever a new event gets added to the conversation state machine, by those members of the conversation that are party to the event. Event messages should never be sent for any other reason.

Event messages, when received, must correspond to the expectation for event messages encoded in the conversation state machine. A conversation's state machine contains a sequence of pending events; for each pending event, the

conversation state machine further records which members of the conversation have yet to contribute to it. The conversation state machine demands that members send event messages *when their corresponding events are created*, *in the order at which the events are created*, and *in no other situations*.

Implementing this, a conversation state machine addressed by a received event message is affected in the following way:

- If the first event in the conversation state machine for which the event's *members* field contains the message *sender* is satisfied by the received message —the details of which are specified for each specific type of event message— the *sender* is removed from that event's *members*. If that leaves the event's *members* empty, the event is removed from the state machine.
- If the first event in the conversation state machine for which the event's *members* field contains the message *sender* is *not* satisfied by the received message, or if the conversation state machine does not contain any event whose *members* field includes the *sender*, the *sender* is removed from the conversation.

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 indicates that the *sender* wants to invite the user *username* into the conversation, assuming that user holds the private key corresponding to *long-term-public-key*. It also provides an opportunity for that user to acquire a copy of the conversation's state machine, allowing them to decide whether or not to accept the invitation. After receiving both the `INVITE` message and this copy of the state machine, the invited user can accept the invitation with a `INVITE_ACCEPTANCE` message.

After receiving an `INVITE` message, the *sender* of this message sends a `CONVERSATION_STATUS` message containing an encoding of the conversation state machine as it is after

processing the `INVITE` message, allowing the invited user to construct and track a copy of the conversation state machine and thereby learn the status of the conversation to which they are invited. All identified members of the conversation, including the *sender*, send a `CONVERSATION_CONFIRMATION` confirming their presence in the conversation.

An `INVITE` message affects the conversation state machine addressed by it in the following way:

1. If the conversation contains an identified member with username *username*, or if the *sender* is not a participant of the conversation, the `INVITE` message has no effect.
2. If the conversation already contains an unidentified invitee with username *username*, long term public key *long-term-public-key*, and inviter *sender*, the `INVITE` message has no effect.
3. Otherwise, a new unidentified-invitee is added to the conversation, with username *username*, long term public key *long-term-public-key*, and inviter *sender*. If the conversation contains any existing unidentified invitees with username *username* and inviter *sender*, those unidentified invitees are removed from the conversation.
4. A conversation-confirmation-contribution event is added to the conversation, for all identified members of the conversation, with invitee-username *username*, invitee-long-term-public-key *long-term-public-key*, and status-checksum equal to the conversation state machine's *status-checksum* after digesting the `INVITE` message. All identified members MUST send a `CONVERSATION_CONFIRMATION` message matching this event.
5. A conversation-confirmation-status event is added to the conversation, after the conversation-confirmation-contribution event above, for the *sender* of the `INVITE` message. This event has invitee-username *username* and invitee-long-term-public-key *long-term-public-key*.

   The *state-machine-hash* of this event is equal to the digest $H(\texttt{state-machine})$, where `state-machine` is the encoding of the conversation state machine as described in Section **??**, with a *status-checksum* after digesting the `INVITE` message, after adding the unidentified-invitee representing the invited user, after adding the conversation-confirmation-contribution event above, *before* adding the conversation-status-contribution event. The *sender* of the `INVITE` message MUST send a `CONVERSATION_STATUS` message, after having sent the `CONVERSATION_CONFIRMATION` message above, with username *username*, long term public key *long-term-public-key*, and conversation state machine equal to `state-machine`.

A user receiving an `INVITE` message addressed to a conversation to which they are not a member, with *username* and *long-term-public-key* matching the user's

identity, is invited to join the conversation to which the INVITE message is addressed. To make an informed decision as to whether or not to accept this invitation, the invited user needs to first determine the status of the conversation, by acquiring a copy of the conversation's state machine. The invited user can acquire such a copy by following the procedure outlined in Section 6.2, recording all messages in the carrier chat room after the INVITE message, until they receive the accompanying CONVERSATION_STATUS message.

7.4.3 CONVERSATION_STATUS

The CONVERSATION_STATUS (= 0x12) message is sent by the sender of a previous INVITE message, to inform the user invited by that message of the status of the conversation state machine.

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

A CONVERSATION_STATUS message is sent in reply to an INVITE message by the sender of the INVITE message. It informs the user with username *username* and public key *long-term-public-key* of the status of the conversation state machine as it was after processing the INVITE message. The sender of an INVITE message MUST send a CONVERSATION_STATUS message in reply after receiving their own INVITE message, as described in Section 7.4.2.

A CONVERSATION_STATUS message is an event message. As such, the effect of a CONVERSATION_STATUS message on the conversation state machine addressed by it depends on the first pending event in the conversation state machine for which the *members* set includes *sender*. If that event is a conversation-status-contribution event, with *invitee-username* equal to *username*, *invitee-long-term-public-key* equal to *long-term-public-key*, and *state-machine-hash* equal to H(*conversation-state-machine*), the *sender* is removed from that pending event. If the first event has any other form, or if the state machine does not describe any pending events that include *sender*, the *sender* is removed from the conversation.

Other than adjusting a conversation's *event-queue* and *status-checksum*, a CONVERSATION_STATUS message does not affect the addressed conversation's state machine.

The user with username *username* and public key *long-term-public-key*, if previously invited to the conversation with an INVITE message by the same *sender*, can construct the status of the conversation state machine and thereby decide

whether or not to join the conversation. To do this, and to correctly interpret future messages addressed to this conversation, it is not sufficient for the invited user to decode the conversation state machine status stored in the *conversation-state-machine* field. Instead, the user must reconstruct the status of the conversation state machine as it is after processing all messages up to and including the `CONVERSATION_STATUS` message.

The invited user can reconstruct this status by following the following procedure, outlined in Section 6.2:

1. The invited user records all carrier chat room events —carrier chat room messages, as well as notifications of users joining and leaving the carrier chat room— after the most recent `INVITE` message, addressed to this conversation, with username *username*, long term public key *long-term-public-key*, and sender *sender*.
2. When receiving the `CONVERSATION_STATUS` message, the invited user constructs a local copy of the conversation state machine described by the *conversation-state-machine* field.
3. The invited user adds to this conversation state machine the conversation-status-contribution event described in Item 5 of Section 7.4.2.
4. The invited user processes all recorded carrier chat room events, starting from the event immediately following the `INVITE` message, up to and including the `CONVERSATION_STATUS` message, that are addressed to the conversation.

After implementing this procedure, the invited user has constructed a conversation state machine identical to the conversation state machine represented by the members of the conversation. The invited user can then interpret further carrier chat room events that affect the conversation in the same way as all other members of the conversation. If the invited user is an unidentified invitee of the conversation —which they became by the `INVITE` message, but which status might have been retracted in the meantime— they can then choose to become an identified member of the conversation by sending a `INVITE_ACCEPTANCE` message.

7.4.4 CONVERSATION_CONFIRMATION

The `CONVERSATION_CONFIRMATION` message is sent by the identified members of a conversation following an `INVITE` message, to confirm their presence in the conversation to the invited user.

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

A CONVERSATION_CONFIRMATION message is sent in reply to an INVITE message by the identified members of the conversation. It confirms to the user with username *username* and public key *long-term-public-key* that the *sender* is indeed part of the conversation.

A CONVERSATION_CONFIRMATION message is an event message. As such, the effect of a CONVERSATION_CONFIRMATION message on the conversation state machine addressed by it depends on the first pending event in the conversation state machine for which the *members* set includes *sender*. If that event is a conversation-confirmation-contribution event, with *invitee-username* equal to *username*, *invitee-long-term-public-key* equal to *long-term-public-key*, and *status-checksum* equal to *conversation-status-checksum*, the *sender* is removed from that pending event. If the first event has any other form, or if the state machine does not describe any pending events that include *sender*, the *sender* is removed from the conversation.

Other than adjusting a conversation's *event-queue* and *status-checksum*, a CONVERSATION_STATUS message does not affect the addressed conversation's state machine.

7.4.5 INVITE_ACCEPTANCE

The INVITE_ACCEPTANCE (= 0x14) message is sent by unidentified invitees of a conversation to promote their status to that of an unauthenticated identified invitee. It also announces the conversation public key the invitee will use for this conversation.

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

An INVITE_ACCEPTANCE message is sent by an unidentified invitee of a conversation to signify their acceptance of an earlier INVITE message. It contains the

invitee's *conversation-public-key* that they will use to sign messages as an identified member, as well as the identity of their inviter. When the unidentified invitee gets promoted to an identified invitee, they can start identifying themself to the other identified members of the conversation, and vice versa.

All conversation messages other than the `INVITE_ACCEPTANCE` message are addressed to the conversation, if any, that contains an identified member with username *sender* and conversation public key *conversation-public-key*. As a consequence, other conversation messages can be addressed at a given conversation only when sent by an identified member of the conversation. The `INVITE_ACCEPTANCE` message is an exception to this rule, which makes it the single message that can be sent by an unidentified invitee to the conversation to which they are invited.

An `INVITE_ACCEPTANCE` message is addressed to all conversations, if any, that contain an identified member with username *sender* and conversation public key *conversation-public-key*, or contain an identified member with username *inviter-username* and conversation public key *inviter-conversation-public-key*. As a consequence, `INVITE_ACCEPTANCE` messages MUST be processed by any conversations matching this broader-than-usual condition; in particular, such conversations MUST digest the `INVITE_ACCEPTANCE` message into their state machine's *status-digest* field, as described in Section 6.4.1.

If a conversation state machine addressed by an `INVITE_ACCEPTANCE` message contains an unidentified invitee with username *sender*, long term public key *my-long-term-public-key*, and inviter *inviter-username*, and also contains a participant with username *inviter-username*, long term public key *inviter-long-term-public-key*, and conversation public key *inviter-conversation-public-key*, then this unidentified invitee gets promoted to an identified invitee with conversation public key *conversation-public-key*. This promotion is implemented by removing from the conversation state machine all unidentified invitees with username *sender*, and adding an identified-invitee with username *sender*, long term public key *my-long-term-public-key*, conversation public key *conversation-public-key*, and inviter *inviter-username*.

If instead a conversation state machine addressed by an `INVITE_ACCEPTANCE` message already contains an identified member —be it an unauthenticated identified invitee, authenticated invitee, or participant— with username *sender*, this member is removed from the conversation state machine.

7.4.6 CONVERSATION_AUTHENTICATION_REQUEST

The `CONVERSATION_AUTHENTICATION_REQUEST` (= 0x15) message is used to request an identified conversation member to authenticate themself for the cryptographic identity consisting of their 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 themself 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 4.1.4, 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.

Other than adjusting a conversation's *status-checksum*, a CONVERSATION_AUTHENTICATION_REQUEST message does not affect the addressed conversation's state machine.

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	hash

A CONVERSATION_AUTHENTICATION message is a confirmation to the identified member *username* that the *sender* holds the private keys corresponding to their long term public key and conversation public key as described by the conversation state machine. The *authentication-confirmation* should contain the au-

thentication confirmation described in Section 4.1.4; if it does, this proves that the sender does indeed hold these private keys.

The member *username* should consider the authentication valid if and only if

- that member previously sent a `CONVERSATION_AUTHENTICATION_REQUEST` to the member *username*, and
- the *authentication-confirmation* field equals the expected authentication confirmation computed from the *authentication-challenge* sent in the accompanying `CONVERSATION_AUTHENTICATION_REQUEST` message, for *username*'s and *sender*'s long term public keys and conversation public keys described in the conversation state machine, as specified in Section 4.1.4.

If both requirements hold, this proves that the *sender* holds the private keys corresponding to their public keys described by the conversation state machine.

Other than adjusting a conversation's *status-checksum*, a `CONVERSATION_AUTHENTICATION` message does not affect the addressed conversation's state machine.

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 authenticated 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.

JOIN [0x19]	
conversation-public-key	publickey
conversation-signature	signature
conversation-message-body	
	⟨ empty ⟩

A JOIN message indicates that the *sender* wants to finish joining the conversation, getting promoted from an authenticated invitee to a full participant of the conversation. If the sender is not an authenticated invitee of the conversation, the message has no effect.

If the conversation state machine addressed by a JOIN message does not contain an authenticated invitee with username *sender*, the message does not affect the addressed conversation's state machine other than adjusting a conversation's *status-checksum*. Otherwise, the authenticated-invitee representing the user is removed from the conversation, and a participant is added instead, with *username*, *long-term-public-key*, and *conversation-public-key* identical to the values present for the authenticated-invitee, and *in-chat* = false.

When the *sender* is promoted to a participant like this, a new key-exchange is added to the conversation state machine. This key exchange has *key-exchange-id* equal to the conversation state machine's *status-checksum* field after digesting the JOIN message; a *stage* of PUBLIC-KEY; and a set of *participants* that contains one key-exchange-participant for each participant in the conversation, including the newly-promoted *sender*, each with all contributions unset.

When this key exchange is created, a key-exchange-contribution event is added to the conversation state machine's event queue, with *key-exchange-id* equal to the *key-exchange-id* of the newly created key exchange, and *stage* equal to PUBLIC-KEY. All participants of the conversation MUST send a KEY_EXCHANGE_PUBLIC_KEY message for this key exchange, containing their public-key contribution for the key exchange process.

7.4.11 LEAVE

The LEAVE (= 0x21) message is sent by an identified member of a conversation to leave the conversation. This removes the sender from the conversation.

LEAVE [0x21]	
conversation-public-key	publickey
conversation-signature	signature
conversation-message-body	
	⟨ empty ⟩

A LEAVE message indicates that the *sender* wants to leave the conversation to

which the LEAVE message is addressed.

The conversation state machine addressed by a LEAVE message removes the member *sender* from its set of members.

7.4.12 CONSISTENCY_STATUS

The CONSISTENCY_STATUS message is sent by the identified members of a conversation as a keepalive message, demonstrating to the other members of the conversation that they are still present. The CONSISTENCY_STATUS message triggers a followup CONSISTENCY_CHECK message, which proves that the sender has a correct representation of the conversation state machine.

CONSISTENCY_STATUS [0x22]	
conversation-public-key	publickey
conversation-signature	signature
conversation-message-body	
	⟨ empty ⟩

A CONSISTENCY_STATUS message is sent periodically by all identified members of a conversation to demonstrate that they are still actively present in the conversation. All identified members SHOULD send such a message every 60 seconds, or risk being declared timed out by other members of the conversation.

A conversation state machine addressed by a CONSISTENCY_STATUS message adds a consistency-check-contribution event to the conversation's event queue, with *status-checksum* equal to the *status-checksum* of the conversation state machine after digesting the CONSISTENCY_STATUS message, and *members* consisting only of the *sender* of the message. The *sender* of the message MUST send a CONSISTENCY_CHECK message with a *conversation-status-checksum* equal to this *status-checksum*.

7.4.13 CONSISTENCY_CHECK

The CONSISTENCY_CHECK message is sent by identified members of a conversation as a followup to the CONSISTENCY_STATUS message. It contains the conversation's *status-checksum* as it was after processing the CONSISTENCY_STATUS message, demonstrating to the members of the conversation that conversation state consensus is maintained.

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

A CONSISTENCY_CHECK message is sent by identified members of a conversation after receiving their own CONSISTENCY_STATUS message. The CONSISTENCY_CHECK message contains the conversation state machine's *status-checksum* as it was after processing the CONSISTENCY_STATUS message. By checking that *conversation-status-checksum* against the copy stored in the conversation state machine's event queue, members of the conversation can verify that the *sender* was still maintaining an accurate copy of the conversation state machine when receiving the CONSISTENCY_STATUS message.

A CONSISTENCY_CHECK message is an event message. As such, the effect of a CONSISTENCY_CHECK message on the conversation state machine addressed by it depends on the first pending event in the conversation state machine for which the *members* set includes *sender*. If that event is a conversation-check-contribution event, with *status-checksum* equal to *conversation-status-checksum*, the *sender* is removed from that pending event. If the first event has any other form, or if the state machine does not describe any pending events that include *sender*, the *sender* is removed from the conversation.

Other than adjusting a conversation's *event-queue* and *status-checksum*, a CONSISTENCY_CHECK message does not affect the addressed conversation's state machine.

7.4.14 TIMEOUT

The TIMEOUT message is sent by participants of a conversation to either declare that they have judged a different member to be timed out, or to declare that they have judged that member to no longer be timed out. This judgement may cause the timed out member to be removed from the conversation.

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

A TIMEOUT message is a declaration by the *sender* that the member with username *username* is either judged timed out (if *set-timeout* = true), or judged no longer timed out (if *set-timeout* = false). This triggers a possible removal of the timed-out member from the conversation, as described in Section 6.3.4.

If the conversation state machine addressed by a TIMEOUT message does not contain an identified member with username *username*, or it does not contain a participant with username *sender*, the TIMEOUT message has no effect on the conversation state machine besides the *status-checksum*. Otherwise, the conversation state machine sets the *timeout-matrix* field $t_{sender,username}$ to true if *set-timeout* = true, and to false otherwise. Afterwards, a symmetric or asymmetric split of the conversation may occur depending on the contents of the *timeout-matrix*, as described in Section 6.3.4.

7.4.15 KEY_EXCHANGE_PUBLIC_KEY

The KEY_EXCHANGE_PUBLIC_KEY message is sent by participants of a conversation to submit their PUBLIC-KEY contributions to a key exchange process. This contribution contains their public key used for the duration of the key exchange.

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

A KEY_EXCHANGE_PUBLIC_KEY message contains the sender's PUBLIC-KEY contribution to a key exchange process, as described in Section 6.3.2. Participants of a key exchange procedure MUST send this message when the key exchange is created.

A KEY_EXCHANGE_PUBLIC_KEY message is an event message. As such, the effect of a KEY_EXCHANGE_PUBLIC_KEY message on the conversation state machine addressed by it depends on the first pending event in the conversation state machine for which the *members* set includes *sender*. If that event is a key-exchange-contribution event, with *key-exchange-id* equal to *key-exchange-id* and *stage* equal to PUBLIC-KEY, the *sender* is removed from that pending event. If the first event has any other form, or if the state machine does not describe any pending events that include *sender*, the *sender* is removed from the conversation.

If a KEY_EXCHANGE_PUBLIC_KEY message does match a pending event as above, the effect of the KEY_EXCHANGE_PUBLIC_KEY depends on whether or not the conversation state machine contains a key exchange with *key-exchange-id* equal to *key-exchange-id*. If it does not, the KEY_EXCHANGE_PUBLIC_KEY message does not affect the addressed conversation's state machine other than adjusting the conversation's *event-queue* and *status-checksum*. If it does, the message's *session-public-key* is stored as the *sender*'s *session-public-key* in the key-exchange referred to.

If, after setting the *sender*'s *session-public-key* field in this way, all participants of the conversation have a *session-public-key* set, the key exchange reaches the SECRET-SHARE stage. When this happens, the key-exchange's *stage* field is set to SECRET-SHARE. Afterwards, a key-exchange-contribution event is added to the conversation state machine's event queue, with *key-exchange-id* equal to the *key-exchange-id* of the key exchange, and *stage* equal to SECRET-SHARE. All participants of the conversation MUST send a `KEY_EXCHANGE_SECRET_SHARE` message for this key exchange, containing their secret-share contribution for the key exchange process.

7.4.16 KEY_EXCHANGE_SECRET_SHARE

The `KEY_EXCHANGE_SECRET_SHARE` message is sent by participants of a conversation to submit their SECRET-SHARE contributions to a key exchange process. This contribution contains their linear combination of secrets shared with their neighbours in the key exchange logical circle.

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

A `KEY_EXCHANGE_SECRET_SHARE` message contains the sender's SECRET-SHARE contribution to a key exchange process, as described in Section 6.3.2. Participants of a key exchange procedure MUST send this message when the key exchange reaches the SECRET-SHARE stage.

A `KEY_EXCHANGE_SECRET_SHARE` message is an event message. As such, the effect of a `KEY_EXCHANGE_SECRET_SHARE` message on the conversation state machine addressed by it depends on the first pending event in the conversation state machine for which the *members* set includes *sender*. If that event is a key-exchange-contribution event, with *key-exchange-id* equal to *key-exchange-id* and *stage* equal to SECRET-SHARE, the *sender* is removed from that pending event. If the first event has any other form, or if the state machine does not describe any pending events that include *sender*, the *sender* is removed from the conversation.

If a `KEY_EXCHANGE_SECRET_SHARE` message does match a pending event as above, the effect of the `KEY_EXCHANGE_SECRET_SHARE` depends on whether or not the conversation state machine contains a key exchange with *key-exchange-id* equal to *key-exchange-id*. If it does not, the `KEY_EXCHANGE_SECRET_SHARE` message does not affect the addressed conversation's state machine other than adjusting the conversation's *event-queue* and *status-checksum*.

If it does, the message's *group-hash* field is checked against the expected value derived from the contents of the conversation state machine. If the *group-hash* is not equal to `groupid`, as specified in Section 4.2.2, the *sender* is removed from the conversation. It it is equal, the message's *secret-share* is stored as the *sender*'s *secret-share* in the key-exchange referred to.

If, after setting the *sender*'s *secret-share* field in this way, all participants of the conversation have a *secret-share* set, the key exchange reaches the ACCEPTANCE stage. When this happens, the key-exchange's *stage* field is set to ACCEPTANCE. Afterwards, a key-exchange-contribution event is added to the conversation state machine's event queue, with *key-exchange-id* equal to the *key-exchange-id* of the key exchange, and *stage* equal to ACCEPTANCE. All participants of the conversation MUST send a `KEY_EXCHANGE_ACCEPTANCE` message for this key exchange, containing their key-digest contribution for the key exchange process.

7.4.17 KEY_EXCHANGE_ACCEPTANCE

The `KEY_EXCHANGE_ACCEPTANCE` message is sent by participants of a conversation to submit their ACCEPTANCE contributions to a key exchange process. This contribution contains their computed value of a digest of the computed key, indicating whether or not the different participants of the key exchange have computed the same key.

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

A `KEY_EXCHANGE_ACCEPTANCE` message contains the sender's ACCEPTANCE contribution to a key exchange process, as described in Section 6.3.2. Participants of a key exchange procedure MUST send this message when the key exchange reaches the ACCEPTANCE stage.

A `KEY_EXCHANGE_ACCEPTANCE` message is an event message. As such, the effect of a `KEY_EXCHANGE_ACCEPTANCE` message on the conversation state machine addressed by it depends on the first pending event in the conversation state machine for which the *members* set includes *sender*. If that event is a key-exchange-contribution event, with *key-exchange-id* equal to *key-exchange-id* and *stage* equal to ACCEPTANCE, the *sender* is removed from that pending event. If the first event has any other form, or if the state machine does not describe any pending events that include *sender*, the *sender* is removed from the conversation.

If a KEY_EXCHANGE_ACCEPTANCE message does match a pending event as above, the effect of the KEY_EXCHANGE_ACCEPTANCE depends on whether or not the conversation state machine contains a key exchange with *key-exchange-id* equal to *key-exchange-id*. If it does not, the KEY_EXCHANGE_ACCEPTANCE message does not affect the addressed conversation's state machine other than adjusting the conversation's *event-queue* and *status-checksum*. If it does, the message's *key-digest* is stored as the *sender*'s *key-digest* in the key-exchange referred to.

If, after setting the *sender*'s *key-digest* field in this way, all participants of the conversation have a *session-public-key* set, the key exchange finishes the ACCEPTANCE stage. When this happens, the consequences depend on whether or not there is consensus about the exchanged key.

If, when finishing the ACCEPTANCE stage, all participants of the key exchange have submitted the same value of the *key-digest* field, the key exchange process has finished successfully. If this happens, the key-exchange is removed from the conversation state machine, and the state machine's *latest-key-exchange-id* field is set to the *key-exchange-id* of the removed key exchange. Afterwards, a key-activation-contribution event is added to the conversation state machine's event queue, with *key-exchange-id* equal to the *key-exchange-id* of the removed key exchange, and *participants* equal to the participants of the removed key exchange. All participants of the conversation MUST send a KEY_ACTIVATION message with *key-id* set to the *key-exchange-id*, indicating that they will henceforth encrypt chat messages with the newly accepted key.

If instead, when finishing the ACCEPTANCE stage, not all participants of the key exchange have submitted the same value of the *key-digest* field, the key exchange reaches the REVEAL stage. If this happens, the key-exchange's *stage* field is set to REVEAL. Afterwards, a key-exchange-contribution event is added to the conversation state machine's event queue, with *key-exchange-id* equal to the *key-exchange-id* of the key exchange, and *stage* equal to REVEAL. All participants of the conversation MUST send a KEY_EXCHANGE_REVEAL message for this key exchange, containing their reveal contribution for the key exchange process.

7.4.18 KEY_EXCHANGE_REVEAL

The KEY_EXCHANGE_REVEAL message is sent by participants of a conversation to submit their REVEAL contributions to a key exchange process. This contribution contains their private key used for the duration of the key exchange, which is used to determine which of the participants of the key exchange caused the key exchange to fail to finish successfully.

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

A KEY_EXCHANGE_REVEAL message contains the sender's REVEAL contribution to a key exchange process, as described in Section 6.3.2. Participants of a key exchange procedure MUST send this message when the key exchange reaches the REVEAL stage.

A KEY_EXCHANGE_REVEAL message is an event message. As such, the effect of a KEY_EXCHANGE_REVEAL message on the conversation state machine addressed by it depends on the first pending event in the conversation state machine for which the *members* set includes *sender*. If that event is a key-exchange-contribution event, with *key-exchange-id* equal to *key-exchange-id* and *stage* equal to REVEAL, the *sender* is removed from that pending event. If the first event has any other form, or if the state machine does not describe any pending events that include *sender*, the *sender* is removed from the conversation.

If a KEY_EXCHANGE_REVEAL message does match a pending event as above, the effect of the KEY_EXCHANGE_REVEAL depends on whether or not the conversation state machine contains a key exchange with *key-exchange-id* equal to *key-exchange-id*. If it does not, the KEY_EXCHANGE_REVEAL message does not affect the addressed conversation's state machine other than adjusting the conversation's *event-queue* and *status-checksum*. If it does, the message's *session-private-key* is stored as the *sender*'s *session-private-key* in the key-exchange referred to.

If, after setting the *sender*'s *session-private-key* field in this way, all participants of the conversation have a *session-private-key* set, the key exchange has finished unsuccessfully. When this happens, the key-exchange is removed from the conversation state machine, and the details of the key-exchange record are used to determine which of the participants of the key exchange caused the key exchange to fail. This judgement is determined in the following way:

1. First, for each participant, the public key corresponding to their recorded *session-private-key* is computed. If for any participant this public key does not match their recorded *session-public-key*, all participants for which these keys do not match are marked as *malicious*, and the judgement halts.

2. If the previous step does not yield any malicious participants, then by using the private keys of all participants, the secret shares for all participants are computed. This makes use of the fact that a Triple Diffie-Hellman secret between two parties can be computed when knowing the ephemeral

private key of both parties, and the long term private key of neither party, as described in Section 4.1.2. If for any participant this public key does not match their recorded *secret-share*, all participants for which these secret shares do not match are marked as *malicious*, and the judgement halts.

3. If the previous steps do not yield any malicious participants, then by using the shared secrets of all participants, the shared key S and key checksum $H(S + \texttt{groupid})$ are computed. All participants for which this computed key checksum does not match their recorded *key-digest*, of which there should be at least one, are marked as *malicious*, and the judgement is finished.

After computing a nonempty set of malicious participants of the key exchange procedure, the malicious participants of the key exchange are removed from the conversation. This triggers the creation of a new key exchange, as described in Section 6.4.3.

7.4.19 KEY_ACTIVATION

The KEY_ACTIVATION message is sent by participants of a conversation after successfully finishing a key exchange. It announces that the sender will henceforth use the negotiated key to encrypt all future chat messages.

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

A KEY_ACTIVATION message indicates that the *sender* has negotiated a new key with a recently-completed key exchange, and will use this key in the future when sending encrypted messages. All CHAT messages received in the future sent by *sender* will be assumed encrypted using this key.

A KEY_ACTIVATION message is an event message. As such, the effect of a KEY_ACTIVATION message on the conversation state machine addressed by it depends on the first pending event in the conversation state machine for which the *members* set includes *sender*. If that event is a key-activation-contribution event, with *key-exchange-id* equal to *key-id*, the *sender* is removed from that pending event. If the first event has any other form, or if the state machine does not describe any pending events that include *sender*, the *sender* is removed from the conversation.

If, after removing the *sender* from the key-activation-contribution event in this way, the key-activation-contribution event does not have any remaining members

left —if that event's *members* field is empty, after removign *sender* from it— then the key identified by *key-id* has been accepted by all participants described in the key-activation-contribution's *participants* field. Because all these participants are also participant to any key exchange processes that may have taken place after the exchange negotiating the *key-id* key, this implies that all these *participants* will have access to all future encrypted chat in the conversation. To mark this fact, the conversation state machine sets the *in-chat* field to true for all participants in the *participants* set.

7.4.20 KEY_RATCHET

The KEY_RATCHET message is sent by a participant of a conversation to announce their opinion that the most recently negotiated chat key has been in use for too long, and should be replaced. This triggers a new key exchange process, which will negotiate a replacement key.

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

A KEY_RATCHET message indicates that the *sender* is of the opinion that the chat key negotiated by the most recent successfully finished key exchange process needs to be replaced with a new key, to avoid using a chat key for a long enough time that attacking the secrets on which it is based is a realistic attack possibility. The message field *key-id* is the id of the key for which the *sender* holds this opinion; if this key is still the most recently exchanged key when the KEY_RATCHET message is received, and no existing key exchange is in progress to replace it, a new key exchange progress is initiated.

If the *sender* of the KEY_RATCHET message is not a participant of the conversation state machine addressed by the KEY_RATCHET message, the conversation state machine is not affected by the KEY_RATCHET message other than adjusting the conversation's *status-checksum*. Likewise, if the conversation state machine's *latest-key-exchange-id* field is not equal to *key-id*, or the state machine's *key-exchanges* list contains at least one active key exchange, the KEY_RATCHET message has no effect.

If the *sender* is a participant of the conversation state machine, *latest-key-exchange-id* is equal to *key-id*, and no key exchanges are in progress in the conversation state machine, a new key-exchange is added to the conversation state machine. This key exchange has *key-exchange-id* equal to the conversation state machine's *status-checksum* field after digesting the KEY_RATCHET message; a *stage* of PUBLIC-KEY; and a set of *participants* that contains one

key-exchange-participant for each participant in the conversation, each with all contributions unset.

When this key exchange is created, a key-exchange-contribution event is added to the conversation state machine's event queue, with *key-exchange-id* equal to the *key-exchange-id* of the newly created key exchange, and *stage* equal to PUBLIC-KEY. All participants of the conversation MUST send a `KEY_EXCHANGE_PUBLIC_KEY` message for this key exchange, containing their public-key contribution for the key exchange process.

7.4.21 CHAT

The `CHAT` message is sent by a participant of a conversation to send an encrypted chat message to the participants of the conversation. This message is encrypted with the key most recently activated by the message's sender using a `KEY_ACTIVATION` message, and can be decrypted by those participants that were party to the key exchange process that negotiated that key.

`CHAT` [0x43]		
conversation-public-key	publickey	
conversation-signature	signature	
conversation-message-body		
	encrypted-message	octet-stream

A `CHAT` message contains an encrypted chat message, encrypted using the key announced in the most recent `KEY_ACTIVATION` message sent by the *sender*. A `CHAT` message can be decrypted by any users that were part of the key exchange that negotiated that key, but it is addressed only to such participants for which *in-chat* is set in the conversation state machine.

Other than adjusting a conversation's *status-checksum*, a `CHAT` message does not affect the addressed conversation's state machine. This ensures that any member unable to decrypt a `CHAT` message can nonetheless participate in the conversation of which it is part.

A `CHAT` message contains an encrypted payload, *encrypted-message*, that is encrypted using the key announced in the most recent `KEY_ACTIVATION` message sent by *sender*. Members of a conversation that do not possess the key announced in that `KEY_ACTIVATION` message SHOULD ignore the `CHAT` message.

Of the participants of a conversation that can decrypt a `CHAT` message, some may have a status of *in-chat* = false set in the conversation state machine. This status indicates that not all encrypted chat sent in the conversation is yet encrypted using a key to which these participants have access; specifically, it indicates that there are participants of the conversation that may yet send

messages encrypted to an older key to which the participant does not have access. As a consequence, such participants with *in-chat* = false may be able to decrypt some subset of the encrypted chat messages sent in the conversation, but not others. In the interest of representing a consistent chat history between different participants of a conversation, implementations SHOULD not attempt to decrypt messages sent in a conversation until having reached the status in which *in-chat* = true. However, other participants of the conversation cannot rely on such a participant not yet decrypting any messages; therefore, implementations SHOULD make a distinction in the user interface between a participant who cannot yet decrypt any messages, and a participant who can —for the moment— decrypt some but not all conversation messages, similar to the conundrum described in Section 6.3.2.

A participant with access to the key with which a `CHAT` message is encrypted can decrypt the *encrypted-message* field. This decrypted field has the following structure:

Structure encrypted-message		
signature	signature	
message-body	octet-stream	
	message-id	integer
	message	string

The decrypted contents of an encrypted message are signed using the session-private-key used by *sender* in the key exchange procedure that negotiated the key with which the `CHAT` message was encrypted. This *signature* is computed over the octet-stream making up the *message-body*. A participant that decrypted a chat message MUST verify this signature before attempting to display the chat message inside, and SHOULD ignore the message if this signature is not valid.

As a protection against replay attacks the decrypted contents of an encrypted message contain a *message-id* field, which holds the number of messages previously sent by *sender* encrypted using this key. To avoid accepting a replayed copy of an earlier message sent by *sender* as valid, a participant that decrypted a chat message MUST verify the correctness of this *message-id* field, and SHOULD ignore the message if the *message-id* is not valid.

If the *signature* and the *message-id* of a decrypted message are both valid, this indicates that the encrypted message is authentic. In this case, the implementation SHOULD display the transmitted *message* to the user, as sent by the participant *sender*.

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.