Network Working Group
Internet-Draft
Expires: October 3, 2007

F. Strauss
TU Braunschweig
S. Ransom
Universitaet zu Luebeck
S. Lahde
O. Wellnitz
TU Braunschweig
April 2007

P2P CHAT - A Peer-to-Peer Chat Protocol
draft-strauss-p2p-chat-08

Status of this Memo

Copyright Notice

Abstract

This memo presents the third version of a protocol for a peer-to-peer based chat system. In this version it is extended for supporting message exchange in Mobile Ad Hoc Networks (MANETs) that suffer from node mobility. Messages can be cryptographically signed, encrypted and anonymized allowing authentic and closed group communication as well as anonymous communication. This work is input for a practical course on software engineering at Technische Universitaet Braunschweig. It is experimental work that is not intended to go to the IETF standards track.

Table of Contents

1. Introduction

Chat systems allow groups of people to exchange text messages and even data like photos, documents, etc. in realtime within so-called channels: While each participant of a channel can write messages to that channel, he/she can also read all messages sent by other people to the same channel. Besides messages, the user can enter commands to his local chat application in order to join or leave channels, list existing channels, etc.

Traditional chat systems are based on servers - either a single server or a static network of servers. Each user has to connect to a well-known server in order to start chat communication. In contrast to that server approach, this document presents a peer-to-peer chat architecture for Mobile Ad Hoc Networks (MANETs). This means that all nodes in the chat network behave equally from the protocol point of view. To establish the network of chat participants the nodes just have to discover potential peers in their neighborhood. Since mobile nodes leave and enter the network any time, the network topology may change continuously and partitioned networks may have to be merged. For this reason, the chat architecture has to be disruption tolerant in order to handle such events.

1.1. Terminology

Node, Peer: A node is one active component in the chat network. Note that there may be multiple nodes located on a common host. Note also that one node may serve multiple local users. From the perspective of one node and regarding a given link, the peer is the node at the remote end of the link.

Link, connection: Two mobile nodes may be connected through a link if they are in transmission range. Each link, once established, can be used in both directions. Although the term "connection" is used to indicate that two nodes are in transmission range and communicate with each other, data is always exchanges via UDP which is not connection-oriented.

Message: Data transferred between two nodes on a link is encoded in a message. Messages are XML documents. See Section 3.

User, User ID: A user is a (typically) human participant in the chat network. Each user has an unique ID which is typically equal to his/her email address, e.g. strauss@ibr.cs.tu-bs.de.

Channel, Channel ID: Communication is organized in channels. Each user (not the node!) may subscribe to and unsubscribe from a channel in order to control which text messages received at a node shall be

presented to that user. A channel is associated with a short one-word pattern, e.g. "smalltalk" and a channel ID. The channel ID consists of two parts separated by an "@". The last part MUST be the local hostname. The generation of first part is an implementation issue, but the nodes SHOULD use UUIDs (universally unique identifier)[ITUX667]. In either case, the channel IDs MUST differ from other known IDs.

Channel creator: Each user can create a new channel.

Public channel: Traffic on a public channel is not encrypted, thus each user can join such a channel, read all messages on such a channel and send text messages to such a channel. Public channels can be identified by their channel name.

Closed channel: Closed channels are associated with a list of channel members given by their user IDs. This list is initialized by the channel creator and may be extended by each channel member. (This is not a security risk, since each member can decode and forward the traffic, anyway.) Closed channels MUST be identified by their channel ID and their channel name.

Anonymous channel: The anonymous channel is omnipresent and allows the users to post messages anonymously. Every active user is automatically a member of this channel. The channel name "Anonymous" is reserved for the anonymous channel and MUST NOT be used for other channels.

Channel member: Participant of a channel.

Channel key: Messages to closed channels are encrypted using a common symmetric channel key. The channel creator decides about the key type and its parameters. The key is distributed through asymmetrically encrypted key distribution messages to the channel members.

Certificate: X.509 certificates are used to prove and verify user IDs and to use their public keys to exchange the symmetric keys of closed channels, verify messages and encrypt anonymous messages in the path-obscuring phase.

Peer list: A list of communication partners that are assumed to be in transmission range. This list is only used in infrastructure mode. If a client operates in infrastructure mode, it will ignore all messages received from nodes that are not listed on the local peer list.

1.2. Terms of Requirement Levels

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. The Architecture

2.1. The Peer-to-Peer Network

The chat network is organized in a peer-to-peer overlay network that is established within a mobile ad hoc network (MANET). Due to node mobility in the underlying network, the topology of the peer-to-peer network may change over time. In general the node MUST use the well-known UDP port 8888 for its communication. Upon startup of a node, it MUST start sending periodic HELLO messages. Moreover, it tries to discover other peer nodes in its transmission range. This is done by listening to HELLO messages sent by other peers in the network.

Each connection can be used symmetrically, i.e., messages can be sent in both directions. The details of peer node and connection management are an implementation issue.

2.2. Infrastructure Mode

The P2P Chat Protocol is designed for MANETs. However, there should be a possibility to use the protocol in infrastructure networks. Thus, each node MUST have a list of peer nodes (IPv4 and port) stored in a persistent local storage which is called peer list further on. This list explicitly defines the available links and is only used if the peer runs in infrastructure mode. In this case it will ignore all messages received from peers that are not listed on the local peer list. The infrastructure mode may be used for establishing a "virtual" overlay network for demonstration or testing purposes. Since it should be possible to test several clients on a single host, the port numbers in this list MAY differ from the well-known port. Note that messages (especially broadcast messages) MAY have also be sent to all nodes using a different port than the well-known one.

2.3. Mobility Issues

An important issue in MANETs is the mobility of communication partners. Due to mobility, the topology of the network may change continuously. It also may happen that the network becomes partitioned into two or more independent networks and later re-merges again. These characteristics also affect the communication in the overlay peer-to-peer network, especially regarding to the channel management.

If two MANETs merge due to mobility there may exist different CHANNELs with the same name that were created independently. In the case of public channels, these channels MUST be merged after the separate networks became one. Each member of one of the channels that discovers duplicate channels having the same channel name, MUST

send a new CHANNEL including the members of both previous channels. The new channel ID of the merged CHANNEL MUST be the smallest ID of the previous CHANNELs (in ASCII representation). Closed channels MUST NOT be merged since for both channels there is an individual shared secret. Thus, these channels (and the respective messages to these channels) MUST be distinguished by their channel IDs.

2.4. Routing

Routing in a MANET is a challenging issue. Each chat message has specific attributes like TTL and FLOOD that are used for routing purposes (see Section 3.2). There are three different ways of routing specific messages in the network.

Some messages (ACK, HELLO, QUIT, ROUTING) are only sent from a sending node to a neighboring peer node without further forwarding. These messages MUST be sent with ttl="1" and with flood="false" (see Section 3.2 for further information on ttl and flood).

Some messages (GETCERTIFICATE, CERTIFICATE, GETKEY, KEY, MESSAGE, OBSCURE, NACK) are multicasted to a set of receivers (maybe one, i.e., unicast). In this case the message is sent with an initial value of e.g. ttl="32" and with flood="false". The selection of an appropriate ttl value is an implementation issue. The message contains an according "receiver" element for every receiver, i.e. it is a so called explicit multicast. When a node receives such a message, it MUST acknowledge the reception to the previous hop by sending an ACK message. Afterwards, it can recognize any local users for whom the message has to be processed, and it also has to forward the message to any remaining receivers. To do this, the routing table is used to determine the appropriate outgoing links and to filter the set of remaining receivers that can be reached on the shortest paths on those outgoing links (the list of remaining receivers SHOULD only include the IDs of that users that can be reached over the specific link). Updates to the routing table are calculated based on the current state of the table and subsequently received ROUTING messages (Section 3.3.11). A message's ttl may becomes zero on its way to the destinations. In this case, the node that holds the message has to inform the source node of that message about the fact that the message has not reached all receivers by sending a NACK message.

Some messages (CHANNEL, JOIN, LEAVE, MESSAGE in case of anonymous messages) are flooded to the whole network. In this case the message is sent with an initial value of e.g. ttl="32" and with flood="true". Note that the TTL value of CHANNEL messages MUST be lower than the minimal channel announcement interval (see Section 3.3.4). The selection of an appropriate ttl value is an implementation issue. If

a peer receives a flooded message for the second time, which can be recognized through the message ID and a ring buffer backlog of received message IDs, the message MUST be dropped.

Message forwarding MUST always be done in a hop-by-hop manner. This means that each hop is responsible for delivering the message to the next hop on the path to the destination. For each multicasted message (GETCERTIFICATE, CERTIFICATE, GETKEY, KEY, MESSAGE, OBSCURE) the peer can be sure that the message reached the next hop if it received an ACK for the respective message. If a peer detects a route failure (if it receives no ACK from the next hop for that message), it MUST find an alternative way to deliver the packet towards the destination. For this, it e.g. may immediately try to find an alternative route to the destination or cache the message for some time and try it later again. The concrete stragegy is an implementation issue. If a message's TTL expires while the message is cached, the node MUST send a NACK message to the sender of the message and drop the original message.

2.5. Security

In order to support signed messages, closed channel communication and anonymous communication, a user MUST have an RSA keypair, the keylength being 1024 bit. In order to be able to prove his/her identity each user must have an X.509 certificate for his/her public key.

Signing of the messages SHOULD always be done with the exception of anonymous messages. The mandatory algorithms are MD5 with RSA. The concatenation of all text sub-nodes of the message starting at the message type node, but excluding any receiver elements, form the input to the signing process (see the XML Schema definition in Appendix A for further details).

All messages of a closed channel MUST be encrypted using symmetric cryptography. Therefore, the creator of a closed channel generates a shared secret key which is encrypted and sent to all participants (upon request and optionally in advance) using the respective public key of each participant. The RSA encryption MUST use the Electronic Code Book (ECB) mode with the padding scheme PKCS1 (v1.5). The mandatory symmetric algorithm is standard 64 (56) bit DES in Cipher Block Chaining (CBC) mode with the padding scheme PKCS5 enabled. Further symmetric algorithms may be added in the future. Note: Only the content of the text element in a MESSAGE message is encrypted. If the message contains one or more attachments, the content of the filename, applicationtype, and data elements is encrypted individually. (see the XML Schema definition in Appendix A for further details).

2.6. Anonymous Communication

In order to allow anonymous posting of messages, an anonymous channel is supported. This channel is omnipresent, i.e., does not need channel announcements, does not die etc. Each active user is automatically a member of it. The channel name "Anonymous" is reserved for the anonymous channel and MUST NOT be used for other channels.

Anonymous communication consists of two phases: the path-obscuring and the posting of the message.

In the beginning, the sender creates a list of at least 3 and up to 5 random active users, which help in obscuring the path and thus the sender's identity. The MESSAGE message that the sender wants to post MUST be encrypted with the public key of the last user in the list (RSA in ECB mode with PKCS1 (v1.5) padding). This data forms the content of the OBSCURE message that is destined at the last user. This OBSCURE message is then again encrypted with the public key of the last but one user in the list and forms the content of another OBSCURE message, which is destined at that user. This scheme MUST be repeated until the list is empty. The final OBSCURE message, is sent to the first user in the list.

Upon reception of an OBSCURE message at its target node, the content MUST be decrypted. If the decrypted message content is another OBSCURE message it MUST be sent on to the receiver that is already placed in the message. In case of the decrypted message being the message to be posted, it MUST be flooded to the network.

In order to prevent message loss during the path-obscuring phase, the original sender SHOULD resend the message if it has not been posted to the anonymous channel after a timeout. In this case, the sender SHOULD create a different list of active users and encrypt the message according to the above defined rules.

3. The Protocol

This section describes the peer-to-peer chat protocol version 2.

3.1. Protocol Message Format

Each message represents a well-formed XML [XML] document. Each message MUST be sent to its destination in an individual UDP datagram [RFC768]. It is not allowed to put several messages in a single datagram. The maximum message size is limited to 63000 bytes. Only NACK messages are allowed to exceed this limit. Each message (including NACKs) MUST fit into one UDP datagram. It is not be possible to split up messages over several datagrams.

An XML Schema definition for messages is given in Appendix A.

3.2. Protocol Message Exchange

Every message is of a specific message type given by the one and only direct child element of the root element of the message XML document. All protocol messages are handled asynchronously, i.e., a node MUST NOT block and wait for a certain reply message when a request message was sent.

Each chat message contains two attributes used for routing purposes: TTL and FLOOD. The TTL value of each message MUST be decremented on each hop as well as for every full second a message is held on a node before it is forwarded to the next hop. Note that the initial TTL values are mandatory, since they may be used to derive routing information. The FLOOD flag indicated whether a message MUST be flooded through the network.

In addition, messages MAY contain a DELAY attribute that states the cumulative queuing time in seconds at all intermediate nodes. This value is incremented for every second a packet is queued on an intermediate node for any reason. This value can be used to determine messages with high delays.

3.3. Protocol Messages

This section describes all protocol message types. The named message types are equal to the name of the one and only direct child element of the corresponding "chat-message" root element. Note that message types are just written all caps in this document for readability.

3.3.1. HELLO

The HELLO message MUST be sent periodically by each node. The interval between two successive HELLO messages MUST be 2 seconds +/- 0.5 seconds. A peer SHOULD choose a new interval randomly between 1.5 and 2.5 seconds for each individual beacon in order to reduce the collision probability. The HELLO message can be used to detect other peers in the local neighborhood. If no further HELLO message is received for three successive HELLO intervals, the remote node may be moved out of the transmission range.

3.3.2. ACK

An ACK message MUST be sent in response to a GETCERTIFICATE, CERTIFICATE, GETKEY, KEY, MESSAGE or OBSCURE message. The message informs the sender or forwarder of a message that this message has arrived at the next hop. It MUST include the message ID of the received message.

3.3.3. NACK

A NACK message MUST be sent if the TTL of a message becomes zero on the way to the destinations. It is used to inform the original sender about the fact that its message has not received all destinations. The actions that are taken by a node that receives a NACK for one of its messages sent is an implementation issue. The NACK message MUST only be sent to the original sender of the message. Moreover, it MUST include the whole original message. In addition, a NACK messages includes the user ID of its sender and a unique message ID.

3.3.4. CHANNEL

A CHANNEL message for a specific channel is sent

- once when a local user creates a channel,
- when the node did not receive a CHANNEL message for the channel for a certain time and as long as a local user is still subscribed to that channel. This time SHOULD be a random value in the range between 50 and 60 seconds.
- Furthermore, CHANNEL messages MIGHT be sent once for each known channel to a newly connected peer node.

This ensures that channels keep alive and known (including the subscribers) to all nodes. A CHANNEL message SHOULD be sent with ttl="32". The TTL value MUST be lower than the minimal channel

announcement interval. The CHANNEL message is always flooded to the whole network.

Each channel is identified by a channel name and a channel ID. The name of a new channel MUST differ from the names of existing channels. While the name is a descriptive identifier for a channel, each channel also has a unique channel ID that MUST differ from other known IDs and SHOULD be generated according to Section 1.1. If a node recognizes two channels with the same names, they have to be handled according to Section 2.3.

The CHANNEL message always contains the list of subscribed channel members. Only these users are allowed to send messages to the channel. Messages for a specific channel coming from other users MUST be ignored. Nevertheless, a node MAY wait for the next channel announcement before dropping the message. In case of a closed channel only the users in the subscription list of the CHANNEL message are allowed to retrieve the session key through a GETKEY/KEY message pair. Users of a closed channel are allowed to add further members to the list by sending a new CHANNEL message with the extended list. CHANNEL messages of closed channels MUST be signed by the sender to prevent non-members from adding themselves or other users to the list.There is no way to reduce the members list of closed channels, since it is obviously not possible to revoke a shared secret.

Note that a public channel cannot later be turned into a closed channel. There is no way to actively close or remove a channel. A channel is "gone", when there are no more nodes that send regular CHANNEL messages.

3.3.5. JOIN

A node sends a JOIN message

- once when a local user subscribes to a channel,
- when a local user is already subscribed to a channel for which a CHANNEL message is received that does not (yet) list that local user.

The JOIN message is always flooded to the whole network.

3.3.6. LEAVE

A node sends a LEAVE message

- once when a local user unsubscribes from a channel,
- when a local user is not subscribed to a channel for which a CHANNEL message is received that does (still) list that local user.

The LEAVE message is always flooded to the whole network.

3.3.7. GETCERTIFICATE

A node may request a certificate for a given user ID by sending a GETCERTIFICATE message. This is usually caused by the wish to verify the signature of a received message, when the according certificate is not yet available at the local node. Note that the sender of the GETCERTIFICATE (and the receiver of the response) is identified by a user ID although certificates may be used to serve all users at the local node.

3.3.8. CERTIFICATE

A node can send a certificate encapsulated in a CERTIFICATE message. This MAY be done in advance (e.g., to all channel members before sending a signed message to a channel) to avoid subsequent GETCERTIFICATE/CERTIFICATE message traffic. Furthermore, a CERTIFICATE message MUST be sent in response to a received GETCERTIFICATE message for a local user. An intermediate node MAY also send a CERTIFICATE message in response to a GETCERTIFICATE message if it has the certificate of the requested user in its local cache. In this case, it MUST NOT forward the GETCERTIFICATE message along the route to the destination.

3.3.9. GETKEY

A node may request a channel key for a given closed channel by sending a GETKEY message to the one member that sent and signed the channel announcement. This MUST only be done if a previous CHANNEL message has been received and the local user's ID is on the members list of that channel.

If a GETKEY message is received and a KEY response (see below) can be sent, the GETKEY message MUST be dropped and not forwarded to any other receivers.

3.3.10. KEY

A node MUST send a channel key encapsulated in a KEY message in response to a GETKEY message sent to the local user. The key MUST be encrypted for that user. The sender MUST be sure that the receiver

is a member of the closed channel and the public key used for the key encryption really belongs to the receiver. This is usually done through a certificate verification process.

3.3.11. ROUTING

A node MUST send a ROUTING message on a regular basis to all its neighboring peers. Each entry of the ROUTING message represents a known user and signals the number of hops it takes to reach that user, i.e., the receiver of the ROUTING message can subsequently reach those users via that link with n+1 hops.

The interval to send ROUTING messages SHOULD be chosen randomly from the interval [8,10] seconds. Additionally, the node MIGHT send ROUTING messages upon changes to the local routing table, but it MUST NOT send more than one message per second.

Note that a node MIGHT also "learn" additional routing information based on other messages than ROUTING messages through the sender and TTL values.

3.3.12. MESSAGE

The users' text messages within channels are distributed in MESSAGE messages. With the exception of anonymous MESSAGE messages, the initial sender of a MESSAGE message MUST put all members of the channel into the receivers list. A MESSAGE message MAY contain one or more attachments. The attached data MUST be Base64 encoded. An attached file is described by a filename and the application MIME type [RFC2045][RFC2046]. In case of a closed channel, the content of messages MUST be sent encrypted with the channel key (which probably must be retrieved through a GETKEY/KEY conversation first).

Each MESSAGE message contains a UTC timestamp which is generated by the original sender. This timestamp MAY be used to put the received message into the context of a discussion. Note that there is typically no global time in a MANET. Therefore, this timestamp depends on the peer sending the message and timestamps from different peers MAY NOT be consistent.

A MESSAGE message MUST NOT exceed the packet size limit mentioned in Section 3.1

A node that receives a MESSAGE message MUST forward the message on each link (except the incoming link) on which it can reach at least one of the receivers through the shortest path. All receivers for which the link does not supply the shortest path are removed before forwarding.

In case of the MESSAGE message being the final message of an anonymization chain, its message ID MUST be inserted by the sender of the message, and the anonymous originator MUST NOT have placed a message ID in the message. Furthermore, the message MUST be destined at the channel 'Anonymous'.

3.3.13. OBSCURE

An OBSCURE message is an anonymous message in the path- obscuring phase. The content MUST contain another OBSCURE message or a MESSAGE message which MUST be encrypted with the receiver's public key.

A node that receives an OBSCURE message MUST decrypt the content with its private key and MUST send on the resulting message in a new chat message.

The message ID of an OBSCURE message MUST be inserted by its immediate sender, i.e., not the initial creator of the OBSCURE message chain.

An OBSCURE message MUST NOT exceed the packet size limit mentioned in Section 3.1

4. Security Considerations

This document defines an application protocol to carry potentially private or authentic information. The protocol addresses these needs through the application of strong cryptographic digest and encryption algorithms. X.509 certificates are used to verify identities of end users. This document requires implementations to support a minimal common set of cryptographic algorithms so that from the specifications point of view secure communication can be guaranteed. However, anonymous communication makes assumptions that cannot be always guaranteed, i.e. an attacker is not able to trace the complete path an anonymous message takes before being posted to the anonymous channel.

At the current status the protocol takes no measures to protect against DoS attacks by peers.

5. Acknowledgements

The protocol described in this memo is the output of a practical course on distributed systems that has been conducted during two terms at Technische Universitaet Braunschweig in April - July, 2003 and in the same summer term in 2004. A first step of the work presented here has been developed by the approx. 48 participating students of the first course in 2003 during a two-weeks design phase and a subsequent protocol design colloquium. Then the second course in 2004 improved routing concepts significantly and added anonymous communication in a similar design and colloquium way. In 2007 the protocol has been extended to support mobility issues in Wireless Ad Hoc Networks. Thus, the transport protocol has be switched from TCP to UDP. In addition, several changes regarding neighbour discovery, disruption tolerance, and node mobility were introduced. The extended protocol is the basis for a practical course on software engeneering at Technische Universitaet Braunschweig in April - July 2007

The authors of this memo are basically the editors who have put the design decisions together in a common specification document along with some refinements and mobility support. Hence, the authors' thanks go to all the ambitious students of the summer 2003 and summer 2004 PVS courses.

Appendix A. XML Schema Definition for Messages

```
<?xml version="1.0"?>

<!-- $Id$
  -
  -->

<xsd:schema
  targetNamespace="http://www.ibr.cs.tu-bs.de/chat-message"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.ibr.cs.tu-bs.de/chat-message"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  xml:lang="en">

  <xsd:annotation>
    <xsd:documentation>
      This XML Schema defines the p2p-chat protocol messages.
    </xsd:documentation>
  </xsd:annotation>

  <!--
    -                              Types
    -->

  <xsd:simpleType name="UserIDType">
    <xsd:annotation>
      <xsd:documentation>
        Elements of this type represent a globally unique user
        identification. It has to contain a user name part followed by
        an @ character and a domain part. It is highly recommended to
        use the users valid Internet email address.  Note that it has
        to be treated case-insensitive.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[a-zA-Z0-9_\.\-]{1,127}@[a-zA-
      Z0-9_\.\-]{1,128}"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="MessageIDType">
    <xsd:annotation>
      <xsd:documentation>
        Elements of this type are used to form a unique identification
        for messages per user. The pair of a UserID and a MessageID
        builds a globally unique message identification. How the
```

```
          MessageID is actually built is an implementation issue. It
          might be a persistent counter incremented with each message
          generation, or it might be based on a timestamp, for example.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[a-zA-Z0-9@_\.\-]{1,256}"/>
      </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="ChannelIDType">
      <xsd:annotation>
        <xsd:documentation>
          Elements of this type are used to form a unique identification
          for channels. The pair of a channel name and a channelID
          builds a globally unique channel identification. How the
          channelID is actually built is an implementation issue. The
          channelID MUST differ from other known IDs. Is consists of two
          parts sperarated by an "@". The first part SHOULD be
          generated randomly on the basis of UUIDs. The second part is
          the name of the local host.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[a-zA-Z0-9_\.\-]{1,127}@[a-zA-
        Z0-9_\.\-]{1,128}"/>
      </xsd:restriction>
    </xsd:simpleType>


    <xsd:simpleType name="TTLType">
      <xsd:annotation>
        <xsd:documentation>
          An integer time-to-live value. Note that the value range is
          restricted.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:int">
        <xsd:minInclusive value="1"/>
        <xsd:maxInclusive value="360"/>
     </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="ChannelNameType">
      <xsd:annotation>
        <xsd:documentation>
          The name of a channel.  Note that it has to be treated
          case-insensitive. The name "Anonymous" is reserved
```

```
          for the
          anonymous channel and MUST NOT be used for other channels.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[a-zA-Z0-9_\.\-]{1,16}"/>
      </xsd:restriction>
    </xsd:simpleType>


    <xsd:complexType name="DestinationType">
      <xsd:annotation>
        <xsd:documentation>
          A destination is part of a routing message. It signals
          the number of hops to reach a user via the link on which
          the routing message is sent.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="user" type=
        "UserIDType"/>
        <xsd:element name="hops" type=
        "xsd:unsignedInt"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="CertificateType">
      <xsd:annotation>
        <xsd:documentation>
          A certificate consists of the user ID to which the certificate
          belongs and the certificate itself encoded in Base64.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="user" type=
        "UserIDType"/>
        <xsd:element name="data" type=
        "xsd:base64Binary"/>
      </xsd:sequence>
    </xsd:complexType>



    <xsd:complexType name="AttachmentType">
      <xsd:annotation>
        <xsd:documentation>
          A message attachment. The applicationtype states the MIME Type
          of the attached file.
```

```
          The application data MUST always be Base64 encoded.
          If the iv attribute is present,
          it is a message of a closed channel and uses the
          initialization
          vector given by the iv attribute. The content of all elements
          MUST be de-/encrypted individually.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="filename" type=
        "xsd:string"/>
        <xsd:element name="applicationtype" type=
        "xsd:string"/>
        <xsd:element name="data" type=
        "xsd:base64Binary"/>
      </xsd:sequence>
      <xsd:attribute name="iv" type=
      "xsd:base64Binary"/>
    </xsd:complexType>


    <xsd:simpleType name="CipherType">
      <xsd:annotation>
        <xsd:documentation>
          A label to identify a symmetric cipher algorithm.  So far,
          just one (reasonable) cipher is defined, but others may be
          added in future revision of the protocol.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:NMTOKEN">
        <xsd:enumeration value="NONE">
          <xsd:annotation>
            <xsd:documentation>
              No encryption.
            </xsd:documentation>
          </xsd:annotation>
        </xsd:enumeration>
        <xsd:enumeration value="DES-CBC">
          <xsd:annotation>
            <xsd:documentation>
              64(56) bit DES in Cipher Block Chaining mode with PKCS #5
              padding.
            </xsd:documentation>
          </xsd:annotation>
        </xsd:enumeration>
      </xsd:restriction>
    </xsd:simpleType>
```

```
    <xsd:simpleType name="SignType">
      <xsd:annotation>
        <xsd:documentation>
          A label to identify a message digest algorithm.  So far, just
          one algorithm is defined, but others may be added in future
          revision of the protocol.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:NMTOKEN">
        <xsd:enumeration value="MD5">
          <xsd:annotation>
            <xsd:documentation>
              MD5 digest (encrypted with the senders private key).
            </xsd:documentation>
          </xsd:annotation>
        </xsd:enumeration>
      </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="SignatureType">
      <xsd:annotation>
        <xsd:documentation>
          A Base64 encoded signature. It is calculated for the
          concatenated UTF-8 encoded values (without attributes) of all
          child elements of the chat-message element in depth-first
          order.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:base64Binary"/>
    </xsd:simpleType>




    <!--
      -                   Root Element and Message Types
      -->

    <xsd:element name="chat-message" type=
    "ChatMessage"/>

    <xsd:complexType name="ChatMessage">
      <xsd:annotation>
        <xsd:documentation>
          This element represents a p2p-chat protocol
          message.
        </xsd:documentation>
      </xsd:annotation>
```

```
        <xsd:sequence>
          <xsd:choice>
            <xsd:element name="hello"          type=
            "HelloMessage"/>
            <xsd:element name="ack"            type=
            "AckMessage"/>
            <xsd:element name="nack"           type=
            "NackMessage"/>
            <xsd:element name="channel"        type=
            "ChannelMessage"/>
            <xsd:element name="join"           type=
            "JoinMessage"/>
            <xsd:element name="leave"          type=
            "LeaveMessage"/>
            <xsd:element name="getcertificate"
            type="GetCertificateMessage"/>
            <xsd:element name="certificate"
            type="CertificateMessage"/>
            <xsd:element name="getkey"         type=
            "GetKeyMessage"/>
            <xsd:element name="key"            type=
            "KeyMessage"/>
            <xsd:element name="routing"        type=
            "RoutingMessage"/>
            <xsd:element name="message"        type=
            "MessageMessage"/>
            <xsd:element name="obscure"        type=
            "ObscureMessage"/>
          </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="ttl"        type=
        "TTLType"
        default="1"/>
        <xsd:attribute name="delay"      type=
        "xsd:int"
        default="0"/>
        <xsd:attribute name="flood"      type=
        "xsd:boolean"
        default="false"/>
        <xsd:attribute name="signtype"   type=
        "SignType"/>
        <xsd:attribute name="signature"  type=
        "SignatureType"/>
    </xsd:complexType>

    <xsd:complexType name="HelloMessage">
      <xsd:annotation>
        <xsd:documentation>
```

```
          The "hello" message is the first message sent by
          each peer of
          a newly established connection. The peer may send an informal
          greeting text. A HELLO message MAY contain more than one
          sender
          if a node serves more than one user.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="sender"    type=
        "UserIDType"
          minOccurs="1" maxOccurs="unbounded"/>
        <xsd:element name="messageid" type=
        "MessageIDType"/>
        <xsd:element name="version"   type=
        "xsd:int"/>
        <xsd:element name="greeting"  type=
        "xsd:string" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="AckMessage">
      <xsd:annotation>
        <xsd:documentation>
          The "ack" message is sent by a peer that receives a
          multicasted
          message from another peer in order to inform the sender that
          this message reached the next hop. It always includes the ID
          of the received message.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="sender"    type=
        "UserIDType"/>
        <xsd:element name="messageid" type=
        "MessageIDType"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="NackMessage">
      <xsd:annotation>
        <xsd:documentation>
          The "Nack" message is sent by a peer that receives a
          multicasted
          message from another peer and is not able to forward it
          towards its destination. The "Nack" message is
          always sent to
          the original source of that message and informs the sender
```

```
          that its message has not received its destination. The
          original message has to be attached to the "Nack".
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="sender"    type=
        "UserIDType"/>
        <xsd:element name="messageid" type=
        "MessageIDType"/>
        <xsd:element name="message"   type=
        "ChatMessage"/>
      </xsd:sequence>
    </xsd:complexType>


    <xsd:complexType name="ChannelMessage">
      <xsd:annotation>
        <xsd:documentation>
          The "channel" message announces an active channel.
          The
          "closed" flag denotes whether it is a closed
          channel.
          Each known subscriber is listed in a "member" child
          element.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="sender"        type=
        "UserIDType"/>
        <xsd:element name="messageid"     type=
        "MessageIDType"/>
        <xsd:element name="channel"       type=
        "ChannelNameType"/>
        <xsd:element name="channelid"     type=
        "ChannelIDType"/>
        <xsd:element name="description"   type=
        "xsd:string"/>
        <xsd:element name="member"        type=
        "UserIDType"
          minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="closed"        type=
      "xsd:boolean"
      default="false"/>
    </xsd:complexType>

    <xsd:complexType name="JoinMessage">
      <xsd:annotation>
```

```
        <xsd:documentation>
          The "join" message signal the subscription of a user
          to
          a non-closed channel.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="sender"        type=
        "UserIDType"/>
        <xsd:element name="messageid"     type=
        "MessageIDType"/>
        <xsd:element name="channel"       type=
        "ChannelNameType"/>
        <xsd:element name="channelid"     type=
        "ChannelIDType"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="LeaveMessage">
      <xsd:annotation>
        <xsd:documentation>
          The "leave" message signal the unsubscription of a
          user from
          a non-closed channel.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="sender"        type=
        "UserIDType"/>
        <xsd:element name="messageid"     type=
        "MessageIDType"/>
        <xsd:element name="channel"       type=
        "ChannelNameType"/>
        <xsd:element name="channelid"     type=
        "ChannelIDType"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="GetCertificateMessage">
      <xsd:annotation>
        <xsd:documentation>
          A node may send a "getcertificate" message to
          request a
          certificate for a given user ID.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="sender"    type=
```

```
        "UserIDType"/>
        <xsd:element name="receiver"  type=
        "UserIDType"/>
        <xsd:element name="messageid" type=
        "MessageIDType"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="CertificateMessage">
      <xsd:annotation>
        <xsd:documentation>
          A node can send a certificate encapsulated in
          a "certificate" message. It's not only the
          owner of
          the certificate who can send such a message.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="sender"      type=
        "UserIDType"/>
        <xsd:element name="receiver"    type=
        "UserIDType"
          minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="messageid"   type=
        "MessageIDType"/>
        <xsd:element name="certificate" type=
        "CertificateType"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="GetKeyMessage">
      <xsd:annotation>
        <xsd:documentation>
          A node may request a channel key for a given closed channel
          through a "getkey" message.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="sender"      type=
        "UserIDType"/>
        <xsd:element name="receiver"    type=
        "UserIDType"/>
        <xsd:element name="messageid"   type=
        "MessageIDType"/>
        <xsd:element name="channel"     type=
        "ChannelNameType"/>
        <xsd:element name="channelid"     type=
        "ChannelIDType"/>
```

```
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="KeyMessage">
      <xsd:annotation>
        <xsd:documentation>
          A node can send a channel key encapsulated in a
          "key"
          message. The actual data part of the key has to be encrypted
          with the according receiver's public key. It is the duty
          of
          the sender of a "key" message to ensure that the
          public key
          really belongs to the receiver and that the receiver is
          really a member of the channel.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="sender"    type=
        "UserIDType"/>
        <xsd:element name="receiver"  type=
        "UserIDType"/>
        <xsd:element name="messageid" type=
        "MessageIDType"/>
        <xsd:element name="channel"   type=
        "ChannelNameType"/>
        <xsd:element name="channelid"     type=
        "ChannelIDType"/>
        <xsd:element name="cipher"    type=
        "CipherType"/>
        <xsd:element name="key"       type=
        "xsd:base64Binary"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="RoutingMessage">
      <xsd:annotation>
        <xsd:documentation>
          Routing messages are exchanged on links to form a converging
          knowledge on the network topology on every node.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="messageid"   type=
        "MessageIDType"/>
        <xsd:element name="destination" type=
        "DestinationType"
          minOccurs="0" maxOccurs="unbounded"/>
```

```
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="MessageMessage">
      <xsd:annotation>
        <xsd:documentation>
          The "message" message carries the actual content of
          the chat
          message in its text element. If the iv attribute is present
          it is a message of a closed channel and uses the
          initialization
          vector given by the iv attribute.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="sender"    type=
        "UserIDType"
          minOccurs="0"/>
        <xsd:element name="receiver"  type=
        "UserIDType"
          minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="messageid" type=
        "MessageIDType"
          minOccurs="0"/>
        <xsd:element name="timestamp" type=
        "xsd:dateTime"/>
        <xsd:element name="channel"   type=
        "ChannelNameType"
          minOccurs="0"/>
        <xsd:element name="channelid"   type=
        "ChannelIDType"
          minOccurs="0"/>
        <xsd:element name="text">
          <xsd:complexType>
            <xsd:simpleContent>
              <xsd:extension base="xsd:string">
                <xsd:attribute name="iv" type=
                "xsd:base64Binary"/>
              </xsd:extension>
            </xsd:simpleContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="attachment" type=
        "AttachmentType"
        minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
```

```
    <xsd:complexType name="ObscureMessage">
      <xsd:annotation>
        <xsd:documentation>
          The "obscure" is created by a user who intends to
          emit a
          "message" message but remain anonymous as the sender
          of the
          message. The "text" payload is in turn an
          "obscure" message
          or a "message" message after decryption.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="receiver"  type=
        "UserIDType" />
        <xsd:element name="messageid" type=
        "MessageIDType"
          minOccurs="0"/>
        <xsd:element name="text"      type=
        "xsd:string" />
      </xsd:sequence>
    </xsd:complexType>


  </xsd:schema>
```

6. Normative References

[ITUX667] International Telecommunication Union (ITU-T), "Information technology - Open Systems Interconnection - Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components", Rec X.667, September 2004.

[RFC768] Postel, J., "User Datagram Protocol", RFC 768, August 1980.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, BCP 14, March 1997.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996.

[RFC2234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[XML] Bray, T., Paoli, J., Sperberg-McQueen, C., and E. Maler, "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C Recommendation 1, October 2000.

Authors' Addresses

Frank Strauss
TU Braunschweig
Muehlenpfordtstrasse 23
38106 Braunschweig
Germany

Phone: +49 531 391-3268
Email: strauss@ibr.cs.tu-bs.de
URI: http://www.ibr.cs.tu-bs.de/

Stefan Ransom
Universitaet zu Luebeck
Ratzeburger Allee 160
23538 Luebeck
Germany

Phone: +49 451 500-5385
Email: ransom@itm.uni-luebeck.de
URI: http://www.itm.uni-luebeck.de/

Sven Lahde
TU Braunschweig
Muehlenpfordtstrasse 23
38106 Braunschweig
Germany

Phone: +49 531 391-3264
Email: lahde@ibr.cs.tu-bs.de
URI: http://www.ibr.cs.tu-bs.de/

Oliver Wellnitz
TU Braunschweig
Muehlenpfordtstrasse 23
38106 Braunschweig
Germany

Phone: +49 531 391-3266
Email: wellnitz@ibr.cs.tu-bs.de
URI: http://www.ibr.cs.tu-bs.de/

Full Copyright Statement

Intellectual Property

Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).