Technical Report RT/21/2010

Adaptable consistency requirements for efficient large-scale multi-user chat

Pedro Alves
INESC-ID/IST/Opensoft
pedro.h.alves@ist.utl.pt

Paulo Ferreira
INESC-ID/IST
paulo.ferreira@inesc-id.pt

Mar 2010
Resumo

Multi-user chat (MUC) applications pose a challenge to developers concerning scalability and efficient use of network bandwidth, due to a potential large number of users exchanging lots of small messages in real-time. Also, sometimes users cannot cope with the overwhelming number of received messages thus, client applications have to provide filtering mechanisms to improve user experience. For example, Joom\(^1\) is a brainstorming-oriented chat application which allows the user to set an active topic, effectively filtering the whole conversation for only the messages belonging to a specific topic. Most current MUC protocols are based on broadcasting every message to every user in the chat session, but if the clients have some form of filtering, this leads to inefficiency as network bandwidth and server resources are wasted with unnecessary propagation of messages which will be filtered anyway. We propose an extension to a well-known MUC protocol, the XEP-045 extension of the XMPP protocol, which, by attaching a special component to the server, will effectively filter the messages before they are broadcast, according to client consistency requirements. These requirements are specific for each client connected to the server and can change during the session lifetime. Since our implementation is an extension to the standard specification, it can be easily deployed on a wide range of available server and client implementations.

\(^1\)Available at http://code.google.com/p/joom
Adaptable consistency requirements for efficient large-scale multi-user chat

Pedro Alves, Paulo Ferreira
Instituto Superior Técnico, Universidade Técnica de Lisboa

1 Abstract

Multi-user chat (MUC) applications pose a challenge to developers concerning scalability and efficient use of network bandwidth, due to a potential large number of users exchanging lots of small messages in real-time. Also, sometimes users cannot cope with the overwhelming number of received messages thus, client applications have to provide filtering mechanisms to improve user experience. For example, Joom\(^1\) is a brainstorming-oriented chat application which allows the user to set an active topic, effectively filtering the whole conversation for only the messages belonging to a specific topic. Most current MUC protocols are based on broadcasting every message to every user in the chat session, but if the clients have some form of filtering, this leads to inefficiency as network bandwidth and server resources are wasted with unnecessary propagation of messages which will be filtered anyway. We propose an extension to a well-known MUC protocol, the XEP-045 extension of the XMPP protocol [10], which, by attaching a special component to the server, will effectively filter the messages before they are broadcast, according to client consistency requirements. These requirements are specific for each client connected to the server and can change during the session lifetime. Since our implementation is an extension to the standard specification, it can be easily deployed on a wide range of available server and client implementations.

2 Introduction

Multi-user chat applications allow a group of geographically dispersed people to communicate with each other, typically by typing text that is broadcast to everyone in the group. Until recently, these applications have been mainly used for entertainment purposes such as casual chatting, meeting new friends, etc, however there is now a growing trend for adopting these collaborative tools in the enterprise context, for professional reasons [17]. The paradigm in which everyone works in their company’s headquarters is slowly shifting to the office-less company, where employees rarely meet physically with each other, rather working from home, some client installations or even a coffee-shop. In addition, organizations are adapting their structures to incorporate world-wide design and

\(^{1}\) Available at http://code.google.com/p/joom
manufacturing planning teams [7]. This new paradigm is supported by wide Internet availability and a myriad of collaborative tools such as email, groupchat, video-conferencing or virtual meeting rooms.

Effective collaboration and knowledge sharing on distributed teams is only possible if there is a supporting infrastructure allowing (at least) these two features:

- **Direct communication** - It must be possible to ask questions to the team and to answer and discuss these questions, until the team reaches an agreement. It is important to notice that these messages may require immediate attention as the asker may need quick feedback in order to continue what she is doing.

- **Context awareness** - It must be possible, for every member of the team, to become aware of what is going on: who belongs to the team, who is online, who is working on what, who is responsible for what, etc. [5]. This group awareness is essential. [1] refers that surreptitious monitoring has been found in many settings to be the basis for learning and knowledge sharing. [14] found that social processes such as informal coordination and ability to resolve intragroup conflicts accounted for 25 percent of the variations on software development quality. Unlike direct communication in which you are actively seeking a specific information, here you are just watching passively what is going on in the chat room. Also, these context messages don’t require immediate attention by the group because they are just informative.

Typically, multi-user chat applications provide both features but don’t distinguish them, processing every message the same way, be it direct communication or context awareness related. This is undesirable for two main reasons: first, it forces every participant to analyze each incoming message to decide if it must be acted upon or if it is just informative; second, it consumes unnecessary server resources and bandwidth propagating every message in real-time when some messages could be just informative and wouldn’t need immediate delivery.

We tried to overcome these limitations by adapting to this particular problem two well-known techniques in other fields: relaxed consistency for replicated data [13] and content-based filtering for publish-subscribe systems [8]. In this paper, we propose an adaptable consistency requirements specification that can be plugged into the MUC protocol which defines bounded lag parameters for message propagation. The main idea can be summarized as propagating important messages immediately and delaying not so important messages, based on client parameterization.

We decided to extend the XEP-045 extension of the XMPP protocol as it is probably the most widely used non-proprietary MUC protocol, used by Google on its chat application (gtalk) among others. There are multiple client applications available using this protocol, as well as server implementations. According to the published specification [11], it defines an XMPP protocol extension for multi-user text chat, whereby multiple XMPP users can exchange messages in the context of a room or channel, similar to Internet Relay Chat (IRC). In ad-
Adaptable consistency requirements for efficient large-scale multi-user chat

The protocol defines a strong room control model. The protocol doesn’t specify any filtering mechanism (which can be considered a specific form of consistency requirements) for exchanged messages, although it’s easy to extend it in order to include this kind of meta-information, since the underlying message format is plain XML. This is precisely how Joom implemented the active topic concept - by extending the message format to include this information.

Besides filter triggering, there is another event that influences the decision on immediate or delayed propagation: the possible relation between exchanged messages. In these applications it makes sense to group certain messages together (e.g. direct replies) [15], and these groups must be carefully processed when the server is under relaxed consistency requirements. In fact, there is a causal relation in many of these messages, therefore we need to guarantee their total ordering [6] across all participants. For example, we have to make sure that users don’t see a reply before or without the original message. We also have to make sure that the asker sees the replies to her question as soon as possible.

3 Related Work

The scalability of large MUC systems can be achieved by using well-known techniques to reduce the number of exchanged messages, either by discarding irrelevant messages as publish-subscribe systems do, or by postponing its dissemination as relaxed consistency systems do. If we want to preserve awareness in MUC applications without compromising scalability, we must use a combination of both techniques described in this section.

3.1 Publish-Subscribe Systems

The publish-subscribe paradigm [3] is receiving increased attention for the loosely coupled form of interaction it provides in large scale settings. It consists of three components: publishers, who generate and feed the content into the system, subscribers, who receive content based on their interests in a topic or pattern, and an infrastructure that distributes events matching subscriber interests with publishers content. There are two kinds of matching systems: subject-based and content-based.

In subject-based systems [8], there are some predefined subjects also known as topics or channels to which both publishers and subscribers can connect. For example, in a subject-based system for stock trading, a participant could select one or two stocks and then subscribe based on stock name if that were one of valid channels.

Content-based systems [8] are much more flexible at the expense of a more complex matching algorithm - they enable subscribers to issue sophisticated queries or predicates that act on a message-by-message basis. In this case, a participant could decide to receive stock information under much more specific
Adaptable consistency requirements for efficient large-scale multi-user chat
conditions such as \{ \text{price}(15,20), \text{Earning-Per-Share} > 0.5 \}. Several group communication applications have been developed using publish-subscribe systems such as XGSP [18]. In particular, subject-based systems fit well in group-chat applications like IRC, since participants may join \textit{rooms} or \textit{channels} from a pre-defined list, effectively aggregating messages by topic. However, there are two important drawbacks of publish-subscribe systems when applied to group communication. First, messages are always delivered as soon as possible, even though the user doesn’t require this immediacy. This is specially significant if the participant is receiving messages in a constrained device such as a cell-phone or the volume of delivered messages is too high for the current network conditions. Second, the participant is completely oblivious of what is going on other channels, rendering impossible any visualization of context awareness information which, as explained before, is crucial to effective knowledge sharing.

3.2 Optimistic Replication

Optimistic replication algorithms increase availability and scalability of distributed data sharing system by allowing replica contents to diverge in the short term [13]. If we consider exchanged messages in MUCs as a form of distributed shared data (albeit short-lived) and that this kind of applications often tolerate some lag on message delivery, it makes sense to study how optimistic replication algorithms can be used to implement group real-time communication applications.

Actually, the oldest optimistically replicated service is a group communication tool called Usenet [16], first deployed in 1979. Usenet is a multi-server system that lets any user post articles to any server, which periodically pushes the newest articles to neighboring servers. Eventually, those articles will reach every server in the world, albeit they can take as long as a week to show up in every Usenet client. This temporal variability is a reasonable cost to pay for its excellent availability.

More recently, [19] describes how TACT, a framework that enforces arbitrary consistency bounds among replicas, was used to implement a Bulletin Board application similar to Usenet. The authors refer to the importance of maintaining causal and/or total order among messages posted at different replicas as well as guaranteeing an upper temporal limit for missing messages in a given replica. However, TACT doesn’t allow consistency bounds to be based on message content: every message is distributed the same way independently of its subject.

4 Design

This paper proposes a relaxed consistency model for MUC message propagation, bounded by three metrics: \textit{Filter}, \textit{Time} and \textit{Volume}:

- \textbf{Filter} - Specifies a list of queries or predicates that are applied to every message in order to decide if it must be propagated immediately or if it can be
postponed. The filter principle is similar to content-based publish-subscribe systems, in which subscribers define their subscriptions issuing predicates to receive only the information they need [3][8]. Unlike publish-subscribe systems, the filtered-out messages are not discarded, only postponed until other requirement specifications like time and volume reach a certain threshold.

- **Time** - Specifies the maximum time a message can be retained in the server. After this time, the message has to be propagated to the client.
- **Volume** - Specifies the maximum volume of retained messages in the server. When the volume of messages reaches this threshold, those messages are immediately propagated to the client.

We also take into account possible message correlation, such as message replies. Thus, our proposal also guarantees causal delivery [6] so that: i) for any reply \( r_n \) in a chain of replies to a certain message \( m \), \( r_n \) will not be propagated before \( m \), even if it is activated by a filter for immediate delivery; ii) for any message \( m_p \), sent by participant \( p \), all replies to that message are instantly sent to \( p \) no matter what the current consistency requirements are.

The remainder of this section starts by describing the protocol upon which the relaxed consistency model is implemented. Next, we explain how the client defines its consistency requirements. Afterwards, we introduce the concept of active topic as an example of a filter. We end up illustrating a possible message workflow that applies these concepts to a real-world usage scenario.

### 4.1 XMPP - Extensible Messaging and Presence Protocol

The core XMPP specification is defined in RFC 3920 [10] and defines a XML stream between two entities over a network [12]. A XML stream is an open-ended XML document that is built over time by sending XML elements over the stream. It usually follows a client-server model similar to email, in the sense that the sender communicates with its assigned server that forwards the message to the recipient server that than delivers the message to the final recipient client. Hence, the protocol defines client-server streams as well as server-server streams. If the sender and the recipient are connected to the same server, the server forwards their messages directly, without needing any server-server connection. The server with which each entity communicates is the server associated with the domain of the client identification. Again, this is similar to email systems: if my client identification is "pedro.alves@host.com", than my XMPP server is obtained through the DNS entry assigned to "host.com". All these connections are long-lived TCP [9] channels and are subject to previous authentication between both end-points. The channel is usually encrypted using TLS [2] but this can be negotiated when the stream is created.

The first child-level elements that are sent over these streams are called stanzas and fall into these 3 types, each with different semantics:

- `<message/>` - This stanza is used for unidirectional information push, from one entity to another. This is similar to an email.
– `<presence/>` - This stanza is broadcast to inform multiple entities of the status and network availability of a particular entity. For example, when I start an Instant Messaging application, a `<presence/>` stanza is sent to everyone in my contact list alerting that I am now online and able to chat.

– `<iq/>` *(Info/Query)* - This stanza is a request-response mechanism, similar to HTTP and is normally used for exchanging meta-information such as getting the list of available services, setting up channel parameters, etc. Unlike a `<message/>` stanza, the entity that sends an IQ stanza must always receive a reply.

These 3 types have proven sufficient for most kinds of real-time communications because the stanzas are easily extended by including child elements that may be qualified by any XML namespace. Throughout this paper, we show how we have extended some stanzas to include consistency and filtering information.

Some of these XMPP extensions are so commonly used that they have been standardized through the Jabber Software Foundation (JSF) [11] and include file-transfer, XHTML-formated messages, service discovery, publish-subscribe, HTTP binding among others. Of particular interest to this paper is the Multi-User chat extension which has been defined in XEP-045 [11] and is now implemented in a wide variety of server and client applications.

Although the main adoption driver of XMPP has been Instant Messaging, its extensibility and ease of development makes it suitable to diverse applications such as large-infrastructures monitoring or real-time financial notification services.

### 4.2 Setting up consistency requirements

In spite of the large number of predefined available extensions, there was none for defining consistency requirements, so we had to create our own extension (stanza). This stanza will be typically sent by the client just after joining a MUC session with the initial definition, but can be sent anytime the client wants to change its consistency requirements.

We have decided to extend the IQ stanza *(Info/Query)* with a `<consistency-requirements/>` child element, featuring the three bounding metrics:

```xml
<iq to="chat@conference.jabber.org" type="set">
  <consistency-requirements xmlns="http://joom.com/extensions">
    <filter>
      <filter-entry element="topic" value="xmpp" active="true"/>
      <filter-entry element="topic" value="android" active="true"/>
      <filter-entry element="topic" value="chess" active="true"/>
    </filter>
    <time>60</time>
    <volume>20</volume>
  </consistency-requirements>
</iq>
```

**Listing 1.1. Initial Consistency Requirements stanza**

2 The Info/Query stanza provides a structure for request-response interactions, similar to GET/POST methods from HTTP.
In this example, the client wants to instantly receive any messages containing the XML child element *topic* with the value *xmpp*, *android* or *chess*. All other messages can be retained by the server until they are 60 seconds old or until there are 20 pending messages. Notice that these consistency requirements affect only `<message />` stanzas. The `<presence/>` and `<iq/>` stanzas will be processed normally.

By extending the standard IQ stanza, we assure that any XMPP server will be able to receive this message, even though only those using our component will know what to do with this message. Notice the *active* attribute of the *filter-entry* element, which can be used to turn off a certain filter. This is because any client wanting to change its consistency requirements can do so by issuing another stanza with only the changed requirements. For example, after sending the initial requirements on Listing 1, suppose the client is no longer interested on the topic *xmpp* and is now interested in the topic *pubsub*:

```xml
<iq to="chat@conference.jabber.org" type="set">
  <consistency-requirements xmlns="http://joom.com/extensions">
    <filter>
      <filter-entry element="topic" value="xmpp" active="false"/>
      <filter-entry element="topic" value="pubsub" active="true"/>
    </filter>
  </consistency-requirements>
</iq>
```

Listing 1.2. Updated Consistency Requirements stanza

Since the `<time>` and `<volume>` elements are absent from the stanza, the server will maintain the previous values (in this case, this would be `time=60` and `volume=20`). Also, applying the same reasoning, the topics *android* and *chess* will remain active.

### 4.3 Filter example - Associating a message with a topic

One of the big advantages of content-based filtering is its flexibility. As long as the specification language expressiveness is rich enough, any filter can be defined. In our case, we can define filters based on the presence of a certain XML child element in the message and on the value of that element. Notice that there are no limitations to the number of defined filters. Nonetheless, we illustrate the use of these filters with the concept of *active topic*. Joom implemented this concept to help participants of a brainstorming session focus on only one topic at a time. Participants choose their active topic independently and any message they send is automatically associated with that topic. At any time they can change their active topic, but they can only have one active topic, so their previous active topic is *disactivated*. Again, this is an usability requirement of this particular example, not a limitation of our extension.

Again, since there is no XMPP standard extension to associate messages with a topic or tag we had to define our own. We extended the `<message/>` stanza to

```
<iq to="chat@conference.jabber.org" type="set">
  <consistency-requirements xmlns="http://joom.com/extensions">
    <filter>
      <filter-entry element="topic" value="xmpp" active="false"/>
      <filter-entry element="topic" value="pubsub" active="true"/>
    </filter>
  </consistency-requirements>
</iq>
```

Listing 1.2. Updated Consistency Requirements stanza

Since the `<time>` and `<volume>` elements are absent from the stanza, the server will maintain the previous values (in this case, this would be `time=60` and `volume=20`). Also, applying the same reasoning, the topics *android* and *chess* will remain active.

### 4.3 Filter example - Associating a message with a topic

One of the big advantages of content-based filtering is its flexibility. As long as the specification language expressiveness is rich enough, any filter can be defined. In our case, we can define filters based on the presence of a certain XML child element in the message and on the value of that element. Notice that there are no limitations to the number of defined filters. Nonetheless, we illustrate the use of these filters with the concept of *active topic*. Joom implemented this concept to help participants of a brainstorming session focus on only one topic at a time. Participants choose their active topic independently and any message they send is automatically associated with that topic. At any time they can change their active topic, but they can only have one active topic, so their previous active topic is *disactivated*. Again, this is an usability requirement of this particular example, not a limitation of our extension.

Again, since there is no XMPP standard extension to associate messages with a topic or tag we had to define our own. We extended the `<message/>` stanza to

```
<iq to="chat@conference.jabber.org" type="set">
  <consistency-requirements xmlns="http://joom.com/extensions">
    <filter>
      <filter-entry element="topic" value="xmpp" active="false"/>
      <filter-entry element="topic" value="pubsub" active="true"/>
    </filter>
  </consistency-requirements>
</iq>
```

Listing 1.2. Updated Consistency Requirements stanza

Since the `<time>` and `<volume>` elements are absent from the stanza, the server will maintain the previous values (in this case, this would be `time=60` and `volume=20`). Also, applying the same reasoning, the topics *android* and *chess* will remain active.
include a `<topic>` child element to achieve this association, with the following format:

```
<message to="chat@conference.jabber.org">
  <body>Hi there! #xmpp</body>
  <topic xmlns="http://joom.com/extensions">xmpp</topic>
</message>
```

**Listing 1.3.** Message stanza with the non-standard topic element

Notice that we also append the topic to the body itself, following an hashtag convention. This way, every standard XMPP client can participate in the MUC session and show the associations between messages and topics. Also, the XMPP servers which don’t recognize the `<topic>` element will simply ignore it.

### 4.4 Usage scenario - Messages workflow

The typical message flow starts with a presence stanza sent by the client to the chat room server. If the client is allowed to join this room, it will start receiving all the messages sent to this room by other clients. Next, the client can send a `<consistency-requirements>` stanza to specify which messages it requires to receive, based on some filter defined by its application. In Figure 1, we can see in step 2 that the client is telling the server to filter only those messages whose topic is `gossip`. In steps 4 and 5 she receives messages that fit within that topic just after they were sent (actually, the client always receives her own messages immediately, regardless of the consistency requirements). In step 6, the client receives a message that has been sent a while ago. It was retained at the server because it didn’t comply with the filter setup on step 2 but can now be sent because it reached the defined time threshold. Since this message was delayed, its payload includes an extra `delay` element. The `delay` is an optional element defined by the XEP-0203 extension (Delayed Delivery)[11], to communicate the fact that an XML stanza has been delivered with a delay, for example because a message has been stored on a server while the intended recipient was offline or because a message is contained in the history of a multi-user chat room. Although the use cases described in this specification don’t include our “temporarily retained” status, it fits well in our consistency protocol and it is recognized by most xmpp clients.

In Figure 2, we illustrate a scenario in which two clients share a MUC session. After joining the room, they both setup the same active topic `movies` so they both receive messages concerning that topic. In step 7, Alice changes the active topic and starts talking about `cooking`. We can see in step 9, that Alice immediately receives her message from step 8. Rabbit will only receive her message after a while, in step 12, with a `delayed` indication. Step 10 illustrates a special case which bypasses all the filters that may exist: Rabbit is replying to a specific...
Fig. 1. Example of xmpp messages exchanged between the client (Alice) the chat room server, during a session.
message sent by Alice, so it must be propagated immediately, otherwise it may create confusion among the users involved.

5 Implementation

We decided to develop ReConMUC (Relaxed Consistency MUC) as a standard xmpp server component, compliant with XEP-0114 [11]. This means that it can be easily deployed in many existing xmpp server implementations, without requiring any change in their code or even recompilation.

ReConMUC extends the standard MUC component with consistency settings for each registered client in the session. These consistency settings define the Filters for each client, as well as the maximum tolerable Time and Volume. These settings can be updated at any time through a &lt;consistency-requirements&gt; message, as described in section 5.1.

Listing 4 shows the pseudo-code for processing received messages. Every time a message is received, the MUC server iterates through all participants in the chat room to decide which ones will need immediate message delivery. In line 4, we can see that it instantly sends the message back to the original sender. This is necessary because the client application of the sender should only show the message after it has been sent and received, to avoid coordination problems, even though it obviously knows its contents. Line 5 checks if the message is a
reply to the current user in the iteration. In that case, it also sends the message right away because we want the user to see as soon as possible any direct replies to the messages she sent. Next, it will apply all the active filters for each user. If the message content matches the filter definition than it will be propagated right away (line 12), but first the server will check if this message belongs to some thread (here, thread means a group of related chat messages, like a reply chain), to guarantee that the recipient never sees a reply before the original message (lines 10-11), preserving causal order. Finally, if the message isn’t fit for immediate propagation, it is appended to a FIFO list of retained messages. This list is periodically processed by the round triggered function (see Listing 5). This function is called by a scheduler and is responsible for checking the time and volume parameters.

```c
function message_received(m):
    foreach u in users:
        if m.from == u:
            send(m, u) #Send imediately for the message origin
        elseif m.is_reply_to_user(u):
            send(m, u) #Send imediately replies to origin
        else:
            foreach filter in u.filters:
                if filter.active(m):
                    if mbelongs_to_thread():
                        send_all_previous_in_thread(u,m.thread)
                    send(m, u)
                    break
            if m.was_not_sent():
                retainedmsgs[u].append(m)

function send_all_previous_in_thread(u, thread_id):
    foreach m in retainedmsgs[u]:
        if m.thread == thread_id:
            send(m, u)
    retainedmsgs[u].remove(m)
```

Listing 1.4. Pseudo-code for processing received messages

```c
function round_triggered():
    foreach u in users:
        if (now - u.last_sent) > kt:
            send_all_and_remove_from_retained_msgs(retained_msgs[u])
    if retained_msgs[u].size > ks:
        send_all_and_remove_from_retained_msgs(retained_msgs[u])
```

Listing 1.5. Pseudo-code for round triggered function

Notice that some steps were omitted to simplify the code. For example, all manipulations of the retained_msgs array must be synchronized since this structure is accessed concurrently by different threads.

6 Evaluation

One of the premises behind this work was that multi-user chat protocols waste network bandwidth by broadcasting messages as quickly as possible, even when that urgency is not required by the end users. We didn’t have a large number of users available to join a test chat room, so we used publicly available chat (IRC) logs from very active channels like #ubuntu. Typically, chat users join and watch multiple rooms at the same time, each one on its own window. We

---

6 Available at http://irclogs.ubuntu.com
tried to replicate this behaviour by setting up only one XMPP MUC room where all messages were associated with a topic (see section 4.3) representing the IRC channel which they were coming from.

We developed an application which parses IRC log files, searching for users, messages and replies. This application then creates a thread for each user, responsible for keeping a connection with the XMPP server and joining the room. Afterwards, each message from the log file is dispatched to the corresponding sender thread which then sends it to the server using its connection. Each user has a current topic (analogous to the IRC window with the current focus) that changes if she sends a message with other topic. As explained in section 5, the replies are immediately sent to the origin.

The server is an Intel Core 2 Duo 2.5GHz machine with 4Gb RAM (Windows Vista) and is monitored in two dimensions: outbound network consumption and memory usage. The client is deployed in a Intel Core 2 Duo 2GHz machine with 1Gb RAM (Mac Os X). These two machines are connected through a LAN.

The experiment consists on feeding the server with two IRC log files, during 5 minutes, resulting in aprox. 1000 messages being broadcast to 150 participants. In order to understand the impact of the ReConMUC extension, we setup this experiment to run under different scenarios. In the first scenario ReConMUC is disabled, so every message is broadcast immediately to every participant, as traditional MUC systems do. In the other scenarios, ReConMUC is enabled but with different average consistency settings. The consistency settings are defined by each participant’s thread through a random function around a globally defined parameter, that is different for each scenario. For example, if the parameter is $t=30$, than each client will calculate a random value between 0 and 60 for $t$, resulting on the global average value of 30. We think this is a more realistic scenario than having the exact same parameters for every participant. Notice that every participant received 1000 messages in all scenarios - what differs is not what she received but when and how she received those messages.

### 6.1 Outbound network consumption

Before running the experiment, we tried to reduce network consumption by using one of the available mechanisms in the XMPP protocol - stream compression (XEP-0138 [11]). All the major server and client implementations support this extension, using for example the ZLIB algorithm to compress all the traffic between the client and the server. Our experiments show a 5x reduction in network consumption when stream compression is enabled. All subsequent measurements were made with stream compression turned on.

Than we ran the experiment while monitoring network usage under different scenarios, achieving the results shown in table 1.

The first surprising result was the network usage increase when we enabled ReConMUC without aggregating messages. This is easily explained by the length increase in all delayed messages, which had to include (verbose) delay information (e.g. `<delay xmlns='urn:xmpp:delay' from='juliet@capulet.com/balcony' stamp='2002-09-10T23:41:07Z'/>`). This increase is sufficient to masquerade the
Adaptable consistency requirements for efficient large-scale multi-user chat

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Outbound network usage</th>
<th>Consumed bandwidth</th>
<th>Relative Consumed Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without ReConMUC</td>
<td>14.89 Mb</td>
<td>50.82 Kb/s</td>
<td>100%</td>
</tr>
<tr>
<td>With ReConMUC (without aggregation) ((t = 30, v = 25))</td>
<td>15.86 Mb</td>
<td>54.14 Kb/s</td>
<td>107%</td>
</tr>
<tr>
<td>With ReConMUC ((t = 15, v = 12))</td>
<td>10.9 Mb</td>
<td>37.2 Kb/s</td>
<td>73%</td>
</tr>
<tr>
<td>With ReConMUC ((t = 60, v = 50))</td>
<td>9.83 Mb</td>
<td>33.55 Kb/s</td>
<td>66%</td>
</tr>
<tr>
<td>With ReConMUC ((t = 90, v = 75))</td>
<td>9.87 Mb</td>
<td>33.69 Kb/s</td>
<td>66%</td>
</tr>
</tbody>
</table>

Table 1. Outbound network usage under different scenarios

reduction of transmitted messages. Since this is a standard XMPP extension [11], we didn’t want to modify this element to make it more concise. After that, we improved ReConMUC based on the fact that retained messages are normally delivered in batch. For example, if I have 20 retained messages in the server and my maximum tolerable volume is 20, the server will send all these messages in a burst. The problem is that the server implementation doesn’t know that and treats all those messages as individual packets. We developed the concept of a composite message (roughly following the composite pattern [4]), an aggregation of messages which the server processes as just one message. This way, the compression algorithm is applied to the whole batch and not to individual parts, and since the messages have similar information it can get high compression rates, effectively dropping network usage in approx. 1/3, as we have observed in the results. As we experimented increasingly relaxed consistency settings, we noticed a decrease in network consumption, probably because as we aggregate more messages, the compression algorithm becomes more effective because of the increased redundancy.

6.2 Server memory usage

Typically, relaxed consistency protocols assume a tradeoff between two dimensions. In this case, we assumed this would be between network consumption and memory usage - we could reduce network consumption at the expense of server memory used to store retained messages. In fact, we observed that the average number of retained messages per user increased as we relaxed consistency requirements:

In order to understand the real impact of this numbers, we used a profiler to keep track of memory usage in the server. The results were somehow surprising - the real memory usage increase was negligible (aprox. 4 Mb). Even the difference with and without the extension was minimal. This is only possible because we used a feature of the MUC extension specification that allows anyone who joins a room to receive the previous messages exchanged in that room (the discussion history). Although this can be turned off, it is usually turned on to reduce the
Adaptable consistency requirements for efficient large-scale multi-user chat

Consistency settings | Avg. retained msgs/user
--- | ---
\(t = 15, v = 12\) | 9
\(t = 30, v = 25\) | 14
\(t = 60, v = 50\) | 35
\(t = 90, v = 75\) | 47

Table 2. Avg. retained msgs/user

Table 3. Retained msgs/user (chart)

sense of lostness of newcomers to a discussion. Our implementation of the algorithm doesn’t store actual retained messages but only pointers to those messages in the discussion history (which exists anyway, with or without the extension). That is, even in the most memory hungry observed scenario, we are only storing a list of 47 pointers for each user.

7 Conclusion and Future Work

With the increasing demand for collaboration tools that assist dispersed teams, specially those that try to maintain context awareness, we predict an explosion in the number of messages broadcast by these systems, many of which don’t require immediate attention of their recipients. In this paper, we propose a relaxed consistency model for the propagation of multi-user chat messages based on three dimensions: time, volume and content filters. We can tweak these parameters for each client, based on its current context, device capabilities, network availability, etc. This model was implemented on top of XMPP, an open and widely used messaging protocol and may, therefore, be easily deployed on any of the available server implementations in the internet. Our evaluation based on real chat logs demonstrates that, by aggregating and compressing retained messages, we can actually significantly reduce network usage without increasing server memory consumption. We are investigating other consistency dimensions such as the social network of the participants - maybe we can relax consistency as we increase the social distance between the sender and the receiver. Also, we plan to study ways of reducing inbound traffic to the server, which can be problematic if we attach non-human clients (bots) to a chat session, providing real-time information.

References

Adaptable consistency requirements for efficient large-scale multi-user chat


