ReConMUC - Adaptable Consistency Requirements for Efficient Large-scale Multi-user Chat

Pedro Alves
INESC-ID / Technical University of Lisbon / Opensoft
Rua Joshua Benoliel, 1, 4C, 1250 Lisboa
pedro.alves@opensoft.pt

Paulo Ferreira
INESC-ID / Technical University of Lisbon
Rua Alves Redol, 9, 1000 Lisboa
paulo.ferreira@inesc-id.pt

ABSTRACT
Multi-user chat (MUC) applications raise serious challenges to developers concerning scalability and efficient use of network bandwidth, due to a large number of users exchanging lots of messages in real-time.

We propose a new approach to MUC message propagation based on an adaptable consistency model bounded by three metrics: *Filter*, *Time* and *Volume*. In this model, the server propagates some messages as soon as possible while others are postponed until certain conditions are met, according to each client consistency requirements. These requirements can change during the session lifetime, constantly adapting to each client's current context.

We developed a prototype called ReConMUC (Relaxed Consistency MUC) as an extension to a well-known MUC protocol, which, by attaching a special component to the server, filters messages before they are broadcast, according to client consistency requirements.

The performance results obtained show that ReConMUC effectively reduces the server outbound bandwidth, without significant increase in memory and CPU usage, thus improving scalability.

Author Keywords
multi-user chat, publish-subscribe, consistency requirements, groupware awareness

ACM Classification Keywords
H.4.3 Information Systems Applications: Communications Applications

General Terms
Algorithms, Experimentation, Human Factors, Measurement, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CSCW 2011, March 19–23, 2011, Hangzhou, China.
Copyright 2011 ACM 978-1-4503-0556-3/11/03...$10.00.

INTRODUCTION
Multi-user chat (MUC) 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 [31]. The paradigm in which everyone works in their company's headquarters is slowly shifting to the officeless 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 structure to incorporate world-wide design and manufacturing planning teams [20]. 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 [2].

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. Messages exchanged 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 team member, 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. [15]. This group awareness is essential, as an understanding of the activities of others provides a context for our own activity [8]. Bjerrum [4] refers that surreptitious monitoring has been found in many settings to be the basis for learning and knowledge sharing. Sawyer [26] found that social processes such as informal coordination and ability to resolve intragroup conflicts accounted for 25% of the variations on software development quality. Unlike direct communication in which you are actively seeking a specific information, in this case the user is just watching passively what is going on in the chat room. Such context

messages do not require immediate attention by the group because they are just informative.

Typically, MUC applications provide both features but do not distinguish them, processing every message the same way, be it direct communication or context awareness related. This is undesirable for two main reasons: i) it forces every participant to analyze each incoming message to decide if it must be acted upon or if it is just informative; ii) propagating every message in real-time consumes unnecessary server resources and bandwidth as some messages are just informative and do not require immediate delivery.

We address these limitations by adapting to this particular problem two well-known techniques from other fields: relaxed consistency for replicated data [25] and content-based filtering for publish-subscribe systems [21]. Thus, we propose an adaptable consistency model for MUC propagation in order to increase scalability and performance. Consistency requirements specification are plugged into the MUC protocol defining bounded lag parameters for message propagation. Intuitively, the main idea is propagating important messages immediately and delaying not so important messages, based on client parameterization. The parameters are *Filter* (based on the message content), *Time* (based on the message age) and *Volume* (based on the number of messages retained on the server). Delayed messages (retained at the server) allow more important ones to be immediately propagated to clients. In addition, as observed experimentally, retained messages cand be aggregated and sent as one, allowing a high level of compression, thus requiring less network bandwith than current systems.

Besides filter triggering, there is another event that influences the decision on immediate or delayed message propagation: the possible relation between exchanged messages. In MUC applications it makes sense to group certain messages together (e.g. direct replies) [29]; such message groups must be carefully processed when the server is under relaxed consistency requirements. In fact, there is a causal relation among many of these messages; therefore, we need to guarantee their ordering [17] across all participants. For example, we have to make sure that users do not see a reply before the original message. We also have to make sure that the asker sees the replies to her question as soon as possible.

We implemented a prototype of our adaptable consistency model (called ReConMUC) as an extension to the XEP-045 extension of the XMPP protocol [22] given that it is probably the most widely used non-proprietary MUC protocol.[1] The XMPP protocol does not 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. In fact, this is precisely how we developed Joom,[2] a desktop brainstorming oriented chat application that allows the user to set an active topic, effectively filtering the whole conversation for only the messages belonging to a specific topic. Joom extends the standard XMPP message format to include specific information such as the active topic of the conversation.

In summary, the contributions of this work are:

- A new adaptable consistency model bounded by three metrics (filter, time and volume) for MUC message propagation. It allows users to easily define their consistency settings, recognizing that each user has different requirements for message delivery, without sacrificing general awareness of what is going on in conversations elsewhere. In spite of the large number of messages, this model ensures good performance and scalability.
- ReConMUC, a MUC prototype of the consistency model built as an extension to a well-known MUC protocol (XMPP); thus, it can be easily deployed in all XMPP server implementations.
- Joom, a brainstorming-oriented chat application, that makes use of ReConMUC to provide enhanced performance and usability to the end-user.

The remainder of this paper is organized as follows. The next section surveys relevant related work. After that, we present the adaptable consistency model, explaining how XMPP is extended to include consistency settings information. The next section describes ReConMUC implementation details. Then, we present performance results and finally we draw some conclusions.

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 these techniques, both described in this section. In addition, since MUC applications are collaborative real-time systems, we provide an overview of Computer Supported Collaborative Work (CSWC) studies related to real-time groupware performance.

Publish-Subscribe Systems

The publish-subscribe paradigm [11] is a loosely coupled form of interaction suitable for 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 [21], there are some predefined subjects also known as *topics* or *channels* to which both

[1]Used by Google and Facebook on their chat applications, among others and reaching more than 500 million users; there are multiple client applications available using this protocol, as well as server implementations.

[2]Available at http://code.google.com/p/joom

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 to them based on their name[3] if that were one of valid channels. Some examples of these systems include Isis [3], Linda [6], Herald [5] and more recently Pubsubhubbub [1] for Internet data feeds.

Content-based systems [21] 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 conditions such as {price(15,20), Earning-Per-Share>0.5}. Elvin [27] and Siena [7] are examples of such systems.

The publish-subscribe communication model has been proposed as a possible solution for large scale group collaboration [18]. In fact, several group communication applications have been developed using publish-subscribe systems such as Corona [16] and XGSP [32]. In particular, subject-based systems fit well in MUC applications like IRC, since participants may join *rooms* or *channels* from a predefined 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 does not 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 the presence of other subscribers (even for the same topic) [16] as well as not knowing what is going on other channels, rendering impossible any visualization of context awareness information which, as explained before, is crucial to effective knowledge sharing. Besides direct communication, publish-subscribe systems have also been used to propagate awareness information (e.g. Tickertape [12]).

Optimistic Replication

Optimistic replication algorithms increase availability and scalability of distributed data sharing systems by allowing replica contents to diverge in the short term [25]. 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 MUC applications.

One of the oldest optimistically replicated system is a group communication tool called Usenet [30], 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.

Hall [16] recognizes as a fundamental group collaboration requirement that different kinds of data require different levels of consistency. From his own experience implementing a collaborative system, the author asserts that a communication service must not enforce a single data consistency police for all types of data. However, he primarily addresses the reliability problem, pertaining data loss and updates ordering. For example, he describes a chat application as a system where is imperative to reliably deliver messages to all users, when in fact, certain messages (e.g. awareness) could be subject to a less restrictive model.

More recently, Yu [33] 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 [17] among messages posted at different replicas as well as guaranteeing an upper temporal limit for missing messages in a given replica. However, TACT does not allow consistency bounds to be based on message content: every message is distributed the same way independently of its subject.

Real-time Groupware Performance

The scalability of real-time distributed groupware applications has concerned the CWSC community for quite sometime [16]. For example, Gutwin [14] shows that network delays due to latency and jitter cause difficulties in group coordination, affecting the usability of collaborative distributed applications.

Smed [28] and Dyck [10] identified several networking techniques used in multi-player computer games (MCGs) which share similar demands and characteristics with general real-time groupware applications. In fact, most MCGs include some form of chat (text or audio) to help team coordination and generate the kind of bursty short messages traffic we can find in MUC applications. Also, MCGs produce different types of messages: awareness messages (e.g. nearby players movement), operations on the data model (e.g. killing a player) and explicit communication such as already mentioned chat messages. One of the most important conclusions of these studies is that although awareness messages are the most common, they are often less important than certain creation/deletion messages.

Dyck [9] provides a taxonomy of application-layer networking techniques, broken into four categories: encoding, routing, reliability and scheduling. Two of these groups assume greater relevance in the context of MUC: encoding and scheduling. Encoding aims to decrease the payload size by changing the representation of the messages, mainly using some form of compression. Scheduling techniques decide when information is sent and how it is packaged. Although most groupware applications use an event-driven

[3] In this case, each stock name constitutes a channel which users can subscribe to, thereby receiving real-time information on that particular stock.

TCP model [10], sending messages whenever an event occurs (as soon as possible), this can cause problems under network constrained environments such as WANs and mobile devices. By scheduling messages to later delivery (based on QoS requirements) and aggregating them in larger packets, applications can achieve good usability and performance even in poor network conditions.

Most groupware performance studies are related to MCGs or collaborative whiteboards where telepointer events propagation still constitute the main problem. Nevertheless, in these applications, the "immediate propagation" paradigm is still the norm, and only the recent tendency of mobile devices usage for distributed collaboration has raised some concern on these applications scalability [19].

ADAPTABLE CONSISTENCY MODEL

This paper proposes an adaptable consistency model for MUC message propagation, bounded by three metrics: *Filter*, *Time* and *Volume*:

- **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 [11, 21]. However, and most importantly, unlike publish-subscribe systems, the filtered-out messages are not discarded, only postponed at the server, until other requirement specifications (time and volume) reach a certain threshold.
- **Time** - Specifies the maximum time interval a message can be postponed (i.e. retained in the server). Once this time interval expires, the message is propagated to the client.
- **Volume** - Specifies the maximum number of retained messages in the server. When the volume of messages reaches this threshold, such messages are immediately propagated to the client.

We also take into account possible message correlation, such as message replies, message threads (specific short-lived topics within the main topic), sender's geo-location, etc. In particular, our proposal guarantees causal delivery [17] so that: i) for any reply r_i in a chain of replies to a certain message m, r_i 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 XMPP, the protocol upon which the adaptable consistency model is integrated into. Note that the consistency model here presented can be applied to other similar MUC protocols. 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-word usage scenario.

XMPP - Extensible Messaging and Presence Protocol

The core XMPP specification is defined in RFC 3920 [22] and defines an XML stream between two entities over a network for real-time communication [24]. The XML elements that are sent over these streams are called stanzas and fall into the following 3 types, each with different semantics: <**message/**> for unidirectional information push, from one entity to another; <**presence/**> for broadcasting the status and network availability of a particular entity and <**iq/**>, a request-response mechanism (similar to HTTP) normally used for exchanging meta-information. 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. In the next sections, we show how we have extended some stanzas to support our adaptable consistency model.

Some of the current XMPP extensions are so commonly used that they have been standardized through the Jabber Software Foundation (JSF) [23] and include file-transfer, service discovery, publish-subscribe among others. Of particular interest to this work is the MUC extension which has been defined in XEP-045 [23] 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 [24].

Setting Consistency Requirements

In spite of the large number of predefined available XMPP extensions, there was none for defining consistency requirements, so we had to create our own extension (stanza). This stanza is typically sent by the client to its server just after joining a MUC session with the initial consistency requirements. It is worthy to note that consistency requirements can also be sent anytime the client wants to change such requirements. These changes are sent by the client incrementally, i.e. the message contains only what has changed since the last consistency requirements message.

```
<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: Initial Consistency Requirements stanza

We have extended the <iq/> stanza with a <consistency-requirements/> child element, featuring the three bounding metrics previously presented. Listing 1 shows a case in

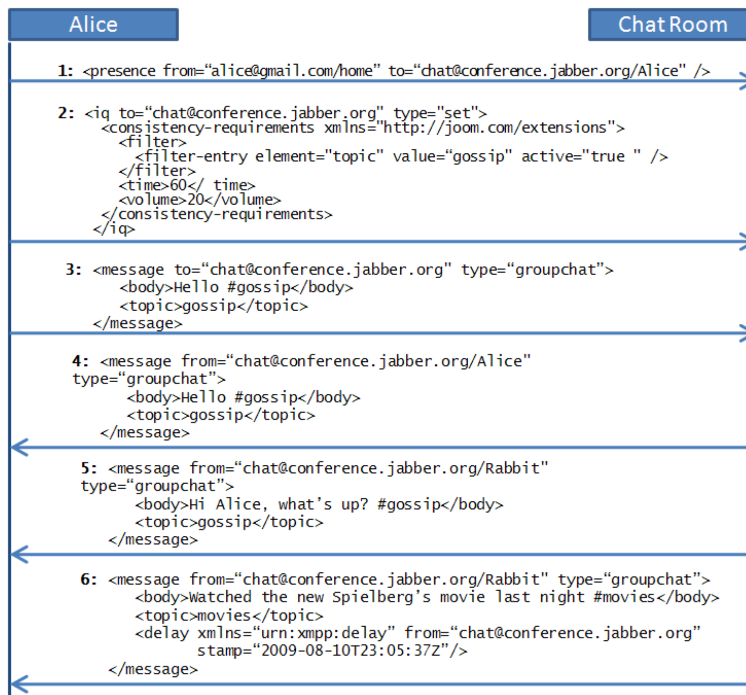

Figure 1: Messages exchanged between the client (Alice) and the chat room server.

which a client joins the chat room 'chat@conference.jabber.org' from which she will receive all exchanged messages. However, she is particularly interested in receiving, as soon as possible, any messages containing the XML child element *topic* with the values *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 are processed normally, i.e. they are immediately propagated by the server. By extending the standard <iq/> stanza, we ensure that any XMPP server is able to receive this message, even though only those using our component know what to do with this message.[4]

```
<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 2: Updated Consistency Requirements stanza

It's worthy to note that the *active* attribute of the *filter-entry* element 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 in the topic *xmpp* and is now interested in the topic *pubsub* (see Listing 2). Since the <time> and <volume> elements are absent from the stanza, the server maintains the previous values (in this case, time=60 and volume=20). Also, applying the same reasoning, the topics *android* and *chess* remain active.

[4]Servers that do not recognize the message respond with an error. To avoid this, the client should use the service discovery extension (XEP-0030) to find out if the server supports our proposed extension, before sending the initial consistency-requirements stanza.

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 ReConMUC, we can define filters based on the presence of a certain XML child element in the message and on the value of that element. Note that there are no limitations to the number of defined filters. Nonetheless, we illustrate the use of such filters with the concept of *active topic*.

In Joom (the brainstorming-oriented client MUC application we developed) 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 at any instant, so their previous active topic is *disactivated*. This is a particular usability requirement of Joom, not a limitation of our extension. This application-level "topic" concept maps well into our protocol-level "filter" - each topic is represented by a filter.

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

Listing 3: Message stanza with the non-standard topic element

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 include a <topic> child element to achieve this association, with the format presented in Listing 3. Note that we also append the topic "xmpp" to the body itself, following an hashtag convention.[5] 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 do not recognize the <topic> element will simply ignore it.

Usage Scenario - messages workflow

The typical message flow starts with a presence stanza sent by Alice to the chat room server (see Figure 1).[6] If Alice is allowed to join this room, she starts receiving all the messages sent to this room by other clients. Next, Alice can send a <consistency-requirements> stanza to specify which messages she requires to receive, based on some filter defined by her MUC client application. Since she is interested in *gossip*, she receives all the *gossip* related messages immediately. Other (unrelated) messages are retained by the server according to the parameters defined in the initial consistency requirements message. Later on, she will end up receiving all those messages (e.g. *movie* messages) with a *delay* indication, so that she can know when they were originally sent.

In Figure 1, we can see in step 2 that Alice is telling the server to filter only those messages whose topic is *gossip*, i.e. she wants to receive all *gossip* messages as soon as possible. In steps 4 and 5 she receives messages that fit within that topic just after they were sent (actually, Alice always receives her own messages immediately, regardless of the consistency requirements). In step 6, Alice receives a message that has been retained at the server because it did not comply with the filter setup on step 2; it is now 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)[23], to communicate the fact that an XML stanza has been delivered with a delay (e.g. 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 MUC room).

IMPLEMENTATION

ReConMUC

As already said, we implemented ReConMUC as a standard XMPP server component, compliant with XEP-0114. This means that it can be easily deployed in all compliant XMPP server implementations, without requiring any change to 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. They can be updated at any time through a <consistency-requirements> message, as described previously in the Adaptable Consistency Model Section.

```
1  function message_received(m):
2    foreach u in users:
3      if m.from == u:
4        send(m,u)      #send imediatly to message origin
5      elsif m.is_reply_to_user(u):
6        send(m.u)      #send imediatly replies to origin
7      else:
8        foreach filter in u.filters:
9          if filter.activates(m):
10           if m.belongs_to_thread():
11             send_all_previous_in_thread(u,m.thread)
12           send(m,u)
13           break
14
15       if m.was_not_sent():
16       retained_msgs[u].append(m)
17
18 function send_all_previous_in_thread(u,thread_id):
19   foreach m in retained_msgs[u]:
20     if m.thread == thread_id:
21       send(m,u)
22       retained_msgs[u].remove(m)
23
24 function round_triggered():
25   foreach u in users:
26     if (now − u.last_sent) > kt:
27       send_all_and_remove_from_retained_msgs(retained_msgs[u])
28     if retained_msgs[i].size > ks:
29       send_all_and_remove_from_retained_msgs(retained_msgs[u])
```

Listing 4: Pseudo-code for processing received messages

The function message_received in Listing 4 represents the pseudo-code for processing messages received at the MUC server. For each message received, the MUC server iterates through all participants in the chat room to decide which ones require immediate message delivery. In particular, in line 4, we see that it instantly sends the message back to the original sender. This is necessary because a sender's MUC client should only show a message after it has been sent and received, to avoid coordination problems, even though it obviously knows its contents.[7] Line 5 checks if the message is a reply to the current user in the iteration. In that case, the server also sends the message immediately because we want the user to see as soon as possible any direct replies to the messages she sent. Next, it applies all the active filters for each user. If the message content matches the filter definition than it will be propagated immediatly (line 12), but first the server will check if this message belongs to some thread,[8] to guarantee that the recipient never sees a reply before the original message (lines 10-11), preserving causal order. Finally, if the message is not 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 4). This function is called by a scheduler and is responsible for checking the time and volume parameters.

[5]Short messages on services such as Twitter or identi.ca may be tagged by including one or more hashtags: words or phrases prefixed with a hash symbol (#).

[6]This is the standard mechanism for joining a chat room, as defined by XEP-045; the client sends a presence stanza to a "virtual" client associated with the chat room.

[7]Coordination problems between members of the chat room may arise if they do not see the messages ordered the same way (e.g. the chat room members need to agree on who was the first person to respond to a given question).

[8]Here, thread means a group of related chat messages, like a reply chain.

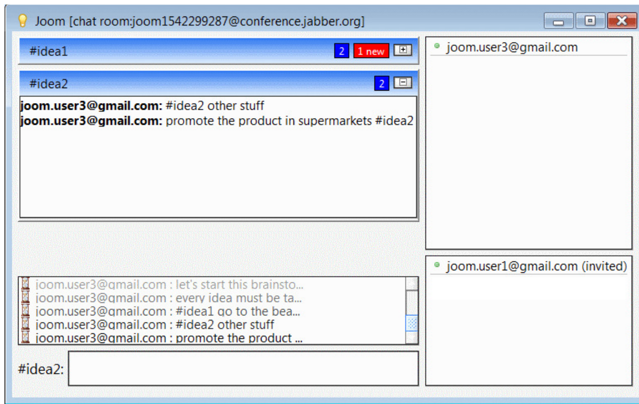

Figure 2: Joom screenshot

Joom

We developed Joom (see Figure 2), a desktop chat application specially suitable for brainstorming sessions with geographically disperse users. With this application, users can not only chat with each other but also create diferent topics and associate chat messages with those topics. Joom differs from other group chat applications by helping users stay focused in only one topic at a time (the active topic) while maintaining awareness of the activity level in other topics (using an unread messages counter and a global timeline). This application uses the XMPP protocol to connect to any compliant chat server (e.g., gtalk server). Joom is able to detect if ReConMUC is installed in the chat server to which it connects. In that case, it starts working in relaxed consistency mode to improve performance and reduce network bandwidth.

EVALUATION

As already said in the Introduction, current MUC protocols waste network bandwidth, thus reducing scalability and performance, by broadcasting all messages as quickly as possible, even when that urgency is not required by the end users. To evaluate ReConMUC scalability and performance, we measured its network, memory and CPU usage and compared it with a non-ReConMUC server in a set of experiments. For that purpose, we used publicly available chat (IRC) logs from very active channels like #ubuntu.[9]

Typically, IRC chat users join and watch multiple rooms at the same time, each one on its own window. We replicate this behavior by setting up only one XMPP MUC room where all messages were associated with a topic (as described in Section) representing the IRC channel which they were coming from. For example, instead of two IRC rooms #ubuntu and #mozilla, we setup only one MUC room where messages from the #ubuntu IRC room were associated with the topic ”ubuntu” and messages from the #mozilla IRC room were associated with the topic ”mozilla”. We only setup one MUC room to increase awareness of every channel's activity.

For this evaluation, we developed an application that parses IRC log files, searching for users, messages and replies. Then, this application 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 the Implementation Section, the replies are immediately sent to the origin.

The experiments consists on creating a chat room on the server, connecting 800 participants and feeding the server with six IRC log files in parallel, during 15 minutes. This simulates 6 simultaneous IRC channels with 800 participants trying to follow the conversations. During these 15 minutes, the server receives aprox. 5400 messages which broadcasts to all participants, resulting in a total of more than 4 million messages.

We setup the experiments to run under different scenarios. In the first scenario ReConMUC is disabled, so every message is broadcast immediately to every participant, as current MUC systems do. In the other scenarios, ReConMUC is enabled taking into consideration that each participant has only one active topic at any instant (analogous to watching just one IRC chat window). Also, in each scenario we change one of the consistency settings, to measure the performance impact. Since the variation of the *time* and *volume* parameters redound in the same effect (as we observed experimentally) we only change the *volume* parameter because it is easier to manipulate, keeping constant the *time* parameter. To better simulate real-life conditions, the *volume* is defined by each participant's thread through a random function around a globally defined *per scenario* parameter. For example, if *vol=30*, each client calculates a random value between 0 and 60 for *vol*, 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. In summary, we evaluate the following scenarios:

- **Without ReConMUC** - We call this the standard MUC scenario given that it is used by current MUC systems that broadcast every message as soon as possible;
- **With ReConMUC (vol=30; no aggregation)** - Non-optimized version of ReConMUC in which retained messages are sent one by one, instead of a single aggregated message;
- **With ReConMUC (vol=10)** - ReConMUC with an average volume of 10 retained messages per user;
- **With ReConMUC (vol=30)** - ReConMUC with an average volume of 30 retained messages per user;
- **With ReConMUC (vol=60)** - ReConMUC with an average volume of 60 retained messages per user.

Note that every participant receives 5400 messages in all scenarios; what differs is not *what* each participant receives but *when* and *how* each participant receives such messages.

[9] Available at http://irclogs.ubuntu.com

Scenario	Avg. Outbound Bandwidth	Rel. Consumed Bandwidth
Without ReConMUC	257.80 Kb/s	100%
ReConMUC (no aggregation) (vol = 30)	296.02 Kb/s	115%
ReConMUC (vol = 10)	210.65 Kb/s	82%
ReConMUC (vol = 30)	148.37 Kb/s	58%
ReConMUC (vol = 60)	145.17 Kb/s	56%

Table 1: Outbound network usage under different scenarios

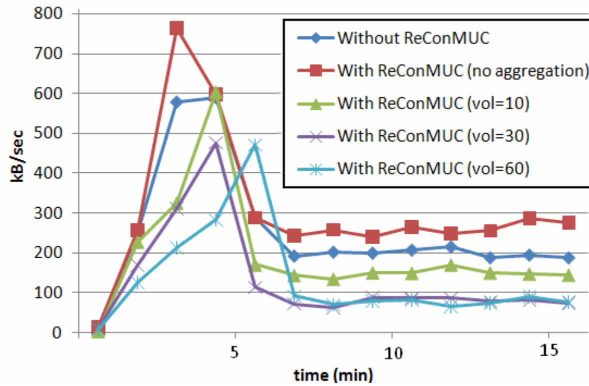

Figure 3: Outbound network usage over time, in kB/sec

The next sections shows how server outbound network consumption, server memory usage and server CPU usage are affected in the five previously mentioned scenarios. We finish with an evaluation of causal delivery propagation time (i.e. how long a direct reply takes); this is an important indicator in MUC applications where related messages propagation should not be affected by other less important messages.

The MUC server is an Intel Core 2 Quad CPU 2.4GHz computer with 8Gb RAM (Linux) and is monitored in three dimensions: outbound network consumption, memory and CPU usage. The MUC clients are deployed in two Intel Core 2 Quad CPU 2.4GHz computers with 8Gb RAM (Linux). These three computers are connected through a LAN.

Outbound Network Consumption

It's important to note that in all experiments we used stream compression (XEP-0138 [23]). As a matter of fact, all the major MUC server and client implementations support this extension using, for example, the ZLIB algorithm to compress all the traffic between the server and clients. Thus, in all experiments for all scenarios, stream compression was turned on.

Table 1 and Figure 3 show the results obtained, while monitoring network usage, for all five scenarios. The scenario where we use ReConMUC without aggregating messages registers the highest bandwidth consumption (even higher than the standard MUC scenario). All other ReConMUC scenarios (that aggregate retained messages at the server) perform better than the standard scenario. The best scenario has the higher volume parameter (vol=60); it decreases bandwidth usage to almost half the bandwidth of the standard scenario.

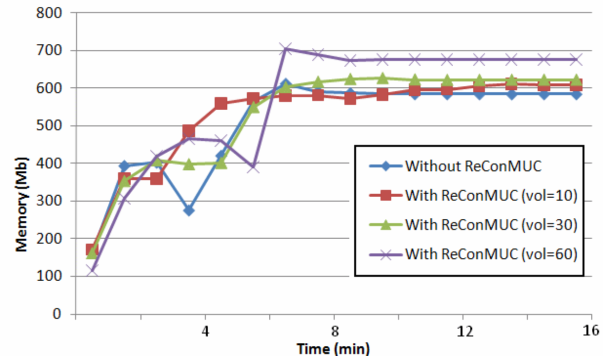

Figure 4: Server memory usage over time

An interesting result is the network usage increase of 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' stamp='2002-09-1 0T23:41:07Z'/>). Since this is a standard XMPP extension, we did not want to modify this element to make it more concise, because that would compromise its portability.

We observed that the ReConMUC server postpones message propagation until a certain condition is met (e.g. the number of retained messages reaches the maximum tolerable volume) and then, it sends all those messages in a burst. However, most XMPP server implementations process such messages as individual packets instead of aggregating them in a larger single packet. By applying the compression algorithm to an aggregated packet instead of its individual parts, the server achieves higher compression rates, since the messages have similar information. As we have observed in the results, this technique effectively drops network usage in 50% approximately (see Table 1). This aggregation is implemented using the *composite* pattern [13], achieving an easier integration with the transport layer of each XMPP server implementation.

As we experimented increasingly relaxed consistency settings (vol = 10, then 30, then 60), we noticed a decrease in network consumption. This is because, as we aggregate more messages, the compression algorithm becomes more effective because of the increased redundancy. Nevertheless, after a certain threshold, the compression efficacy starts to diminish, as we can observe from the very slight improvement when we go from vol = 30 to vol = 60.

This is also easily observed in Figure 3. There is an initial message burst as every participant joins the room: 3 participants per second were joining the room, so it takes aprox. 5 minutes until everyone has joined the room. During this initial phase, most exchanged messages are related to authentication, presence propagation and discussion history (when a participant joins a room, it receives the last 100 messages posted to that room). After that, the consumed outbound bandwidth remains almost constant in all scenarios, being the scenarios with ReConMUC at vol = 30 and vol = 60 the most efficient ones.

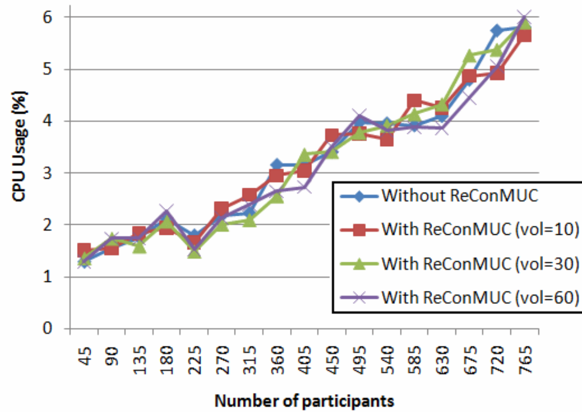

Figure 5: Server CPU usage with increasing number of participants

Server Memory Usage

Typically, relaxed consistency protocols assume a trade-off between two dimensions: network consumption and memory usage. We could reduce network consumption at the expense of server memory used to store retained messages. Unsurprisingly, we observe that the average number of retained messages per user in the server is equal to the corresponding vol parameter for each scenario. For example, if vol is 10, the server retains 10 messages per user; if vol is 30, the server retains 30 messages per user and so on.

In order to understand the real impact of these numbers, we monitored memory usage in the server, obtaining the results shown in Figure 4. Initially, as the participants start joining the chat room and chatting, there is a noticeable increase in memory consumption in all scenarios. After everyone has joined the room, memory usage remains constant, between 600 Mb for the standard "Without ReConMUC" scenario and 700 Mb for the scenario where ReConMUC is used with the highest volume parameter (vol = 60).

Although ReConMUC uses more memory, the increase is negligible in most scenarios. Only when vol = 60, we observe a significant increase, although less than 100 Mb. This low memory consumption is due to a feature of the MUC extension specification that allows anyone who joins a room to receive the previous messages exchanged in that room (i.e. the discussion history). Although this can be turned off, it is usually turned on to reduce the sense of lostness of newcomers to a discussion. Thus, ReConMUC does not store actual retained messages but only pointers to those messages in the discussion history (which exists anyway). That is, even in the most memory demanding scenario, ReConMUC only stores a list of 60 pointers for each user. Also note that the discussion history is usually limited - it only stores the last n messages, being n a parameter that can be configured in most XMPP servers. In our experiments, we setup this value to 100 messages.[10]

[10]ReConMUC does not allow a client to define a vol parameter that is larger than the discussion history limit.

Scenario	Average direct reply propagation time
Without ReConMUC	60 ms
vol = 10	8 ms
vol = 30	9 ms
vol = 60	6 ms

Table 2: Avg. direct reply propagation time

Server CPU Usage

Given that ReConMUC runs inside a loop that iterates through all the MUC participants (see Listing 4), we measured the CPU usage impact of adding participants to the chat room. As we can observe in Figure 5, there is a CPU usage increase, as the number of participants grows, which is similar in all scenarios. Thus, ReConMUC does not significantly impact CPU usage. This is because ReConMUC algorithm is very simple, when compared to all the necessary operations in a standard MUC (e.g., guaranteeing authentication and authorization of the participants).

Causal Delivery Propagation Time

Although network, memory and CPU consumption are important indicators of ReConMUC performance, it is also important to evaluate how causally related messages are propagated. In particular, we measured the average propagation time of a direct reply message, until it reaches its recipient, in the same five scenarios used in the other experiments.

In order to measure the propagation time of a direct reply, the sender attaches a timestamp to the message.[11] When the recipient receives the message, it compares the timestamp of the message with the current time. Every participant keeps track of these propagation times and, in the end, an average of these values is calculated.

The results are presented in Table 2 and show that with ReConMUC, direct replies take approximately 10x less to reach the recipient (w.r.t. the standard scenario). This is due to the fact that its special meaning is considered, i.e. the direct reply is not processed as an ordinary message. This is an important observation: with ReConMUC we deliver important messages, such as direct replies, as soon as possible, even if this means that remaining messages may take longer propagation times. We believe this is more aligned with users expectations, when using real-time communication tools.

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 do not require immediate attention of their recipients. This raises serious scalability problems to such systems.

[11]We modified the MUC client application to attach the timestamp in this particular test, since the normal behavior is not sending any timestamp.

In this paper, we propose an adaptable consistency model for the propagation of MUC messages based on three dimensions: time, volume and content filters. These parameters are configured for each client, based on their current context, device capabilities, network availability, etc. ReConMUC implements this model on top of XMPP, an open and widely used messaging protocol; therefore, our model can 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 and CPU consumption, thus improving the scalability of the system. As future work, we plan to investigate other consistency dimensions such as the social network of the participants (e.g. relax consistency as we increase the social distance between the sender and the receiver).

REFERENCES

1. Pubsubhubbub - http://code.google.com/p/pubsubhubbub/, 2009.
2. G. Bafoutsou. Review and functional classification of collaborative systems. *International Journal of Information Management*, 22(4):281–305, August 2002.
3. K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *ACM SIGOPS Operating Systems Review*, 21(5):138, 1987.
4. E. Bjerrum and S. Bø dker. Learning and living in the 'new office'. In *Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work*, pages 199–218, Helsinki, Finland, 2003.
5. L. Cabrera, M. Jones, and M. Theimer. Herald: achieving a global event notification service. *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 87–92, 2001.
6. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.
7. A. Carzaniga, D. Rosenblum, and A. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*, pages 219–227, 2000.
8. P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. *In Proc ofCSCW'92, Toronto, ACM Press*, pp:107–114, 1992.
9. J. Dyck. A Survey of Application-Layer Networking Techniques for Real-time Distributed Groupware, 2006.
10. J. Dyck, C. Gutwin, T. Graham, and D. Pinelle. Beyond the LAN: Techniques from network games for improving groupware performance. In *Proceedings of the 2007 international ACM conference on Supporting group work*, pages 291–300. ACM, 2007.
11. P. Eugster, P. Felber, R. Guerraoui, and AM. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.
12. G. Fitzpatrick, S. Parsowith, B. Segall, and S. Kaplan. Tickertape: awareness in a single line. *Conference on Human Factors in Computing Systems*, (April):281–282, 1998.
13. E. Gamma, R. Helm, R. Jonhson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Co, 1995.
14. C. Gutwin. The effects of network delays on group work in real-time groupware. In *Proceedings of the seventh conference on European Conference on Computer Supported Cooperative Work*, page 318. Kluwer Academic Publishers, 2001.
15. C. Gutwin, K. Schneider, D. Paquette, and R. Penner. Supporting Group Awareness in Distributed Software Development. In *Engineering Human Computer Interaction and Interactive Systems*, pages 383–397. Springer Berlin / Heidelberg, 2005.
16. R. W. Hall, A. Mathur, F. Jahanian, A. Prakash, C. Rasmussen, and A. Arbor. Corona : A Communication Service for Scalable, Reliable Group Collaboration Systems. In *Proc. Conf. on Computer-Supported Collaborative Work (CSCW)*, pages 140–149, 1996.
17. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
18. A. Mathur, R. Hall, F. Jahanian, A. Prakash, and C. Rasmussen. The Publish / Subscribe Paradigm for Scalable Group Collaboration Systems. *Ann Arbor*, 1001(313):48109, 1995.
19. R. Messeguer, S. F. Ochoa, J. A. Pino, E. Medina, and L. Navarro. Building Real-World Ad-Hoc Networks to Support Mobile Collaborative Applications : Lessons Learned. *Groupware: Design, Implementation, and Use*, 578:1–16, 2009.
20. J. Olson and S. Teasley. Groupware in the wild: Lessons learned from a year of virtual collocation. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 419–427, 1996.
21. B. Plale and Y. Liu. Survey of Publish Subscribe Event Systems. *Technical Report TR574, Indiana University*, 2003.
22. P. Saint-Andre. RFC 3920: Extensible Messaging and Presence Protocol (XMPP): Core, 2004.
23. P. Saint-Andre. XEP - XMPP Extension Proposal, 2005.
24. P. Saint-andre and R. Meijer. Streaming XML with Jabber/XMPP. *IEEE Internet Computing*, 9(5):82–89, September 2005.
25. Y. Saito and M. Shapiro. Replication: Optimistic approaches. *Hewlett-Packard Labs Technical Report HPL-2002*, 2002.
26. S. Sawyer and P. J. Guinan. Software development: processes and performance. *IBM Systems Journal*, 37(4), 1998.
27. B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with elvin4. In *Proc. AUUG'00*, volume 61, 2000.
28. J. Smed, T. Kaukoranta, and H. Hakonen. Aspects of networking in multiplayer computer games. *The Electronic Library*, 20(2):87–97, 2002.
29. M. Smith, J. J. Cadiz, and B. Burkhalter. Conversation trees and threaded chats. *Proceedings of the 2000 ACM conference on Computer supported cooperative work - CSCW '00*, pages 97–105, 2000.
30. A. Tanenbaum. *How USENET is implemented*, pages 675–677. ISBN, Prentice-Hall, 1996.
31. Walker. Instant Messaging Is Growing Up, Going to Work, The Washington Post, 2004.
32. W. Wu, G. Fox, A. Uyar, and H. Altay. Design and Implementation of a collaboration Web-services system. *Neural, Parallel & Scientific Computations*, 12(3):391–406, 2004.
33. H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. *Proceedings of the 4rd Symposium on Operating Systems Design and Implementation*, 2000.