# An Analysis of Instant Messaging and E-mail Access Protocol Behavior in Wireless Environment

**IIP Mixture Project**

Simone Leggio

Tuomas Kulve  Oriana Riva  Jarno Saarto

Markku Kojo

March 26, 2004

University of Helsinki - Department of Computer Science

# TABLE OF CONTENTS

# LIST OF FIGURES



# LIST OF TABLES

# 1 Introduction

This document discusses the behaviour of Instant Messaging and E-mail Access protocols in a wireless environment. The document is divided into three parts: the first part presents a description of some of the most common Instant Messaging and Presence (IMP) systems and protocols and two mail access protocols, followed by an analytical study on the message exchange of the protocols. In the second part we chose Jabber IMP system [jabber] for more detailed study and performed a set of experiments for testing the behaviour of Jabber IMP system in presence of a wireless link in the client-server path. The results of the experiments are reported. Finally, in Part III, based on the results obtained from the experiments, we outline some general guidelines for implementing IMP servers in order to allow them to behave efficiently in wireless environments.

# PART I: BACKGROUND AND PROTOCOL ANALYSIS

# 2 Instant Messaging

Instant Messaging applications are tools that allow users to exchange messages with remote users, similarly to what provided by e-mail services, but in an instantaneous fashion. The number of users of instant messaging software is quickly growing as it not only helps to communicate better, but also provides a cheapest way to communicate on long distance using free voice/video conferencing or to share files and directories. In this section, we introduce and compare the protocols of some of the most popular leading brands among instant messaging services. We provide for each of them a description of the main features and protocol operation and conclude with a comparison of the main advantages and disadvantages related to each of them.

The main issue in instant messaging (IM) is that it does not exist a precise standardization of the architecture and protocols, as they are most often proprietary and the owners do not generally allow exchanging instant messages with users of rival IM applications, nor they disclose contents of the source code. All the information and software based on such systems, unless provided by the owner, is due to reverse engineering work. With the main purpose of defining a standard protocol for IM, the

IETF working group IMPP [impp] is working to develop an architecture for instant messaging and presence service in order to provide a specification on how authentication, message integrity, encryption and access control must be integrated.

The most popular IM protocols are the following:

- AOL (Oscar) [Oscar]
- MSN Messenger [Min03]
- ICQ [Isak01]
- Yahoo [yahoo]
- Jabber [jabber]
- IRC [RFC1459]

Considering Figure 1 based on [Oett01] report we notice that the top three brands of instant-messaging applications are AOL, MSN, Yahoo and ICQ. We focus our study also on two other less widespread protocol: Jabber and IRC. They are particular interesting in our analysis since they are the only two open source protocols.

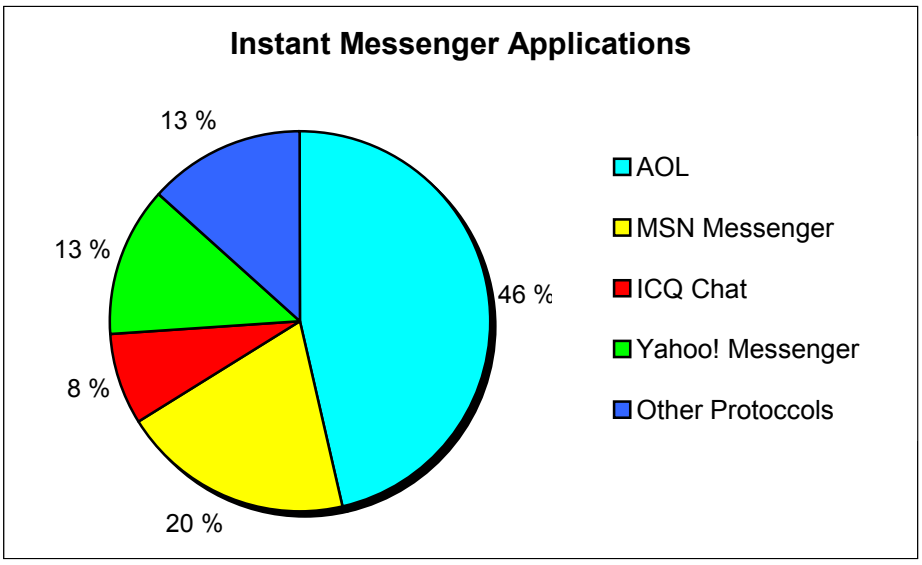



**Figure 1**. **Unique Users of Instant Messenger Applications September 2001 at Home in the U.S.**

The document also provides an introduction to the IMAP protocol [imap], used for the access to electronic mailboxes following the online paradigm. The protocol is an enhancement to the widely used POP protocol [pop], supporting also the online and disconnected mode, while POP is meant for an offline access (supported by IMAP as well). With “online paradigm” we mean that user can access

and manage their mailboxes from remote, even though they were acting locally, as opposed to offline access mode, which implies that all the managing operations are performed locally.

# 3 ICQ

## 3.1 Overview

ICQ [Isak01] stands for "I Seek You". It was originally developed by Mirabilis. Mirabilis was founded in July 1996 by four young Israeli computer users. Yair Goldfinger, Sefi Vigiser, Arik Vardi, and Amnon Amir created the company in order to introduce a new communication tool for the Internet. At that time, Internet provided a connection for each people, but the interconnections among users were still missing. As a result, they developed the technology to allow people connecting to the Internet to find and locate each other more easily and to communicate. In November 1996, the first version of ICQ was introduced to the Internet. The ICQ protocol was bought by AOL in 1998. Although the specific details of ICQ protocols are not made public by Mirabilis, there have been several groups that have attempted to reverse-engineer it. Most of them are cooperating now, sharing the information they have obtained about it. However ICQ99 was the last official ICQ client. With ICQ2000, the official client ICQ client uses a modified version of AOL/Oscar described later.

ICQ is a revolutionary, user-friendly Internet tool that informs the user who is on-line at any time and enables him to contact them at will. ICQ alerts him in real time when the other users log on. The need to conduct a directory search each time one user wants to communicate with a specific person is eliminated. ICQ is one of the most popular *peer to peer instant messaging system* on the Internet, boasting over one million registered users. It was also the system that sparked the instant messaging boom, seen on the Internet in recent years. The most popular features of ICQ are the possibility of sending instant messages, SMS and e-mail, launching net meeting with multiple users for video and telephony chat, sending file of any type and size and even sharing them, chatting in real time with one or more friends, sending on line voice message and supporting 13 versions of languages. The popularity (and relatively old age) of the ICQ system has lead to several benefits and disadvantages we consider later [Sin03].

## 3.2 Protocol Operation

According to the ICQ protocol specification document [Isak01], ICQ operates in a server-based, peer to peer fashion. A conceptual view of the ICQ protocol architecture is given in Figure 2.

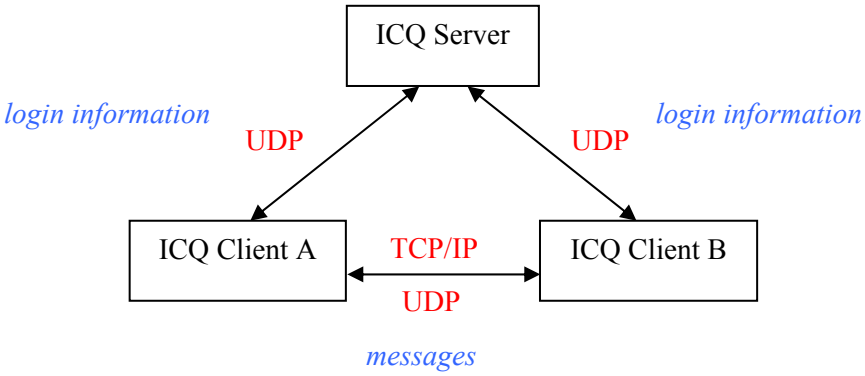



**Figure 2. ICQ Protocol Operation**

There two main type of communication: between Client-Server and between Client-Client [Sin03]. ICQ is based on both UDP and TCP. Later we also provide two examples of normal and abnormal operation [ICQ01].

### 3.2.1 Client to Server

The Client communicates with the ICQ Server using UDP connections. Each UDP packet sent from the client to the server is encrypted before being sent, unlike the packets sent from the server to the client. The most important fields in the ICQ Packet Header are the SEQ_NUM1, SEQ_NUM2, the COMMAND and the SESSION_ID. The SEQ_NUM1(2) is set to a random number in the login packet and increased by one all (most of) the times a packet is sent. The COMMAND field is used to specify the type of the message sent such as login message (CMD_LOGIN), acknowledgment reception (CMD_ACK), etc. The SESSION_ID is a random number, chosen when the login packet is sent and kept until the user logs out. It allows the server to identify the packets sent by a specific client and ignore packets not coming from registered users. At the same time in each answer sent back from the server the same number is included in order to make sure the client that the packet received is not a "spoofed" packet. It has just to compare the number in the received packet to the one sent. Every time the client sends a packet to the server it must receive an acknowledgment (SRV_ACK) from the server

side, otherwise it will retransmit the segment. At the same time the server expects acknowledgments from the client side, with the exception of the SRV_ACK packet.

### 3.2.2 Client to Client

The communication is TCP-based. At the beginning the clients start to communicate using the UDP protocol to send the IP, LAN IP and each of the client User's ID (ICQ number). After receiving this packet, the communication is TCP-based. The size of the TCP packet is usually sent first than the packet itself. The TCP is "sizeless" and if the receiver knows before receiving the packet the actual size of it he can check if it is correct or not. After the connection establishment the messages start to be sent and the packets must be acknowledged from the receiver side.

### 3.2.3 Normal Operation

When a user wants to build a conversation with someone must log into the ICQ server. In such a way his status is set to be online. After that the server sends a message containing the contact information (IP address, port number, etc) of the new logged-on user to all the other users already connected or the users that wish to know the online status of the newly logged-on user. The user then sends a copy of his contact list to the ICQ server that responds with the contact details, such as IP address and port number of anyone in the list that is also online. If one user wishes to send a message to another user the relative port is opened and the message streamed down the port as it is typed. The messages are sent directly from Client to Client.

### 3.2.4 Abnormal Operation

It is possible to send messages through the server in cases direct TCP connections are not possible. In such cases the communication is streamed down a port that is always kept open between the client and server and it is usually utilized for passing system messages. The ICQ server then forwards the message to the intended recipient via a similar UDP connection between that user and itself.

## 3.3 Discussion

### 3.3.1 Advantages of ICQ

- When a user send a message to offline user the ICQ server stores that message and sends it to the user as soon as he comes online. This is advantageous for systems subject to experience network or software failure on the listening side. However if a large volume of messages is sent while the receiver is not online, when he comes online again he may be overloaded.
- ICQ provides great functionalities such as the capability to share directories on users' machines, which is similar to file sharing software, Napster or Kazaa. A user can download a file freely or server as a server station allowing others to download from it. Anyway this could cause security and privacy bandwidth issues as harmful material could be passed around without any control relatively comprehensive directory services for users to find other ICQ users to add to their contact lists. We could use this same directory service to allow subscribers to find devices they wish to receive messages about.

### 3.3.2 Disadvantages of ICQ

- ICQ is not an open-source protocol and this means that all the information we have comes from reverse engineering work.
- The main issue in ICQ protocol implementation regards the security aspect. The type of ICQ communication between server and client or client and client is the main cause of these problems. When an ICQ user sends a message to another user, it will include the IP source and destination addresses. Moreover the messages are not encrypted and information such as the user ID and his IP address are not protected. This may allow someone to intercept the packet and successfully decrypt it and modify the messages exchanged or even to use ID of other users improperly.
- ICQ is based on synchronous communication hence compared to other asynchronous protocol, such as MSN ICQ might be slower. Indeed for every ICQ message sent a reply has to be received to know if the message has been accepted.
- Many operations on client-side.

# 4 AOL/Oscar

## 4.1 Overview

Oscar [Oscar] (Open System for Communication in Realtime) is the official IM protocol developed by AOL. Oscar is a closed protocol and all the knowledge available about it comes from reverse engineering. It is TCP-based and binary. Since Oscar is not an open protocol, non-proprietary clients do not support a lot of the features it offers, while it is easier to provide all the functionalities included in TOC [Proto].

## 4.2 Protocol Operation

The architecture of the Oscar protocol is illustrated in Figure 3. The server is distributed and the two main components are the *Authorizer*, which validates username and password and the *BOS* (Basic Oscar Service). Before connections are made to any of the BOS or special-purpose servers it is necessary to be authorized by the *Authorization Server* (login.oscar.aol.com). It will reply sending a cookie that automatically authorizes the user to connect to any of the BOS or special-purpose (for example Advertisement, Chat, etc) servers.

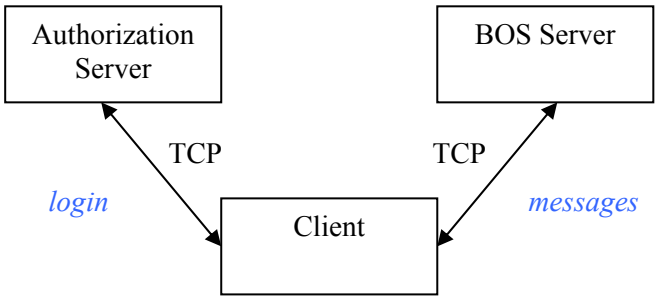



**Figure 3. The Oscar protocol structure**

The normal steps taken to create an average AIM session are as follows [Oscar]:

1. Connect to Authorizer and retrieve Cookie.
2. Connect to the Authorizer-recommended BOS server and initiate BOS service

3. (Optional) Connect to Advertisements server and retrieve first block of ads (repeat at regular interval)
4. (Optional) Connect to any other non-BOS services that may be available (AFAIK, none at this point)

The last three steps can actually follow any order, but authorization must always be the first.

*Connect to Authorizer and retrieve Cookie*

OSCAR has a sense of the "single-login" concept. In the first step the client logins and gets a "cookie" that automatically authorizes him to utilize any of the OSCAR-associated services, just by sending them his cookie.
First of all he has to connect to the Authorizer. It currently resides at the DNS address login.oscar.aol.com. The server also sends for each new connection a *Connection Acknowledge* message. After the connection, the client must send the *Authorization Request.* The response to this, the *Authorization Response* contains the cookie to be used for the BOS and other connections. But, if the Authorization Request fails, the client will receive one of the several *Authorization Errors.* After receiving the cookie, it is safe to disconnect from the Authorizer.

*Connect to the Authorizer-recommended BOS server and initiate BOS service*

The second step is usually to connect to and initiate service with the BOS. The address of the BOS is contained in the *Authorization Response.* First of all the client sends the *BOS-Signon* command to the server, but it may be better to wait to send this message until the *Connection Acknowledge* command is received from the server immediately after the connection opens, although this is optional and can be processed later.
A typical sequence of messages exchanged is:

1. Server sends *Connection Acknowledge*
2. Client sends *BOS Sign On* command.
3. Server sends *BOS Host-Ready*. Sent by the server to notify the client that it is ready to begin service
4. Client sends *Rate Information Request*. The client sends it in order to know how fast it can send SNACs. If this rate is disobeyed, it will be (at worst) disconnected
5. Server sends *Rate Information Response*
6. Client sends *Rate Information Acknowledge*.

7. Client requests several information: Request User Information, Request New Service, …
8. Server sends all the information requested
9. Client sends up buddy list using the *Add Buddy to Buddy List* command (It adds a number of buddies to his buddy list, causing AIM to send us on/off events for the given users. One of the main AIM features is this function called the "Buddy List". It is analogous to the "Contacts" in ICQ terminology. Basically, at login, you send a list of screen names to the message server. These names get watched for login/logoff events, and you will get notified when these things happen. )
10. Client sends up user's profile using the *Set User Information* command.
11. Client sends the *Set Initial ICBM Parameter* command.
12. Client sends the *Client Ready* command (Notifies the server that he is on-line and ready to receive messages)

*Logout*

This is a very simple operation. The easiest and abrupt way to do it is just closing the connection to the main message server. Sometimes, though, the AIM client sends a small command to the server before it closes, but expects no response.

# 5 MSN Messenger

## 5.1 Overview

MSN [Min03] Messenger is a popular instant messaging client included in MS Windows operating systems. It is currently maybe the fastest growing messenger: it had 9.6 million users in 2000, and 18.5 million users in 2001. The MSN Messenger includes support for the following features:

- **Send IM, SMS and email:** MSN supports conversations with 1-14 users concurrently. It implements an automatic typing indicator for alerting users whenever one is typing a response.
- **Support for a number of languages**
- **Voice and Video -conferencing**
- **Send and receive file**

- **Chat with people in MSN Member Directory**
- **Invite friends to play a game**
- **Block instant messages from selected or unknown people**
- **Remote assistance**
- **Work with a user with same program or whiteboard**
- **Microsoft.Net alert**

MSN Messenger client has been implemented also for Linux OS with limited features. All information between client and server are sent unencrypted. This includes passwords and email addresses sent when identifying a client.

## 5.2 Protocol Operation

MSN Messenger uses TCP protocol for all communication between hosts. For normal messaging all traffic is between client and server. Only for special features (for example file transfer) client to client traffic takes place. This has a number of benefits including security reasons such as attacks by malicious users and easy firewall configuration, because only outgoing traffic from specified ports have to be accepted (as well as established traffic). A client can be connected to multiple servers concurrently. In this kind of architecture the server's could easily become congested. The architecture, however, supports arbitrary number of servers each of them able to be replaced at any time. The MSN protocol architecture is depicted in Figure 4.

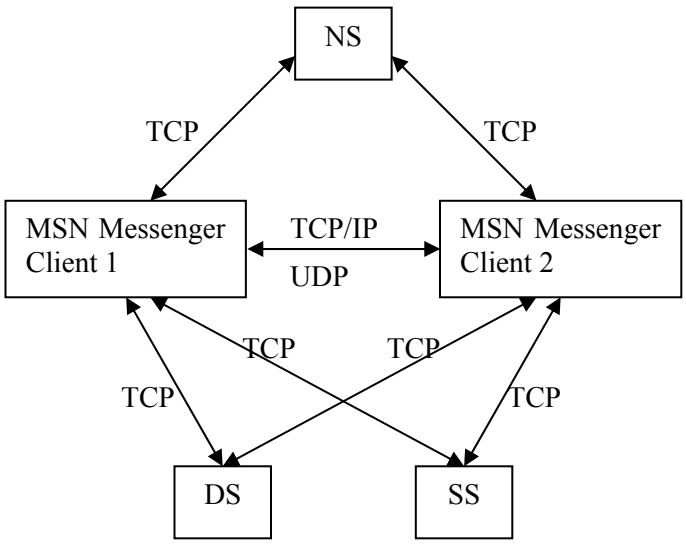



**Figure 4. Architecture of MSN protocol [Sin03]**

A client connects first to a **Dispatch Server** (DS). This server has knowledge about available **Notification Servers** (NS), and refers the client to one. For most of the time a client is connected to the NS and all notification messages are transmitted between client and NS. The client for example informs the NS when there is a change in the client's state and invitation requests.

When a client wishes to contact another client, it sends a message to its NS. The NS refers the client with a **Switchboard Server** (SS), which is server dedicated to lightweight communication sessions. Once the client has connected to the SS, the destination client's NS asks the destination client to connect to the same SS. Messages from a client to another are then passed through the SS.

Messages between a client and server are totally asynchronous, which makes the clients able to start writing new messages instantly after a previous message has been typed. Messages are identified with a `Transaction Identifier`, which is a 32 bit unsigned integer. When a client places a request, it includes a new transaction identifier. The server includes the identifier in its response when the transaction eventually completes. The messages include a mime header, which defines the type of the content. Message types are identified with three letters. Examples of requests as well as more detailed information of the MSN protocol are available at [Min03].

## 5.3 Discussion

The MSN messenger is a popular instant messaging system, which has many functionalities. It is based on an asynchronous protocol, which ensures that new messages can be sent without waiting response of previous messages.
Many of the features are optional, and thus the messenger can be configured not to send for example typing information or on-line indications.

Because all connections are from a client to a server, the messenger can be used in most systems and firewall configurations. The centralized server architecture also simplifies group messaging, but may cause problems if the servers become crowded. The messenger architecture is, however, scalable and any of the servers can be replicated arbitrary number of times. Because the MSN Messenger is part of MS Windows, the number of registered users is likely to be increased.

The protocol is, however, closed and much of the documentation available regarding the protocol functionality is gathered by reverse engineering. Although there are clients also for other operating

systems besides Windows, they are often incomplete and do not include all features of MSN Messenger. If Microsoft changes something in the protocol, problems will arise.

# 6 Yahoo

Yahoo! Messenger rates right up there with ICQ and AOL Instant Messenger. Features include instant messaging, voice chat, file transfer, and conferencing capabilities, as well as news, weather, stock, and sports reports [yahoo]. Yahoo! Messenger offers excellent integration with the Web, and specifically with the My Yahoo! Web site. It can also be a part of a home page, telling when the author of the page is online and allowing the visitors to start a conversation with the author. Extensive configuration options are available for customizing the program. Yahoo! Messenger also has a strong and robust search feature for finding other users on the system as well as information from the WWW. The protocol is closed and all unofficial documentation is based on reverse engineering.

We just give a short overview of the protocol based on [yahoop]. The Yahoo protocol consists of the following steps:

1. Contact theYahoo! Server.
2. Send Login
3. Receive the strings to encode the login and password. The server also sends a *Session ID* that has to be passed for every communication with the server.
4. Send hence worth MD5 encoded strings viz login and password along with the session ID.
5. If accepted the server sends buddy list and details of various groups of friends etc. It also sends a couple of cookies.
6. Then the server sends the list of ONLINE buddies along with their status messages. At this moment the client becomes "Available" and ready to chat. The message sent must include session ID, length of body, receiver, sender, message and termination.

As regards the chat functionality, there are several rooms available on yahoo chat server each with a max capacity of 40-50 and classified according to some category. The server stores the number of rooms in each category and assigns them an ID that is used to join a room.

# 7 IRC

## 7.1 Overview

The IRC (Internet Relay Chat) protocol was specified in 1993 by Oikarinen and the protocol is specified nowadays in five RFCs [RFC1459, RFC2812, RFC2812, RFC2812, RFC2813]. The IRC protocol is based on ASCII strings with terminating character over TCP connections.

*The Internet Relay* Chat (IRC) protocol has been designed over a number of years for use with text based conferencing [RFC1459]. IRC itself is a teleconferencing system, which (through the use of the client-server model) is well-suited to running on many machines in a distributed fashion. A typical setup involves a single process (the server) forming a central point for clients (or other servers) to connect to, performing the required message delivery/multiplexing and other functions. The IRC protocol is probably the oldest ”chatting” protocol still used. In total there were 534 separate IRC networks from which 449 were active at the time of writing. The active networks had 1.3M users at that time [STAT].

## 7.2 Protocol Operation

A layout of a small IRC network is shown in Figure 5. Using the IRC protocol clients always talks to other clients through a server using single TCP connection. The message can go through multiple servers before it reaches the other client.

With the IRC protocol a client can talk to one other client using messages designated to that specific client. When talking to multiple users a *channe*l is used. A client can talk to everyone on a channel by sending a message designated to the channel. All IRC protocol messages are in ASCII format. An IRC client connects to a server specified by the user. First client sends **NICK** message followed by **USER** message to the server. After these the server sends a welcome message back to the client.

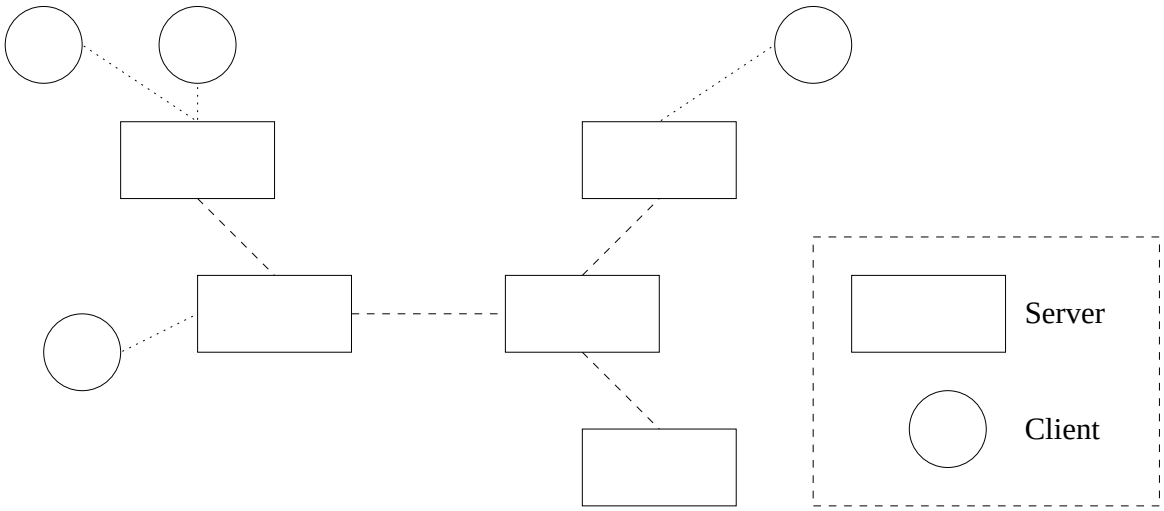



**Figure 5. Example of IRC network**

Every server knows every client in the network and when a client wants to talk to another client, all it needs to know is the *nickname* of the other client and the server delivers the possible message through other servers to the destination client. These nick names are unique in the whole network. Often conversations are discussed in a channel. A channel is created when a client *joins* a channel that does not yet exist. There can be large number of clients talking in a channel and there everyone sees messages sent by other clients. Some clients can be *channel operators* and they have functions to keep the order in the channel. These functions include methods like removing a client from a channel or preventing one to send messages to the channel.

Talk messages sent and received are **PRIVMSG** messages. Here are three basic examples, two private messages and one public message to a channel.

”Wiz” receives a message from ”Angel”:
**:Angel!wings@irc.org PRIVMSG Wiz :Are you receiving this message ?**
A client sends a message to ”Angel”
**PRIVMSG Angel :yes I'm receiving it !**
A client sends a message to channel called ”#test”
**PRIVMSG #test :Anyone here?**

A maximum length of an IRC protocol message is 512 bytes and therefore one IRC message can be usually transmitted in one TCP segment (assuming 576 MTU).

## 7.3 Discussion

The main problem of the IRC protocol concerns only servers; they all have to know every client and every server in the network all the time. A client situation is much simpler. A client needs only to know one server to connect and the nicknames of the clients it wants to talk with or the name of channel to join.

The communication between a client and a server is simple and overhead caused by the protocol is low, although a binary format would take even less space than the ASCII format. In a test where both the server and the client were run in a single Linux computer, a connection establishment to a server took roughly 15 times the RTT because every message was sent in its own TCP segment and ACKed before next message was sent. This might not be the case with a server running on separate host. Opening the connection included NICK, USER and the server welcome message as well as some checks of the client identity.

# 8 SILC

## 8.1 Overview

Secure Internet Live Conferencing (SILC) is a IRC-like messaging protocol [SILC1]. SILC was designed by Pekka Riikonen in year 2000 and it is now specified by IETF in four internet drafts [SILC1, SILC2, SILC3, SILC4].

SILC is IRC-like to a user because it has very similar commands and the only current client implementation uses the same interface as one popular IRC-client (irssi). SILC combines features from both protocol styles, IRC and instant messaging, and can be implemented as either IRC-like system or IM-like system. Some of the more advanced and secure features of the protocol are new to all conferencing protocols. SILC also supports multimedia messages and can also be implemented as a video and audio conferencing system.

## 8.2 Protocol Operation

Messages between SILC network components are binary messages sent over TCP connections. The SILC network has three different kinds of components: clients, servers and routes. An example network is shown in Figure 6. A client connects to a server. A server knows every client and every channel that was created by it, but it does not know the global state. Every server connects to a router and all routers in a SILC network is connected to some another router in such a way that they form a circle. Also every router and every server has a backup connection that is taken in use when the primary target stops responding. Every router has to know the global state and therefore the SILC is designed so that there is lesser amount of routers in SILC network compared to servers and the routers can be more powerful computers.

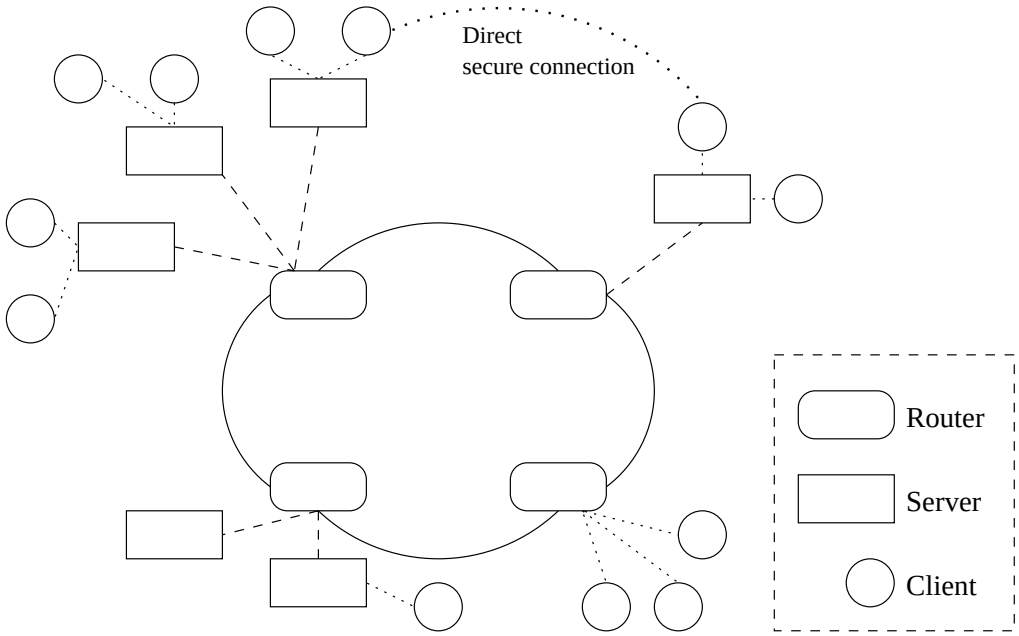



**Figure 6. Example of SILC Network.**

The SILC protocol is designed from the beginning to be secure and this complicates the protocol significantly. Every client-server, server-router and router-router connection is encrypted with their own session keys. The session keys are agreed in the beginning of the connection using strong and secure key exchange protocol [SILC3]. Normally servers and routers decrypts the messages sent by a SILC client, but a SILC client can also choose not to trust the servers and generate session keys with another SILC client outside the SILC network. Also channels can be secured with a password so that SILC servers cannot decrypt the messages sent to the channel.

## 8.3 Discussion

SILC looks very similar to IRC from the user point of view. The protocol itself is much more complicated because of the secure messaging. The protocol uses binary messages that can be also compressed, so in this sense it is suitable for low bandwidth connections. On the other hand there are more messages sent between the network components because of the security issues. These additional security related packets are mostly sent when opening a new connection or joining a channel. When sending a chat-message from a client to another, only one binary coded message is sent. CPU power can be an issue with mobile devices as the encryption and decryption of messages requires much computing power.

# 9 Jabber

## 9.1 Overview

Jabber [] is the most widespread open source platform, using an XML encoded protocol, especially tailored to provide instant messaging (IM) and presence services over the Internet; however, Jabber is not designed just for this purpose, but several are the applications that may benefit and use the Jabber protocol suite. The protocol is totally free from legacy rights; both on the server and on the client side, which means that anyone can design its own Jabber client and even that any organization can freely implement an internal jabber server. Many are the advantages that come out from this approach:

- The fact that the protocol is open lead to a better understanding of it, as everyone can learn from the work previously done and make available its code to other developers for the same purposes.
- XML allows easy extensibility to the main features of the protocol. The Jabber Software Foundation accounts for the common extensions.
- Decentralized approach. Since any organization can have its Jabber server, the resulting architecture is more scalable as lighter load is posed on the single servers, compared to a centralized approach.

Although the protocol itself does not provide means to achieve interoperability with other IM protocols, this is possible by means of server-side gateways, which take care of the communication

between users in the Jabber space and users in the space of other (possibly) proprietary protocols. Jabber-related activities are ongoing in the Extensible Messaging and Presence Protocol (XMPP) IETF working group [XMPP]; the working group has been chartered to discuss extensions to the XMPP protocol, which is the core of the Jabber platform, especially to be compliant to the requirements posed by [RFC2779], RFC from the IETF Instant Messaging and Presence Protocol (IMPP) [impp] working group, chartered to define a standard protocol for providing instant messaging and presence services. A good document describing Jabber main features is [SA01].

## 9.2 Protocol Operation

Jabber is essentially a client-server architecture, where users register to a server to have access to the Jabber system, and use that server as intermediary when exchanging messages with other users, also if they are registered with another server. User names in Jabber are in e-mail like format, such as userA@serverA.dom.

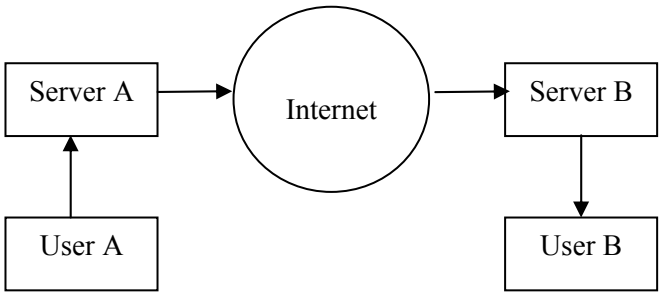



**Figure 7. Simplified Jabber network architecture**

Figure 7 above shows the route followed by a message sent by user A to user B, when both users are registered to different servers. The Server A can contact the Server B and retrieve its address because server names are domain resolvable, e-mail like. DNS can be used for the purpose.
The Jabber protocol basically foresees the following kinds of messages:

- <message/> used to actually carry the messages exchanged.
- <presence/> used for the service of presence
- <iq/> (info/query) used for messages of other kind, such as for authorization purposes.

The exchange of message in the Jabber protocol is stream-based, and a Jabber session is indeed identified by two XML streams flowing in the two client-server directions. The protocol runs over

TCP, using the well known, at protocol level, port 5222. A typical message exchange in a Jabber session, could be the following:

1. A Jabber client sends a message to a Jabber server to open a connection and initiate an XML stream.
2. The server sends its reply and opens an XML stream for the server to client direction.

The connection established is persistent and there is no need for clients to poll the server, as it happens in e-mail; as soon as a message destined to a user is received by the correspondent server, it is immediately delivered, according to the philosophy of instant messaging. After the sockets have been opened, as described in the steps (1) and (2) above, the client must authenticate within a certain time with the server, to avoid a closing of the connection from the server side. Jabber allows many authentication ways, from the simple plaintext password to encrypted passwords and so on. In the example, reported with the details about the exchanged messages in [SA01] the client requests to the server a list of the authentication methods supported, the server replies and the client can thus choose the preferred one. It is not specified in [SA01] whether the client can store somehow the list of the server supported authentication methods to avoid this preliminary exchange of messages; however, [SA01] describes the protocol, and whether the client can provide such a feature is an implementation dependent issue.

3. The client sends an <iq> type message requesting the list of authentication methods supported by the server.
4. The server replies with another <iq> message providing the list.
5. The client provides its authentication information using a format supported by the server using again an <iq> message.
6. The server acknowledges the transaction with an <iq> message.
7. The normal message flow can begin and all the features of Jabber can be used, by using the three above-mentioned messages.
8. The Jabber client closes the XML streams sending a proper message to the server, which in turn will close the TCP connection.

The number of messages exchanged to open a connection is thus minimal.

## 9.3 Discussion

In this section the Jabber system for instant messaging and presence has been briefly introduced. The most relevant feature of this system is that is totally open source, both in the client and in the server side, and it has grown to be the most widespread open source instant messaging and presence platform used. The fact that also server side code is free source, allows organizations to build their own Jabber server implementation, which has the advantage to add scalability to the system, as there is not only a centralized server to manage all the operations, but the burden can be scattered throughout multiple servers.

Moreover, organization can decide to implement only an internal service of instant messaging, isolating the Jabber server from the Internet, which is highly desirable in terms of security and not possible using proprietary instant messaging services, like ICQ or MSN, as the server is located outside the local network. The Jabber community has oriented the efforts in developing simple clients, where all the low-level functionalities are handled by library functions, while the major effort for implementers is in designing user interfaces. More complicated is the implementation for Jabber servers, as they must handle various functions, like authentication, presence, offline message storing and so on; the basic idea is to build servers in a modular way, having a core which handles the basic functions and wrap around advanced features, such as translation into other protocols.

Another strength of the protocol is its flexibility, thanks to the possibility to use XML namespaces feature to allow extensions to the base protocol; the XML format is in a human readable format, and this helps in the implementation. A drawback in using the XML for Jabber is that it is not the most efficient coding system, but there is a price to pay for flexibility and human readability. The protocol allows many of the features that proprietary protocols do, such as storing of messages at the server for delayed delivery in case the user is not online, join chat rooms or browsing other user directories. Anyway, it is not purpose of this document describing such features, as they are dependent from the Jabber client used. There is also support for security, and sessions can be executed over SSL.

The actual major drawback of Jabber is that is not so widespread, compared to proprietary instant messaging protocols such as ICQ, MSN or Yahoo that can count on million of users, which makes interoperation difficult. Jabber does provide means for interoperation, not implemented inside the protocol, but through the use of gateway that translate the messages from Jabber format to the one of the remote user. The problem in this is that the translation can happen only by means of reverse engineering work on the proprietary protocol, as the code is most often not publicly available, and thus

the interoperability cannot be full. Moreover, the firms holding the rights for legacy instant messaging protocols are very unwilling to favor interoperability among protocols, as their aim is to have as many users as possible using their protocol for messaging. However, the problem of interoperability is general, and does not regard Jabber only. That is why IETF, chartering the IMPP working group, is trying to define a standard protocol to have a unique base for instant messaging and presence services. The advantages and disadvantages of Jabber are summarized in Table 1 below.

| Advantages | Disadvantages |
|---|---|
| Fully open source | Not so widespread as some legacy protocols |
| Flexible thanks to XML | XML heavy message coding |
| Most of the more important IM features supported | |
| Support for security and interoperability | |
| Scalable as there is no a centralized server | |
| Simple clients implementation | |
| Addressing system e-mail like | |

**Table 1. Advantages and disadvantages of Jabber**

# 10 Other Protocols

## 10.1 TOC

TOC [TOC98] was created by AOL to enable unofficial clients to connect to the AIM service. The protocol is built on TCP. There is usually only one connection made in a normal session. Passwords are sent using a weak encryption algorithm. Since TOC is not the official AIM protocol and no official clients have ever used it, TOC protocol is not kept up with the official protocol in term of functionality and it does not support some popular features already quite widespread. TOC is an old protocol and is being depreciated in favor of Oscar.

TOC can be considered a subset of OSCAR. All TOC commands also appear in OSCAR, but there are some OSCAR commands not deployed in TOC. For example, commands such as file transfer have been implemented in official clients, but are not part of TOC. The user must sign in the protocol and then it has several functionalities available, such as sending and receiving messages, managing its

buddy list and chat rooms, reformatting its username, changing its password, searching for other users, etc. Since in general, the client and server use completely different formats for their communication, special care must be paid to the parsing of commands.

The protocol is currently ASCII based, and you must take care when sending and receiving message to the separators. The separator and the rules of separation are different for messages in bound to TOC and out bound to the client. For example, in client to server communication, it is necessary to remember to put quotes around arguments that include spaces and to use a backspace before dollar signs, square brackets, parentheses, quotes and backslashes. When sending commands to the server the client will not get a response back confirming that the command format was correct or not. But in some cases if the command format was incorrect the connection will be dropped. Server to client communication does not encode in this manner, instead, it only allows confusing characters to exist in the last field of each command, where their purpose as plaintext is obvious.

The TOC server is built mainly to service the TIC and TiK clients. Since the TIC client is a Java applet, TOC does not support multiple TOC protocol versions at the same time. Hence, TiK users will be required to upgrade if the protocol version changes. TOC sends down the protocol version it expects the client will use.

## 10.2 Napster

Napster is a protocol for sharing files between users. With Napster, the files stay on the client machine, never passing through the server. The server provides the ability to search for particular files and initiate a direct transfer between the clients. In addition, chat forums similar to IRC are available. OpenNap extends the Napster protocol to allow sharing of any media type, and the ability to link servers together [OpenN].
Napster uses TCP for client to server communication. Typically the servers run on ports 8888 and 7777. Each message to/from the server is in the form of <length><type><data>, where <length> and <type> are 2 bytes each. <length> specifies the length in bytes of the <data> portion of the message. The <data> portion of the message is a plain ASCII string.

File transfer is directly between clients without passing through the server. The file transfer can occur in 4 different ways: *upload, download, firewalled upload and firewalled download*. The normal method of transfer is that the client who wants to download a file makes a TCP connection to the client holding the file on their data port. However, in the case where the client sharing the file is

behind a firewall, it is necessary for them to "push" the data by making a TCP connection to the downloader's data port.

Commonly the downloading client first issues either a *search request* command to the server or asks directly for the list of the files shared by a specific client through the *browse* command. The server returns a list of files and information on the client sharing the file. To request a download, the client sends a *get request* to the server and the server responds with a *get ack* containing more detailed information.

If the *get ack* says that the remote clients data port is 0 and the sharing client is not firewalled (*normal downloading*), the client can request the remote client to send the data to its own data port and a TCP connection to the data port specified in the *get ack* message from the server is built. The remote client should accept the connection and immediately send one ASCII char, `1'. After this the client can send a request for the file it wants to download. Then the remote client returns the file size and following it the data stream, or an error message if the file is not available. Once the data transfer is initiated, the downloader should inform the server about the beginning and the ending of the transfer.

In the case the sharing client is firewalled (*firewalled downloading*), which means that it can only make an outgoing TCP connection because of firewalls blocking incoming messages, the downloader sends an *alternate download request* message to the server that sends to the uploader an *alternate download ack* Upon the reception of the ack the uploader can make a TCP connection to the downloader's data port specified in the ack message. Hence the downloader's client sends one byte, the ASCII character `1'. The uploader should then send the string "SEND" in a single packet, and then its napster user name and some file information. At this point the transfer can start.

There is also the possibility of a direct client-to-client browsing of file lists. To request a browse, a client uses the *direct browse request* command. The server then sends the same request to the client that is getting browsed with the nick of the client that is requesting the browse. If the client accepts the browse request, it sends back a *direct browse accept* to the server with the nick of the client requesting the browse. The server then informs the requesting client. The browsing client then makes a TCP connection to the remote client's data port and thus can receive the list of files being shared.

## 10.3 Zephyr

The Zephyr protocol [zep], also known as “Project Athena Zephyr notification system” has been designed at MIT to allow inter campus instant messaging service, such as printer job notifications and server status warnings, but it is valuable for external purposes too. A Zephyr client is build putting together several blocks of client each performing different dedicated functions.

Unlike most of instant messaging protocols, Zephyr messages are exchanged over UDP and in order to keep messages reliable a "host manager" is employed. Every machine that acts as either a client or a server for Zephyr must be running one and only one instance of the Zephyr host manager. Thus we can identify three different kinds of entities participating in the zephyr protocol: *servers, host managers and client programs*]. The servers maintain a database of subscriptions and locations for each user who uses Zephyr. The servers stay in contact with one another and provide a reliable backup system in case of network failures. Each client machine on the network runs a host manager program that is the link between the Zephyr servers and the client programs. We can also distinguish two different types of message format: displayable and control messages. Generally the client programs send displayable messages to the host manager that forwards them to the nearest server for action. The host manager is responsible for ensuring that the notices reach a server and for finding a new one if its server fails to respond. The server discards displayable messages if it receives them from a source port different from the host manger port, 2104. Afterwards the server sends the notice to the receiving client programs without passing through the host manager. The control messages may originate from any party. The client program sends control messages to the host manager, except for status request messages that can be sent directly to the server. The host manager forwards the control messages to the server that replies eventually sending back its acknowledgement to the host manager.

In Figure 8 we represent the Zephyr Protocol architecture. We don not explain all the details and the meaning of the messages exchanged. Our aim is just to recognize here the main entities described before. Each user on the network usually runs a WindowGram client program automatically upon login.

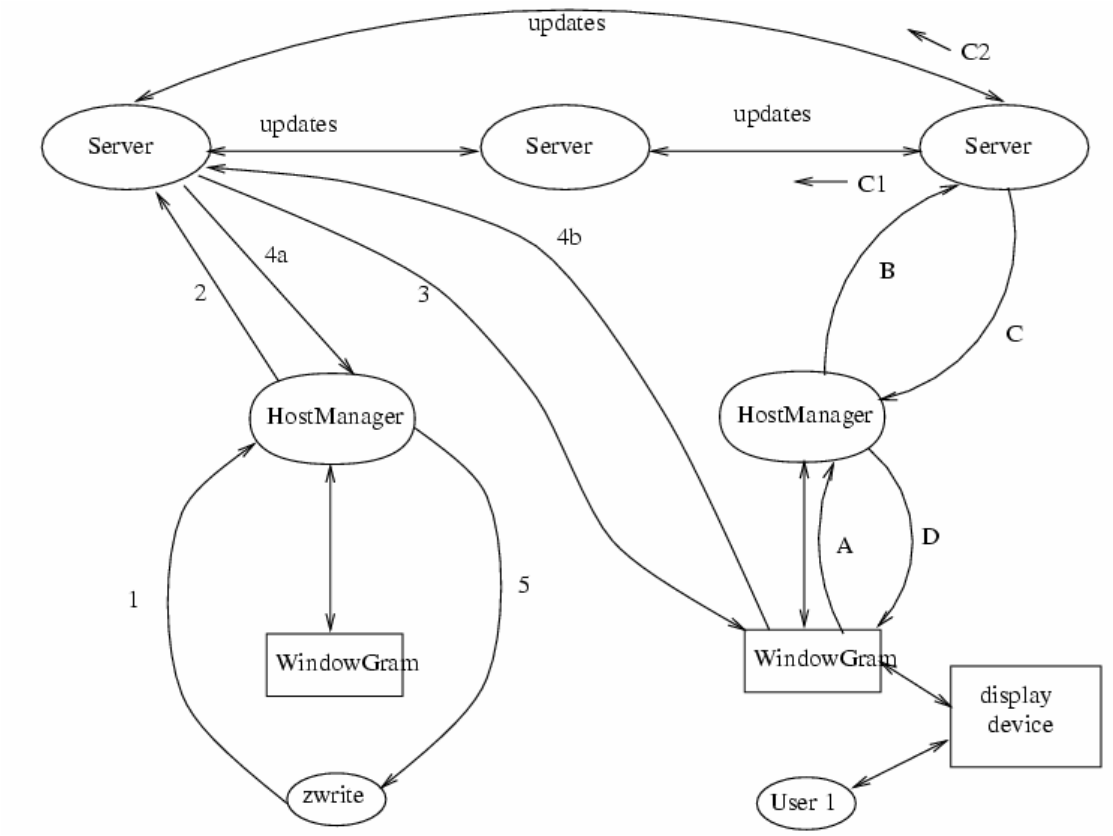



**Figure 8. Architecture of Zephyr protocol [zep].**

# 11 The Internet Message Access Protocol

The Internet Message Access Protocol (IMAP) [imap] permits to e-mail client applications to access and subsequently modify e-mails messages stored in a mail server, in a way that the user perceives the e-mail messages stored in the server as if they were local. This protocol enhances the functionalities provided by the nowadays highly widespread Post Office Protocol (POP) [pop] by providing other than the offline access to e-mail, as done by POP, online and disconnected access too.

The reason that lead to the development of IMAP lies in the fact that most users nowadays, and the number is constantly increasing, need to have access to their e-mail messages from more than a single machine, being more and more common situations where one wants to read e-mail first from a machine at the office, afterwards from home or, why not, from a laptop on the move. In such a scenario, it is not clearly advisable to download e-mails to one machine removing it from the mail server and having to copy it from machine to machine, thing that represents the major drawback of POP. IMAP was designed indeed mainly to overcame this limitation and meet the needs of a growing

number of users to have e-mail always available wherever they connect from as if it was on the local machine.

However, this is not the only improvement to POP protocol carried by IMAP; other features include support for carrying messages in MIME format, as defined in [RFC2822] and [RFC2045]RFC2045, the possibility to personalize own mailbox by creating and manipulating folders, handling folders nested one inside the other and bandwidth optimization for IMAP use in wireless links, by allowing a selective download of the desired part of messages only (e.g. just the header, just header and body and not the attachment and so on). IMAP is not intended to actually deliver e-mail messages, but is only used by e-mail clients to access mailbox in a remote server.

## 11.1 The mail access paradigms

Offline, online and disconnected are the three e-mail access paradigms, defined in [RFC1733], which differ from the physical location where the user elaborates his electronic mail messages. In the offline mode, the only supported by POP protocol, the message are downloaded from the server where they are stored, hence deleted from the server and manipulated locally. In the online mode this is done on the remote server, with the difference the messages are not deleted when manipulated (unless specifically requested by the user) while the disconnected mode is middle way between the two others and consists basically in downloading a copy of the message to manipulate locally and upload the changes to the mail server to keep consistency; the mail server will always store the main copy of the mail, which mains that the locally manipulated copy is meaningful only locally until the user has not uploaded the changes to the server.

It may seem that POP supports the online mode too, as one of the options of POP e-mail clients allows leaving a copy of the messages in the server. Nevertheless, this does not mean online access to the mail as POP does not allow managing own mail folders but simply provides access to the mailbox and support for message downloading, nor is designed for bandwidth optimization for use in wireless links as does not permit to select which part of the message to download; instead this must be done at once for each message, even if it comprises a heavy attachment, which is not clearly suitable if one is attached on a low bandwidth capable link.

## 11.2 Description of IMAP features

The additional capabilities, other than the support for online access, carried out by IMAP can be divided into three categories [Gray95]:

- Remote folder manipulation
- Multiple folder support
- Online performance optimization

POP protocol does not support any of these capabilities, as its aim is to provide a simple store-and-forward paradigm for retrieving e-mails from the server and subsequently delete them. Features belonging to the first category of capability set, remote folder manipulation, are:

- Ability to append messages to a remote folder: this is the possibility to save messages in other directories than INBOX.
- Ability to set standard and user-defined message status flags: this gives the opportunity to the client program to record message status.
- Support for simultaneous update and update discovery in shared folders: this feature helps to keep consistency when the mailbox is shared among multiple users.
- New mail notification: this service is provided by IMAP server that notifies clients if new mail has arrived for them in their responses, even if the correspondent request was for something else than request of new mail.

Multiple folder support operations include:

- Ability to manipulate remote folders other than INBOX: this property is fundamental to support online and disconnected mode, and it is missing in POP protocol. Messages can be saved in more directories, not only in Inbox, and support for message filtering is comprised too, meaning that incoming messages can be filtered and directed to the proper folder in an automatic way.
- Remote folder management (list/create/delete/rename): this property is a direct consequence of the above property of manipulating remote folders.

- Support for folder hierarchies: this feature allows nesting folders.
- Suitable for accessing non-email data; e.g., NetNews, documents.

Online performance optimization features include:

- Provision for determining message structure without downloading entire message: this feature allows users to know how a message is composed, e.g. if it contains attachments to be downloaded afterwards.
- Selective fetching of individual MIME body parts: this feature is direct consequence of the previous one, and it permits to choose which part of the message has to be downloaded.
- Server-based searching and selection to minimize data transfer: this is a powerful tool to minimize the transfer of data over low bandwidth links. After performing a search, based on the criteria received, the server can send only the messages matching such criteria, thus minimizing the amount of traffic on the link.

For further details on the various features presented, refer to Section VII of [Gray95].

## 11.3 Discussion on IMAP

It is not fair to say that IMAP is better than POP because it allows doing many more nice things. These two protocols cannot be compared as they are meant for different purposes. POP was designed as a protocol to handle offline access to e-mail, and it accomplishes its role in a perfect way; we cannot ask POP to support online or disconnected mode because it was not designed for this. The decision of whether using POP or IMAP depends on what it is needed to do; if a user needs only an offline access to its data, POP is more than enough and using IMAP is excessive.

The best thing in POP is its simplicity, due to the aim for which the protocol is intended. Improvements of the protocol to support more features are discouraged by IETF, which instead invites to use IMAP if more complex functionalities are needed, as IMAP is a flexible protocol that can be extended to support future functionalities. Thus, one major drawback of IMAP is its implementation complexity, but since several are the functionalities supported, such complexity is unavoidable. An important outcome of IMAP performance optimization over wireless links is for its use on mobile phones as it not very likely that users want to download permanently e-mails on their phone, or to

download together with the message heavy attachments as well; the IMAP approach is a good solution in such a situation.

Another disadvantage of IMAP is that, in online mode, all the operation are performed at the remote server and thus the processing load for an IMAP server operating in online mode, which is the most likely case however, is higher than the offline or disconnected mode where the only operation requested at server are request for mail and download. Server storage capacity must be increased as well. Anyway, this is not a major issue nowadays due to the availability of high computing power processors and high capacity storage devices.

## 11.4 The IMAP state machine

The IMAP state machine, illustrated in Figure 9, illustrates the exchanges involved when clients connect to mail servers using the IMAP protocol. The IMAP protocol runs over TCP, preferably, using port 143, thus it is appropriate to speak of establishment of IMAP connection. An IMAP connection consists in an initial greeting from the server, followed by subsequent client/server transactions, often in the form of client request followed by the server response; it is possible also that servers send clients unsolicited messages. The server always sends clients a completion result response to acknowledge clients about the outcome of the operations. Details about the format of the messages and the protocol exchanges are provided in [imap].

Upon establishment of the connection between IMAP client and server, there is a phase of server greeting. In the most likely case that the connection has not been pre-authenticated, then the “Not-authenticated” state is entered, where the user must provide the login information necessary to access to the own mailbox at server. If the information is correct, or if the connection was pre-authenticated, the state “Authenticated” is entered, where the client must choose a mailbox to which access before executing other commands. After mailbox selection, the “Selected” state is entered where the normal protocol steps are performed. The connection is closed in the normal case by the client sending a specific logout command, which will bring the protocol in the “Logout” phase, after which both sides perform the closing connection steps.

The IMAP protocol foreseen for each client command a server response notifying the exitus of the command, which could be OK in case of success, NO in case of operational failure or BAD if some error has occurred, such as for example a bad syntax of client request. Besides this message, the server can send also, depending on the kind of request, a certain amount of data in different messages. In

other words, if the client requests for example to fetch messages in the personal mailbox, the server will reply (unless some error occurs) with the messages stored in the mailbox and after the last one, with a response indicating the completion of the fetch operation.

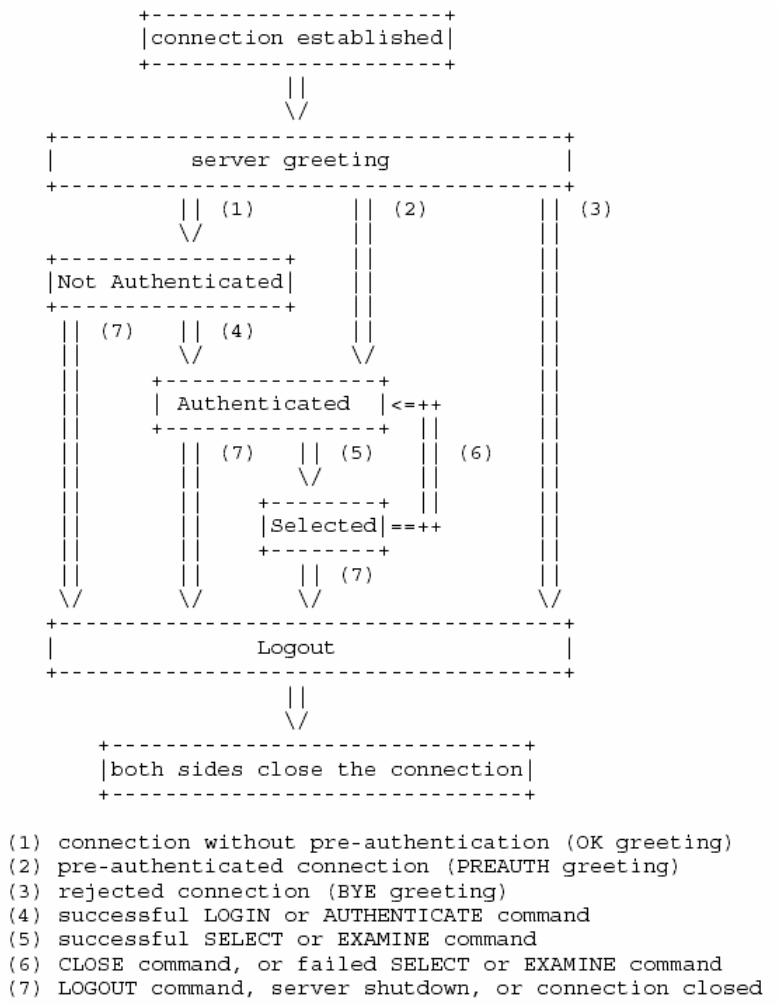



**Figure 9. The IMAP state machine [imap]**

It has to be further deepened the behaviour of IMAP in a wireless environment; the fact that the protocol runs over TCP ensures protection for data loss, segment duplication and reordering, but in case of IMAP message loss, [imap] does not specify clearly the actions taken by the IMAP protocol, nor whether it sufficient to rely on the service provided by TCP.

# 12 Detailed analysis of the selected protocols

This section presents a detailed discussion about the IRC, Jabber and SILC protocols for instant messaging, and IMAP for e-mail access. For each of these protocols, a detailed example of the messages exchanged to set up a session with the purpose of foreseeing how they would behave in a wireless environment. The main reason for choosing these protocols is that they are all open source, facilitating their analysis and possible proposals for improvements.

## 12.1 Detailed analysis of IRC

Starting an IRC connection by the book is a very simple procedure. The client opens a TCP connection to the user specified server and port. After the TCP handshake the client sends two messages, NICK and USER as shown in Figure 10. The server responds with RPL_WELCOME message and after these messages the IRC connection is established. Usually messages from client to server are short, they include only the short command the messages written by the user. Messages from a server to a client are a bit longer as they often include the nickname, user name and the host name of the other client when chatting or the name of the server if messages are announcements from the server.

In preliminary tests the NICK and USER messages were sent in separate TCP packets and the client waited for TCP ACK before sending the USER message after NICK message. This could be avoided by using bigger initial window or by possibly sending both messages in one write-function call. The server responded with RPL_WELCOME message that is usually longer, over 1kB, message informing about the server status and policy. The MSS in preliminary test was 1460 bytes and the whole RPL_WELCOME message could have been sent in one TCP packet but still two TCP packets were sent. With a lower MTU more packets will be naturally used.

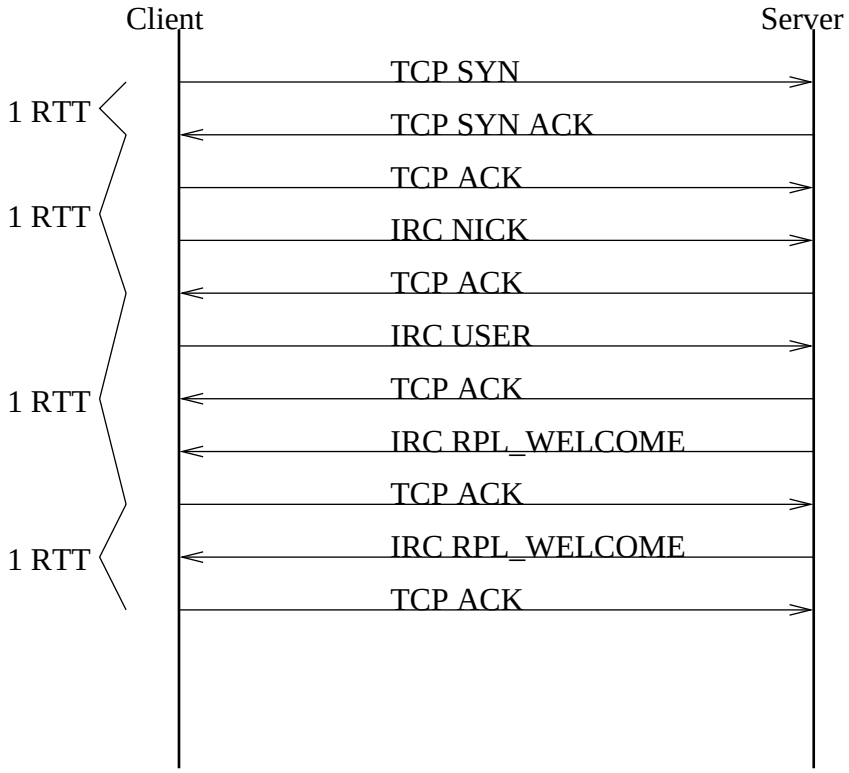



**Figure 10. Connection registration with IRC protocol**

In preliminary tests the connection registration took 4 RTTs (see Figure 10) when measuring only the IRC TCP connection. Actually it took much longer as the IRC server made additional tests that are not specified in RFCs but still commonly used among the IRC servers. The IRC server first checked if the client host had an Auth-service. Using the Auth-service the server asks if the user is really on the host it claims. Next the server checked existence of a proxy (port 3128) or a web cache (port 8080) services. These services are often used by hostile users to hide their real host addresses.

It will take a long time for the IRC server to determine that there are no services in these ports if there is a firewall running in the client host that drops incoming TCP SYN-requests without answering with a TCP reset packet. In this case the IRC server will continue only after it has sent several TCP SYN packets and the TCP connect-function call fails. If there is no firewall or the firewall rejects the TCP SYN packet by sending a TCP reset packet, the TCP connect-call of IRC server fails immediately and the server can continue with the connection registration.

When sending a chat-message from one user to another there are only a little amount of overhead and no additional RTTs. The chat-message sender sends a message that contains the PRIVMSG keyword, nick name of the receiver and the actual message. The receiver gets a message containing the nickname, user name, host name, PRIVMSG keyword and the actual message.

## 12.2 Detailed analysis of SILC

Connection registration is shown in Figure 11. It starts with a SILC Key Exchange Protocol (SKE), which is used to exchange shared secret between connecting entities. First a client sends a SILC_PACKET_KEY_EXCHANGE packet to the server. This packet has a list of all security properties (hash algorithms, encryption algorithms, etc.) that the client supports. The server selects one security property in each of the categories and responds with the same packet listing the selected security properties. Next a Diffie-Hellman key exchange algorithm is executed. Third phase is to authenticate the client to the server and this is done with SILC connection authentication protocol. The client sends authentication data (e.g. pass phrase or certificate) to the server in SILC_PACKET_CONNECTION_AUTH packet and server respond with SILC_PACKET_SUCCESS packet if the authentication succeeded. Finally the client sends client information (e.g. username and nickname) in SILC_PACKET_NEW_CLIENT packet and the server responds giving and internal ID for the client in SILC_PACKET_NEW_ID packet. After this the client can start using the SILC network by sending commands to e.g. enter a channel or to send chat-messages to other users.

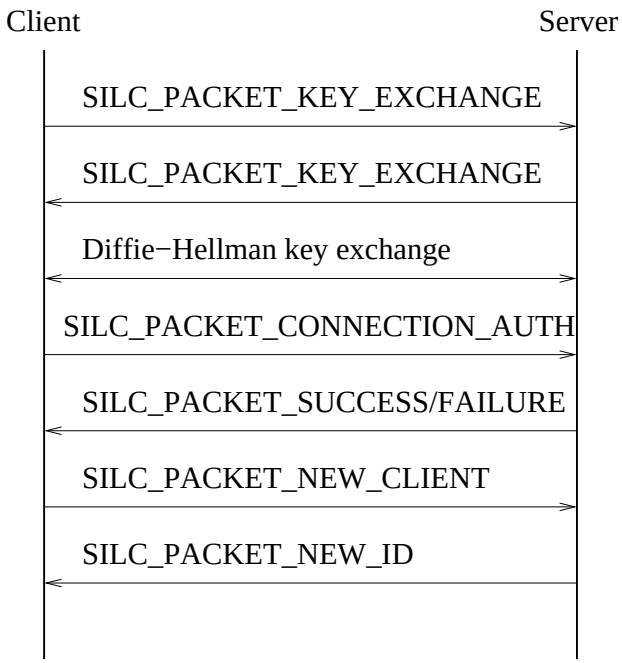



**Figure 11. SILC connection registration.**

Sending a chat-message to other user in the SILC network is straightforward compared to connection registration. The SILC client takes the chat-message from the user and puts it in SILC packet and then encrypts the packet with a session key and sends it to the server. The server responds with

SILC_PACKET_SUCCESS or SILC_PACKET_FAILURE when it has decrypted the message and processed it.

The SILC protocol uses binary coded messages and is therefore more compact than e.g. IRC, which uses ASCII based coding, or Jabber, which uses heavy XML coding. SILC protocol supports also message compression, which can make the messages even smaller. Compression and encrypting/decrypting requires much computing power which can be a problem to small mobile devices.

Because of the security issues there are more packets sent than the actual chatting requires when opening a connection or joining a channel. These packets cannot be easily avoided without compromising the security. The actual chat-message sending does not required additional packets; the acknowledgement for the message is got in one RTT.

## 12.3 Detailed analysis of Jabber

Two documents describe in detail the basic structure of the Jabber protocol; they have been written as Internet drafts in the [XMPP] IETF Working Group. This working group deals with the specification of the XMPP protocol, which is not the original Jabber protocol as it was initially designed, but a slight modification to make it compliant to IETF activities, namely with the requirements posed by [RFC2779]. Particularly, [core] describes the defines the core features of the XMPP protocol, while [imdraft] describes extensions to the XMPP Core to provide the basic functionalities expected from an instant messaging (IM) and presence application, as defined in [RFC2779] indeed. Further extensions to the protocol providing more complex functionalities are described in [].

We will present an example of Jabber message exchange sequence concerning the establishment of a session between a client and a server. It involves the initial opening of the streams in both direction, request for data encryption (optional) and the authentication procedure (obviously mandatory) as well. The message formats are taken from the examples reported in the abovementioned drafts. “C:” indicates messages sent by a client to the server, S: is the label for communication in the opposite direction. It has not been taken into consideration server-server communication, as it is out of our scopes.

1) In order to connect with a server, the client must open a stream with it, by sending an opening <stream> tag (optionally, a line indicating the XML version and the character encoding supported can be present). The streams are carried over a TCP connection on the port 5222.

```
C: <?xml version='1.0'?>                                  //The optional opening line
    <stream:stream                                        //The stream tag
      to='capulet.com'                          //The “to” stream attribute indicates the recipient
      xmlns='jabber:client'                               //Namespace default declaration
      xmlns:stream='http://etherx.jabber.org/streams'     //Stream namespace declaration
      version='1.0'>                                      //The “version”  attribute.
```

The stream element, in fact, must possess two name space declarations, beginning with “xmlns”. The default one indicates that the stream is in a client-server communication. The stream namespace declaration is mandatory according to the protocol specifications.

2) The server replies to the client, opening a stream in the opposite direction:

```
S: <?xml version='1.0'?>
    <stream:stream
      from='capulet.com'
      id='id_123456789'
      xmlns='jabber:client'
      xmlns:stream='http://etherx.jabber.org/streams'
      version='1.0'>
```

The meaning of the fields is similar. The “from” attribute indicates the recipient of the first stream opening request (i.e. the server), when the server replies. The optional “id” attribute establishes a session key. For further details on stream attributes and namespaces refer respectively to Section 4.2 and 9.2 of [core].

3) The server, after the phase of streams opening, sends to the client the STARTTLS ([TLS] protocol in fact is the only mandatory to implement technology for data confidentiality) extension along with authentication mechanisms and any other stream features supported.

```
S: <stream:features>
   <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'>
     <required/>
   </starttls>
   <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
     <mechanism>DIGEST-MD5</mechanism>
     <mechanism>PLAIN</mechanism>
   </mechanisms>
```

</stream:features>

In the message are present the starttls feature, together with the qualifying namespace, adding a <required/> element as child to the <starttls/> one to signal that TLS is required for interaction with the server.

4) The client sends the STARTTLS command to the server:

C: <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>

5) Server informs client to proceed or if an error has occurred informs client that TLS negotiation has failed and closes both stream and TCP connection:

S: <proceed xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
or
S: <failure xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
</stream:stream>

6) The steps to complete a TLS negotiation, over the existing TCP connection, can now be performed. If the TLS negotiation is successful, the client compulsorily initiates a new stream by sending an opening XML stream header to the server.

C: <stream:stream
xmlns='jabber:client'
xmlns:stream='http://etherx.jabber.org/streams'
to='capulet.com'
version='1.0'>

7-8) Server responds by sending a stream header to client along with any remaining (not the starttls) negotiable stream features:

S: <stream:stream
xmlns='jabber:client'
xmlns:stream='http://etherx.jabber.org/streams'
from='capulet.com'
id='12345678'
version='1.0'>
S: <stream:features>
<mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
<mechanism>DIGEST-MD5</mechanism>
<mechanism>PLAIN</mechanism>
<mechanism>EXTERNAL</mechanism>

```
    </mechanisms>
  </stream:features>
```

Now the authentication phase can begin. The specification foresees that this cannot happen unless the above described data encryption phase has ended. The authentication method that must be supported is the SASL DIGEST-MD5 one [SASL].

9) Client selects an authentication mechanism:

```
C: <auth
      xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
       mechanism='DIGEST-MD5'/>
```

10) If necessary, the server challenges the client entity by sending a <challenge/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace:

```
S: <challenge xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
      bm9uY2U9Imd4MFAwMVJSVm9PZkpWcklKNmFOa0ZsaW5oOW5NL1VwOWRhel
      ZURlRxMWM9IixyZWFsbT0iY2FwdWxldC5jb20iLHFvcD0iYXV0aCxhdXRo
      LWludCxhdXRoLWNvbmYiLGNpcGhlcj0icmM0LTQwLHJjNC01NixyYzQsZG
      VzLDNkZXMiLG1heGJ1Zj0yMDQ4LGNoYXJzZXQ9dXRmLTgsYWxnb3JpdGht
      PW1kNS1zZXNz
    </challenge>
```

11) Client responds to the challenge:

```
C:  <response xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
      dXNlcm5hbWU9InJvYiIscmVhbG09ImNhdGFjbHlzbS5jeCIsbm9uY2U9Ik
      9BNk1HOXRFUUdtMmhoIixjbm9uY2U9Ik9BNk1IWGg2VnFUclJrIixuYz0w
      MDAwMDAwMSxb3A9YXV0aCxkaWdlc3QtdXJpPSJ4bXBwL2NhdGFjbHlzbS
      5jeCIscmVzcG9uc2U9ZDM4OGRhZDkwZDRiYmQ3NjBhMTUyMzIxZjIxNDNh
      ZjcsY2hhcnNldD11dGYtOCxhdXRoemlkPSJyb2JAY2F0YWNseXNtLmN4L2
      15UmVzb3VyY2Ui
    </response>
```

This series of challenge/response pairs continues until either:

1. The client aborts the handshake by sending an <abort/> element to the server.
2. The server reports failure of the handshake by sending a <failure/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace to the client.

3. The server reports success of the handshake by sending a <success/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace to the client.

12) Server informs client of successful (or failed) authentication:

S: <success xmlns='urn:ietf:params:xml:ns:xmpp-sasl'/>

13) Client initiates a new stream to server:

```
C: <stream:stream
     xmlns='jabber:client'
     xmlns:stream='http://etherx.jabber.org/streams'
     to='capulet.com'
     version='1.0'>
```

14) Server responds by sending a stream header to client, with the stream already authenticated (not followed by further stream features):

```
S: <stream:stream
     xmlns='jabber:client'
     xmlns:stream='http://etherx.jabber.org/streams'
     id='12345678'
     from='capulet.com'
     version='1.0'>
<stream:features/>
```

15) Once a client has authenticated with a server and has authorized a full JID, it should request that the server activates an instant messaging session for the client:

```
C: <iq type='set' id='sess_1'>
    <session xmlns='urn:ietf:params:xml:ns:xmpp-session'/>
  </iq>
```

16) The server informs the client that a session has been created (or alternately about the failure, by reporting the reason of failure):

```
S: <iq type='result' id='sess_1'/>
```

or

```
S: <iq type='error' id='sess_1'>
    <session xmlns='urn:ietf:params:xml:ns:xmpp-session'/>
    <error type='wait'>
      <internal-server-error
```

```
      xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

At this point the message sequence for establishing a session is complete and the client can perform the desired operations (messages exchange, presence notifications and so on).
Figure 12 and Figure 13 synthesize the sequence of messages exchanged, in the encryption phase and in the authentication phase, respectively.

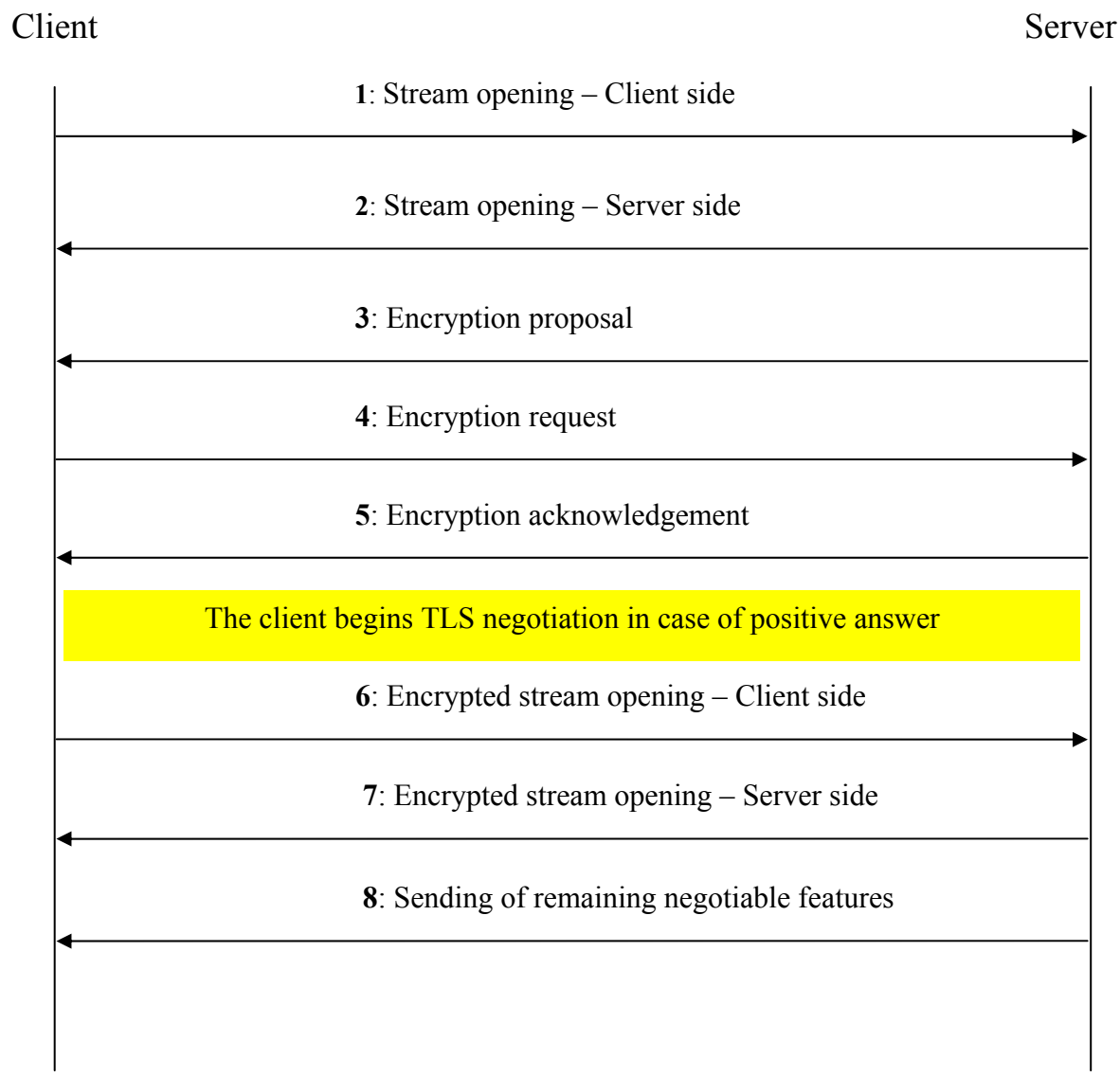



**Figure 12: Message exchange sequence - Encryption phase**

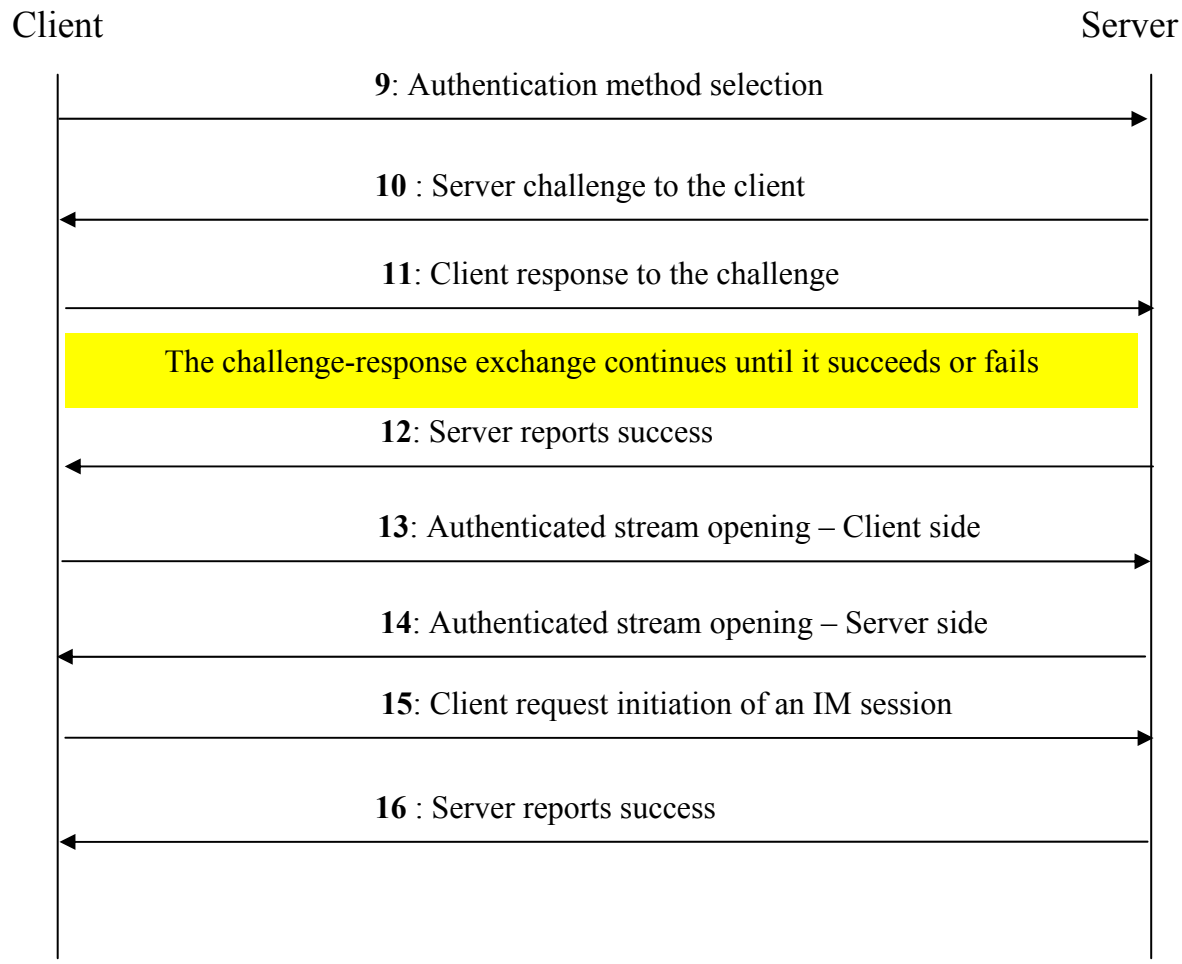



**Figure 13. Message exchange sequence - Authentication phase**

### 12.3.1 Discussion on the deployment of XMPP in a wireless link

This section aims to analysing and predict the behaviour of the Jabber protocol (or XMPP, which is practically the same) when the last-hop link is wireless. Particularly, we will try to predict the behaviour of the protocol when the above described message sequence is exchanged over a wireless link. The metrics over which such behaviour is analysed are:

- Bandwidth
- Latency
- Reliability

Additional interest is posed on the behaviour in disconnected environments, when the link connection is lost and again regained within more or less short time intervals. Such aspect is strictly related with

the reliability, as the major drawback of losing connection is eventually losing packets, and handling lost packets is a matter of reliability. The XMPP protocol faces reliability problems in the most obvious way, relaying on the services provided by the underlying TCP protocol, which will thus ensure management for lost, duplicate or reordered packets. This is an issue to consider, as the protocol is asynchronous, and allows sending consecutive packets to the remote part; a counterexample comes from the ICQ protocol, which is synchronous, and allows sending another message only after an answer to the previous one has been received. This may lead to a communication not much real-time as promised by instant messaging services, especially when a slow link is in the end-to-end path. From this point of view, the choice of realizing an asynchronous protocol is correct, as it is a better solution for wireless links.

Latency is not a major issue, meaning that the timing constraints are not so strict as they would be for a multi media application, such as real-time video or Voice over IP. Of course, as stated above, since the service is of "instant" messaging, users expect to exchange messages with their interlocutor in a relatively small amount of time, but variations in this delay may be accepted and tolerated, on the contrary of multi media applications. We consider two main components for the total latency of message delivery: the propagation time, which is not dependent on the kind of message carried, but is a characteristic of the link, time that can be high especially when dealing with wireless as it is our case, and the transmission time, dependent from the bandwidth of the link and from the size of the message.

As noticed above, the overall latency is thus strictly related to available bandwidth, as low bandwidth is often meaning of high latency; the most likely employment, in a wireless link, of instant messaging services, is chatting over GPRS, at least until UMTS will be commonly widespread. In case of GPRS, the bandwidth available is relatively low, and this may cause the latency for carried messages to increase. In such a case, it would be desirable to keep the size of the messages as small as possible, by means, for example, of efficient coding. From this point of view, XMPP, employing XML for its messages, completely fails the goal, as XML formatting is heavy and thus inefficient when bandwidth saving is an issue. The choice of XML has been done because of the advantages it carries out, like flexibility and human readability; a good compromise, thus, could be to compress XMPP messages so that the messages can be carried in a more bandwidth friendly fashion. However, it is to verify whether the complexity introduced by compression algorithms is justified by a considerable increase of performance.

In order to predict the behaviour of XMPP over wireless links, we will make operating assumptions. We suppose thus that the messages are carried over GPRS, that the bandwidth available is 42 Kbit/sec

(correspondent to two downlink channels of 21 Kbit/sec) and that latency is 1000ms (according to values reported by [CP02]). The values chosen are merely exemplifications, as bandwidth and latencies in GPRS links are highly variable (for example, downlink and uplink latencies are different); nevertheless, for this preliminary rough analysis, we need to fix some values in order to draw some conclusion. We will concentrate only in the wireless link, neglecting the wired path comprised between the access router and the Jabber server itself.

### 12.3.2 Analysis of session establishment sequence

As first method to carry out the analysis, we count how many characters have a XMPP message, supposing that one character is one byte, and then we try to draw conclusions from this. Of course such an evaluation would be highly empirical as the number of characters, and thus of bytes needed depends on whether some optional feature has been included, on the names of entities involved and so on. We also count the characters that do not change at all for each message (for example, namespace identifiers, name of types and attributes and so on). The message number 1 in the sequence, which we report for simplicity, is:

<?xml version='1.0'?> = 21 fixed chars
<stream:stream = 14 fixed chars
to='capulet.com'= 5 fixed and 11 variable chars
xmlns='jabber:client' = 21 fixed chars
xmlns:stream='http://etherx.jabber.org/streams' = 47 fixed chars
version='1.0'> = 14 fixed chars

The first line is optional, but is (as well as the entire second line) part of the “fixed” message. The third line has as fixed parts only “to=” and the two opening and closing tags. Let’s say thus that the first message has:

- Fixed characters: 122 chars
- Variable characters: 11 chars

This makes a total of 133 bytes, of which 122 are needed regardless of the recipient of the message, almost 92% of the total size of the message. Again we recall that this is an example, and that some lines are optional. To this value is to be added the overhead introduced by any underlying protocol, such as TCP/IP. Let’s neglect this overhead, the transmission time for a 133 byte message in a 42 Kbit/sec link is: 133 bytes / 42 Kbit/sec = 25 msec.

2) Message number 2. Reasoning similarly we obtain:

- Fixed characters: 129
- Variable characters: 23
- Total characters: 152
- % of fixed chars: 85%
- Transmission time: 29 msec

3) Message number 3.

- Fixed characters: 219
- Variable characters: 15
- Total characters: 234
- % of fixed chars: 94%
- Transmission time: 45 msec

The size of this message, for example, depends on how many mechanisms are supported and on their name.

4) Message number 4.

- Fixed characters: 51
- Variable characters: 0
- Total characters: 51
- % of fixed chars: 100%
- Transmission time: 10 msec

We have considered the namespace declaration as fixed, for all the messages exchanged.

5) Message number 5.

- Fixed characters: 50
- Variable characters: 0
- Total characters: 50
- % of fixed chars: 100%
- Transmission time: 10 msec

6) Message number 6.

- Fixed characters: 101

- Variable characters: 11
- Total characters: 122
- % of fixed chars: 83%
- Transmission time: 23 msec

7) Message number 7.

- Fixed characters: 129
- Variable characters: 23
- Total characters: 152
- % of fixed chars: 85%
- Transmission time: 29 msec

8) Message number 8.

- Fixed characters: 170
- Variable characters: 23
- Total characters: 193
- % of fixed chars: 88%
- Transmission time: 37 msec

9) Message number 9.

- Fixed characters: 59
- Variable characters: 10
- Total characters: 69
- % of fixed chars: 86%
- Transmission time: 13 msec

10) Message number 10.

- Fixed characters: 64
- Variable characters: 244
- Total characters: 308
- % of fixed chars: 21%
- Transmission time: 59 msec

11) Message number 11.

- Fixed characters: 62

- Variable characters: 304
- Total characters: 366
- % of fixed chars: 17%
- Transmission time: 70 msec

12) Message number 12.

- Fixed characters: 51
- Variable characters: 0
- Total characters: 51
- % of fixed chars: 100%
- Transmission time: 10 msec

13) Message number 13.

- Fixed characters: 101
- Variable characters: 11
- Total characters: 122
- % of fixed chars: 83%
- Transmission time: 23 msec

14) Message number 14.

- Fixed characters: 126
- Variable characters: 19
- Total characters: 145
- % of fixed chars: 87%
- Transmission time: 28 msec

15) Message number 15.

- Fixed characters: 80
- Variable characters: 6
- Total characters: 86
- % of fixed chars: 93%
- Transmission time: 16 msec

16) Message number 16.

- Fixed characters: 25

- Variable characters: 6
- Total characters: 31
- % of fixed chars: 81%
- Transmission time: 6 msec

The total transmission time depurated of overhead from any lower level protocol is thus 433 msec. To this time has to be added the propagation time. Thus the total one-way delay from the client to the access router (different from the Jabber server) is D = 16 * prop-del + 433 msec.

### 12.3.3 Behaviour in disconnected environments and in case of packets losses

[core] does not specify a particular behaviour in case of loss of packets, it is thus to believe that all the related operations are handled at TCP level. If for example, during the phase of session establishment, message 9 gets lost (we recall that message 9 was sent by the client to the server in order to select an authentication mechanism) it is to assume that the sequence is stopped until the server gets the message with which a method of authentication is selected. In this sense, the session establishment phase can be considered sort of "synchronous", as every message is related to the previously received one.

More complex is the situation where the client, running on a mobile terminal, loses its connection with the network. Let's try to understand what can happen. If network connectivity is lost, then the client (or the server, depending on which entity has to send the message) keeps on resending the message until it receives an ack, according to TCP operations. If the break in connectivity is short, then the outcome can be simply a delayed establishment of session; however, if connection is lost for a longer time, after a certain number of retransmissions (implementation dependent) the TCP connection will be lost and thus also the XML streams, carried over the TCP connection, will be lost, and the session establishment sequence has to be restarted.

## 12.4 Detailed analysis of the Internet Message Access Protocol

The main document that describes the structure and the messages carried by the IMAP protocol is RFC 3501 [imap]. The document provides a brief overview of protocol functionalities, and proceeds by listing the set of commands that can be sent by clients and the subsequent responses of servers. Note that a server can unilaterally send responses to clients, which must be always prepared to process any unexpected data coming from the server. The transport protocol over which IMAP messages are

carried is the reliable TCP, on the port 143; therefore, for IMAP are valid the same reliability considerations already carried out for the XMPP (Jabber) protocol in Section 12.2.

An IMAP connection is instantiated when a client connects to an IMAP server using TCP port 143; subsequently, there is a "greeting" phase where the server communicates to the client that it is ready to process its commands. Afterwards, the normal steps of the protocol consist in the client sending tagged commands to the server, which replies by sending responses. Responses can be of two types: untagged or tagged. The second kind is used to report the outcome of a command, and such a response has the same tag of the command sent by the client to which it refers. Untagged responses, prefixed by the indicator "*", are used for the actual carrying of data sent by server, as result of a client command or after an unilateral decision of server (for example sending a notification for new mail). IMAP messages have the form of strings with an ending CRLF; that is they consist in textual commands and responses, using 7-bit US-ASCII coding as general rule. Nevertheless, IMAP supports also 8-bit characters and binary coding, according to [RFC2045] directives.

An IMAP transaction thus consist generally in a command send by the client and at least two responses by the server, one for carrying the data, and for reporting the outcome. An example could be:

```
C: abcd CAPABILITY
S: * CAPABILITY IMAP4rev1 STARTTLS AUTH=GSSAPI LOGINDISABLED
S: abcd OK CAPABILITY completed
```

The command, with tag `abcd,` requests a list of the capabilities supported by the server; the untagged response carries the list of capabilities, and the tagged one, with a tag matching the command to which it refers, reports the completion of the transaction. Nevertheless, also commands requiring no actual data sending do exist, in such a case the only response will be the tagged one as well as commands requiring more that one data line sent by the server. We will now discuss an example of IMAP message exchange, reported in Section 8 of [imap].
This is a rather complete example of IMAP operations, as it comprises the authentication phase, some inquires sent by the client to the server and eventually the logout phase. The meaning of each message is explained below. For messages number 5 and 9, our understanding from [imap] is that the server sends several message lines (in the case of message "5", 6 lines, for example); this seems rather inefficient, as a single line could be sent, even if each line has a different meaning. A better understanding will be achieved if the actual protocol connection will be tested with experimental test runs. Message 7 is a single line broken for clarity reasons.

```
1)    S: * OK IMAP4rev1 Service Ready
2)    C: a001 login mrc secret
3)    S: a001 OK LOGIN completed
```

```
4)    C: a002 select inbox
5)    S: * 18 EXISTS
      S: * FLAGS (\Answered \Flagged \Deleted \Seen \Draft)
      S: * 2 RECENT
      S: * OK [UNSEEN 17] Message 17 is the first unseen message
      S: * OK [UIDVALIDITY 3857529045] UIDs valid
      S: a002 OK [READ-WRITE] SELECT completed
6)    C: a003 fetch 12 full
7)    S: * 12 FETCH (FLAGS (\Seen) INTERNALDATE "17-Jul-1996 02:44:25 -
      0700"
      RFC822.SIZE 4286 ENVELOPE ("Wed, 17 Jul 1996 02:23:25 -0700 (PDT)"
      "IMAP4rev1 WG mtg summary and minutes"
      (("Terry Gray" NIL "gray" "cac.washington.edu"))
      (("Terry Gray" NIL "gray" "cac.washington.edu"))
      (("Terry Gray" NIL "gray" "cac.washington.edu"))
      ((NIL NIL "imap" "cac.washington.edu"))
      ((NIL NIL "minutes" "CNRI.Reston.VA.US")
      ("John Klensin" NIL "KLENSIN" "MIT.EDU")) NIL NIL
      "<B27397-0100000@cac.washington.edu>")
      BODY ("TEXT" "PLAIN" ("CHARSET" "US-ASCII") NIL NIL "7BIT" 3028
      92))
      S: a003 OK FETCH completed
8)    C: a004 fetch 12 body[header]
9)    S: * 12 FETCH (BODY[HEADER] {342}
      S: Date: Wed, 17 Jul 1996 02:23:25 -0700 (PDT)
      S: From: Terry Gray <gray@cac.washington.edu>
      S: Subject: IMAP4rev1 WG mtg summary and minutes
      S: To: imap@cac.washington.edu
      S: cc: minutes@CNRI.Reston.VA.US, John Klensin <KLENSIN@MIT.EDU>
      S: Message-Id: <B27397-0100000@cac.washington.edu>
      S: MIME-Version: 1.0
      S: Content-Type: TEXT/PLAIN; CHARSET=US-ASCII
      S:
      S: )
      S: a004 OK FETCH completed
10)   C: a005 store 12 +flags \deleted
11)   S: * 12 FETCH (FLAGS (\Seen \Deleted))
      S: a005 OK +FLAGS completed
12)   C: a006 logout
13)   S: * BYE IMAP4rev1 server terminating connection
      S: a006 OK LOGOUT completed
```

1) This is the greeting message, with which the server informs the client it is ready to begin a session, after that the client opened a connection over the TCP port 143.

2) The client logins to the server, with the “login” command. The use of such a command is obviously discouraged, as login information is carried in plain text format. However, this messages sequence has only exemplification purposes. Note that the client generated a tag for the command.

3) The server does not need to send any data back to the client, but it just verifies the login information provided and replies with a tagged response with value matching the one received with the message 2. If the login provided was incorrect, the server would have replied with a NO response, and a textual explanation of the failure. This mechanism is common for whatever situation of failure reported by server.

4) With this message the client selects a mailbox (in the example inbox) to which access. Only a mailbox can be accessed for each connection; this means that if a client wants to access to two different mailboxes it has to open two different connections.

5) This message, comprising several lines, is the answer that the server provides when it receives the SELECT command. The [imap] specification, in fact, states that the answer to such a command must compulsorily comprise several features describing the status of the user mailbox before the tagged reply indicating successful processing of the command. See Section 6.3.1 of [imap] for further details.

6) The client requests data for the specified message.

7) The server replies, before with the actual data and after with the completion tagged response.

8) The client requests the header of the message number 12.

9) The server replies with the requested data.

10) The client asks to mark the message number 12 as deleted.

11) The server sends marks the message and sends the updated status of the message with another FETCH response. The reply for a FETCH or STORE command is always a FETCH response.

12) The client logs out.

13) The server sends the BYE message and closes TCP connection after sending the tagged response.

A visualization of the above exchange sequence is reported in Figure 14 below. For brevity, only one FETCH command and response has been reported.

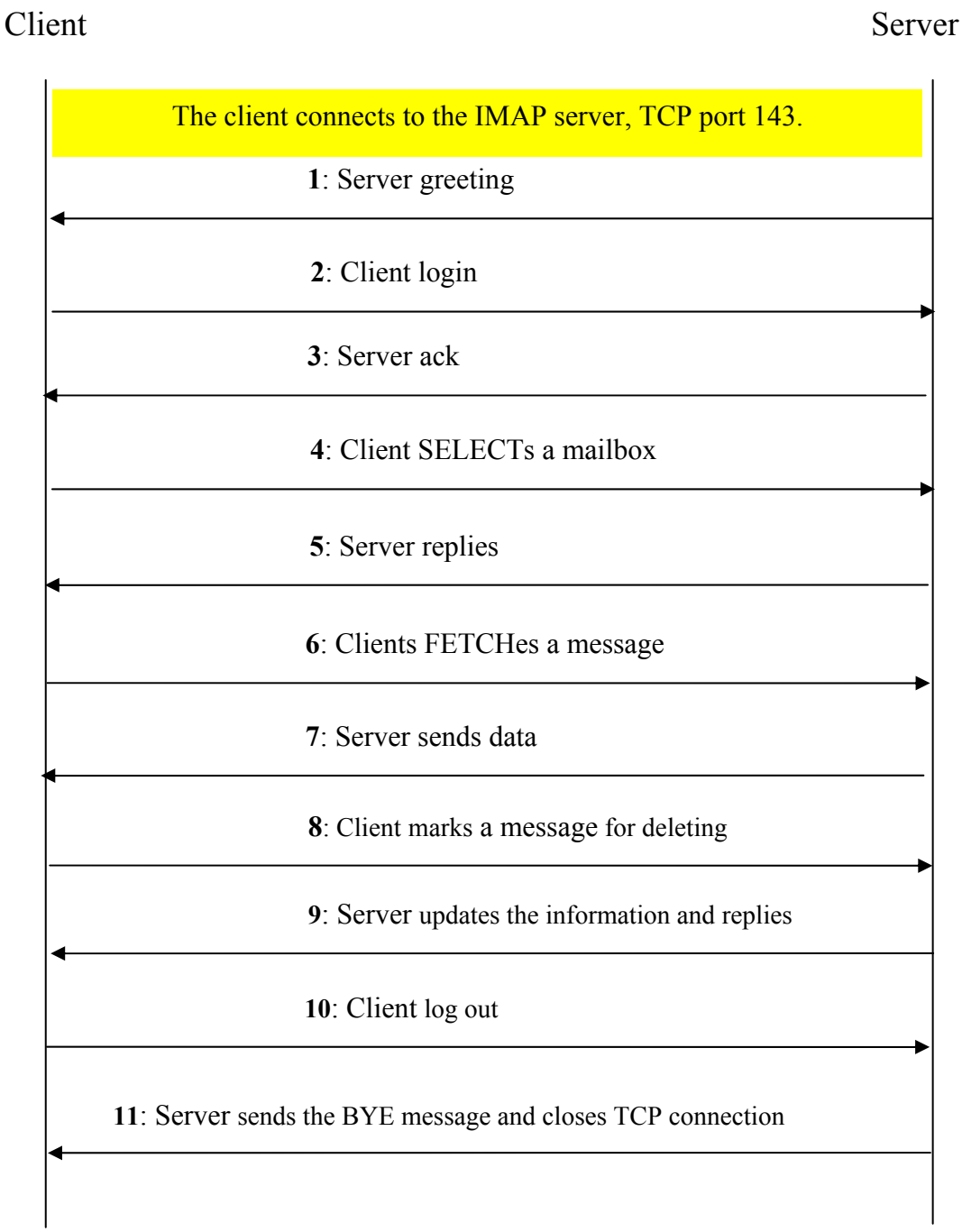



**Figure 14. Example of an IMAP message exchange**

### 12.4.1 Discussion on the deployment of IMAP in a wireless link

This section aims to analysing and predict the behaviour of the IMAP protocol when the last hop between a client and the server is a wireless link. The analysis will follow the schema already

presented for Jabber in 12.3.1 and the behaviour of IMAP will be compared over metrics of latency, reliability and bandwidth consumption. Interest will be paid in the behaviour of the protocol in situation of poor connectivity as well.

The IMAP protocol faces reliability problems in the most obvious way, relaying on the services provided by the underlying TCP protocol, which will this ensure management for lost, duplicate or reordered segments. The last aspect is very important, as it is not possible to bind the protocol neither to the “asynchronous” category nor to the “synchronous” one; although clients are allowed to send multiple consecutive commands without waiting for the correspondent server response, on the other hand for other commands they are obliged to wait that to receive from the server a sort of “authorization” to proceed and send another command (or complete to send the previous one).

One could think that latency is not an issue for a mail access protocol, but this is not true. IMAP has a different role than SMTP, for which the timeliness of messages delivery is really non relevant. IMAP users expect to handle their mailboxes as if they were locally stored in their computer, so if they request to display a message and such a operation takes a long time, then the goals of IMAP are not satisfactorily met. This may be an issue for a wireless link, although in most cases IMAP users in a wireless link expect a certain slowness in completing the requested operation.

The IMAP is optimised for saving bandwidth, as already noticed, by allowing to selectively download only some parts of the message. Users connected through a slow wireless link can avoid downloading heavy attachments. From an operational point of view, the choice of sending multiple responses to certain commands (such as SELECT) instead of a single message reporting all the needed information, seems to lead to a waste of bandwidth, as more overhead is needed for sending the same amount of data. On the other hand, the messages are shorter, being split in more lines, and this is an advantage in a slow link as it reduces consistently the transmission time. For example, the sequence of messages 9 is formed by several short lines; it could have been used a single long line to carry the same information: this reduces the overhead that lower layers protocol headers introduce but increases the transmission time as the size of the message is bigger.

The messages are also text coded, even if support for transferring binary data is present too; however, the overhead of a text based coding, although certainly heavier than binary coding, is not as high as in case of the XML based XMPP protocol. Most of the lines are in fact, as noticed above, relatively short (around 25 bytes long), and the heaviest messages are those sent by the server carrying the actual data requested. The longest line in the example sequence is the number 7, around 500 bytes, sent by the server as response to a client command requesting to fetch FULL information about a message.

### 12.4.2 Behaviour in disconnected environments and in case of packet losses

[imap] does not specify a particular behaviour in case of loss of packets, it is thus to believe that all the related operations are handled at TCP level. The reasoning is very similar to the same carried out for the XMPP protocol in Section 12.3.3. If for example, during the phase of login, message 2 gets lost (we recall that message 2 was the plain text login sent by the client to the server) it is to assume that the sequence is stopped until the server gets the message with the login information. Section 5.5 of [imap] deals with this topic and reports few example of allowed and forbidden sequences of commands and responses. In this sense, the authentication phase can be considered sort of "synchronous", as every message is related to the previously received one.

More complex is the situation where the client, running on a mobile terminal, loses its connection with the network. If network connectivity is lost, then the client (or the server, depending on which entity has to send the message) keeps on resending the message until it receives an ack, according to TCP operations. If the break in connectivity is short, then the outcome can be simply a delayed response received by the client; however, if connection is lost for a longer time, after a certain number of retransmissions (implementation dependent) the TCP connection will be lost and thus the client will need to reconnect from scratch to the server, once the connection has been regained at link level, opening the TCP connection and performing all the prescribed steps.

# 13 Protocols Comparison

This section has presented an introduction to the most popular instant messaging protocols used nowadays in the Internet, evidencing the main features, the system architecture supporting them and the main strengths and flaws of each of them. Furthermore we have provided an introduction to the IMAP protocol, used to achieve online access to electronic mailboxes. The most common instant messaging protocols are proprietary and the vendors do not make available the source for such protocols, and all the studies and clients implementations related are consequences of reverse engineering work. The result is that in most cases, the functionalities of the protocol cannot be fully utilized. The problem is aggravated by the fact that each proprietary company likes that as many users as possible utilize their protocol, so interoperation work among such protocols is often difficult.

There exist some free source architectures, like IRC or Jabber, but they do not reach the same impressive amount of users than the proprietary ones. Their advantage comes from all the benefits that

an open source solution carries out. The major strength of ICQ is the great set of functionalities that it provides to its users, one example for all, directory sharing, but on the other side this opens security flaws. The choice of having a synchronous communication may lead to unnecessary time spent to wait acknowledgements before sending a new message, and the direct communication between users is a problem for those who chat behind a firewall, which may block the incoming connections. Finally, it is not open source, with all the related consequences.

MSN, although it is not open source as well, addresses some of the problems presented by ICQ, allowing asynchronous communication, server based communication and it is generally well designed. However, the protocol is pretty complex and does not provide such a rich set of functionalities as ICQ. Yahoo also provides a very common instant messaging protocol; unfortunately the lack of information available about does not allow us to write more. Finally AOL, based on Oscar protocol, although is the most widespread chat protocols is also closed source and the reverse-engineered information available is really limited.

With regard to the open source protocols, IRC is pretty old, there are RFCs describing it, thus it can be considered well understood, and anyway allows to design simple clients; however, since it was designed long time ago, it allows only a text based communication, which is something clearly unsatisfactory for the nowadays needs. The SILC protocol is similar to IRC, but it allows also secure communication between peers, and does not provide pure text based message exchange. A strength of SILC, from the bandwidth optimisation point of view, is that it is binary coded, especially compared to IRC, ASCII encoded, or XMPP-Jabber, which is even XML coded. A major limitation of SILC seems to be its really poor diffusion.

Regardless of the heavy XML coding, Jabber seems a promising instant messaging system, which, with its full open source approach, both on server and on client side, allows good comprehension of the overall architecture. Among the strengths of Jabber, it is to mention the possibility of undertake encrypted session, by means of SSL secured communication and the great scalability of its server network configuration, as every association can build its server, in an e-mail like fashion, and clients need to know only about the server where they are registered. It will be task of server contact the other party server to complete the communication; this approach leads to a general simplicity in clients' implementation. Jabber also allows interoperability among instant messaging protocols, by executing translations between Jabber and the other protocol. However, it is not at all widespread as the proprietary protocols, and due to its XML message data coding, its messages can be sometimes pretty heavy and consume much bandwidth. Table 2 summarizes what discussed above.

| Protocol | Advantages | Disadvantages |
|---|---|---|
| **ICQ** | Rich set of functionalities<br>Widespread protocol | Not open source<br>No support for interoperability<br>Synchronous communication<br>Security issues, firewall problems<br>Only one centralized server<br>Complex client implementation |
| **MSN** | Asynchronous communication<br>Few firewall problems<br>Load distribution on servers<br>Widespread protocol | Not open source<br>No support for interoperability<br>Complex Protocol |
| **Jabber** | Open source<br>Support for interoperability<br>Simple client implementation<br>Security support<br>Scalability | Heavy XML coding<br>Not so widespread |
| **IRC** | Open source<br>Asynchronous communication<br>Simple client implementation | Only text-mode supported<br>No authentication<br>Not so widespread |
| **OSCAR** | Widespread Protocol<br>Load distribution on servers<br>Security support | Not open source |
| **SILC** | Security support<br>Binary coded | Not widespread |

**Table 2. Comparison of the most popular chat protocols**

With regard to the mail access protocols, the IMAP protocol introduces many benefits respect to its older counterpart, the POP protocol. These benefits range from a more complete set of functionalities to, thing that is really the major enhancement respect to POP, the support for online and disconnected access mode; indeed, the set of functionalities added by IMAP is meant specifically to handle the online access mode. A relevant feature of IMAP is its possibility to selectively decide which part of a message to download, useful as it allows to not downloading a heavy attachment when a message is read from a bandwidth limited link. On the contrary, the only negative thing about IMAP seems to be its complexity, especially compared to POP simplicity, but this is unavoidable due to the number of functionalities supported; also the diffusion of IMAP, although not comparable with that of POP, is growing. The only reason of preferring POP to IMAP seems to be if just offline access mode is needed, as it would be excessive to use IMAP only for offline access, where instead POP is the best solution as it offers the best trade-off performance-simplicity. It is to say too, if a decision whether to support online or offline mode has to be takes, that online mode, even if it guarantees a richer set to functionalities than the offline one, poses more load on the server side, as servers are in charge to store users mail and execute all the necessary processing to handle the requests.

# PART II: JABBER TEST REPORT

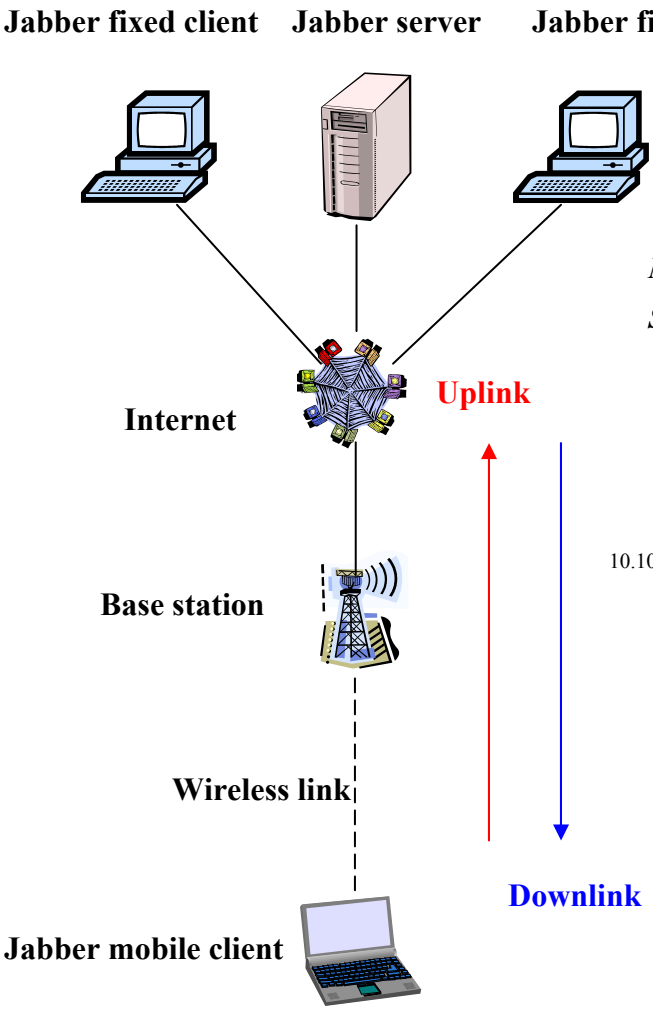
The part II of the document describes and comments the tests run to identify problems in how the Jabber Instant Messaging (IM) platform handles situations where one of the clients connects to the server over a slow, error prone, wireless link. The shift of the market towards the wireless, the growing wish of users to use mobile devices not only for traditional voice services, but also for more sophisticated data service, (such as Instant Messaging) motives the choice of such an environment for our tests.

# 14 Test environment

This section describes the general environment of the test runs. Figure 15 shows the target environment (Figure 15a) and the emulation testbed (Figure 15b). We have analyzed Jabber message exchanges between a client that is connected through a wireless link, referred to as mobile client in the rest of the document, and one or more clients accessing over a wired link, deemed as fixed clients. Fixed clients and the Jabber server are arbitrary end points in the Internet. The aim of the tests was to study the effect that packet delays and losses provoke to a Jabber session; wireless links, together with user mobility, are very likely causes of such phenomena.

The general idea behind all the tests is to analyze the flow of packets at TCP level, when a session is ongoing between two Jabber clients, and see how the application level messages are mapped into TCP segments. We emulated scenarios that could be possible in a real life situation where one of the clients in a Jabber session is attached to a wireless link. The test cases analyzed include:

1. The situation when the network delays one of the messages sent by a client or the server.
2. The situation when messages are sent to a client when it is still disconnected from the Jabber server. Such messages will be delivered from the server to the recipient client, when it connects.
3. A combination of the two above: one of the messages that the server is delivering to the client after its connection is delayed.



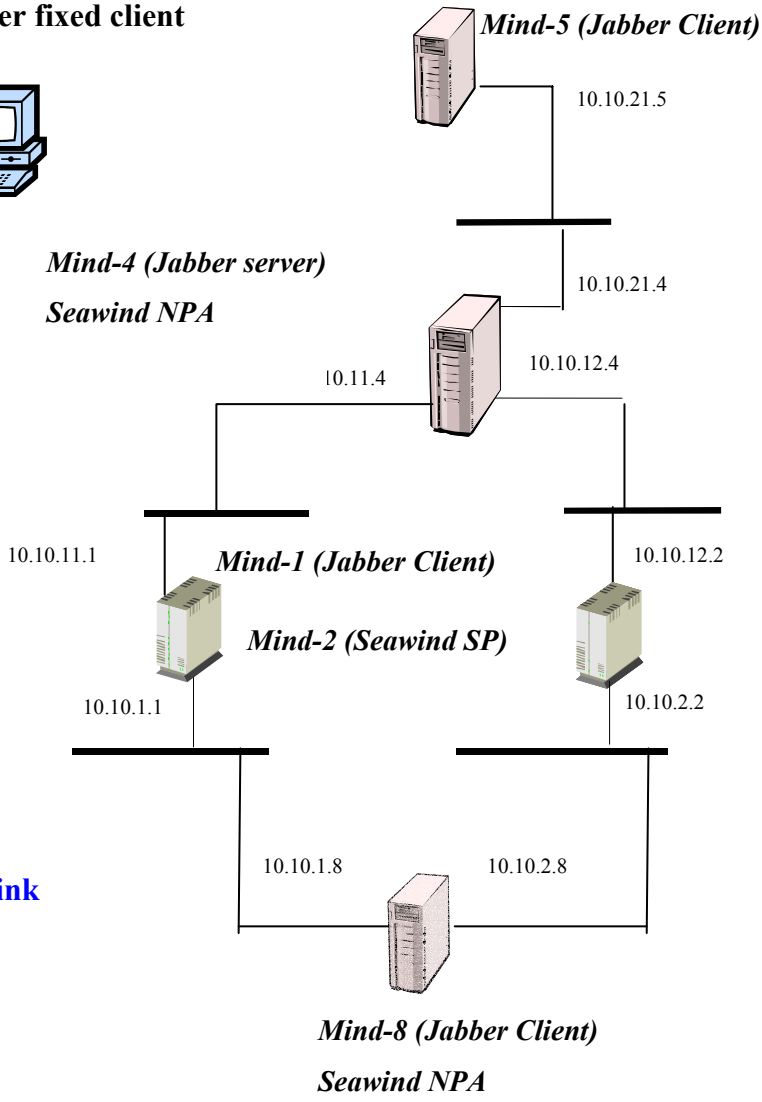




**Figure 15a: Target environment**

**Figure 15b: Emulation testbed**

In all test cases, the clients were connected to the same server. Situations implying inter-server communication, where the clients are registered at different servers, were not studied. The usual configuration for the tests was a one-to-one communication between clients; however, some of the test cases were repeated using two source clients. The Jabber server used is the official server developed by the Jabberd project, jabberd2.0b3 release [jabberd]; some preliminary tests were run also using another version of server: jabberd-1.4.2 [jabberd].

At the moment of starting our tests, the version 1.4.2 of the server was the latest stable released by the Jabber community. However, it is not strictly compliant to the specification of the Jabber protocol, as

it was designed before that the Jabber protocol was raised as issue in the IETF XMPP working group, with the name of XMPP (Instant Messaging and Presence Protocol). The core XMPP protocol [core] is based on the Jabber protocol, developed independently by the Jabber community developers; however, the XMPP extendibility properties extend the main features trying to define a standard platform for instant messaging and presence [imdraft]. The maintenance of the old 1.x server will be discontinued, in favour of the jabberd2 server, whose beta3 version was in use in our tests. This server was totally rewritten by developers of the Jabberd project to be strictly compliant to the protocol specification, and to the extensions that are being proposed, and this was the reason for having chosen it as server for the tests. Currently, a stable version for Jabberd 2 has been released, but it fixes only minor bugs, not affecting the results of our tests, compared to the one we have used.

With reference to the testbed, the Jabber server is running in Mind-4. Three identical Jabber clients (Gaim-0.57 [Proto]) were installed in Mind-5, Mind-8 and Mind-1; refer to each test case for details on their specific utilization. In brief, the mobile Jabber client is Mind-8, while the fixed Jabber clients are Mind-5 and Mind-1, which was used only for test runs with two source clients. In order to emulate the wireless link, Seawind was utilized. The SP machine was in Mind-2, and the two NPAs in Mind-4 and Mind-8. The wireless link has a bandwidth of 28.8 Kbps, to approximate GPRS rate and a propagation delay of 200 msec. No particular metrics were defined before tests execution. After choosing a set of tests cases that could resemble a real life scenario, we analyzed misbehaviors, or at least incoherent sequences of messages exchange, to draw out conclusions and locate eventual problems. What misbehavior or incoherent exchanges exactly means, will be clear analyzing each test case in detail.

During the tests, the terms *uplink* and *downlink* will be extensively used. Uplink, or upstream, refer to data transfer from the mobile client to the fixed client, thus in the direction shown by the red arrow in Figure 15; downlink, or downstream, indicates the opposite direction of messages, from the fixed client to the mobile one, spotted by the blue arrow in Figure 15. A common characteristic of all test cases was sending Jabber messages only in one direction, one client being recipient and the other sender for all the duration of the test, which allowed us to better study the message exchange in one particular direction.

## 14.1 Best-case scenario

The best-case scenario is when two Jabber clients can communicate without suffering from any delay or packet losses. In such a situation, preliminary tests show that as soon as the Jabber application

passes one Jabber message to the TCP level, this is immediately sent (because the PUSH flag is set) to the intended TCP receiver. When the TCP receiver of the message gets it, it acknowledges it by sending a TCP ack. It is important to notice that Jabber has no application level reliability mechanism, such as acknowledgements, as it relies on TCP as transport protocol; therefore, in the rest of the document with the term ack it must be considered the TCP ack.
In all the test cases, the initial authentication phase was not included in the analysis, as it is a peculiar moment of the session executed only at the beginning. The phase of exchange of presence information was not treated in detail as well; the reason is that in these tests, the attention was focused in the interactions between Jabber and TCP and improving the phase of presence information exchange would involve changes in the internal mechanism of Jabber protocol. At this stage, we believe there are more margins for improvements in the interaction of the protocol with the TCP level and that analyzing the phase of messages exchange, which takes most of the time in a Jabber session must therefore have priority over the rest. However, we propose some improvement also for the phase of subscription in each other contact list. See Section 20.2 for more details on the preliminary tests run, and for the discussion on the phase of user subscription exchange.

# 15 Test case 1: Delay in the downlink direction

This test case emulates the real life situation where a user accessing from a mobile device opens an instant messaging session with a remote user, buddy in Jabber terminology. We have supposed the remote buddy accessing the network through a fast Ethernet link; the situation of both clients connected through wireless links was never analyzed in the tests, as it is a particular case of the studied scenarios. Figure 16 below shows the emulated network for this test case.

TEST DESCRIPTION: In this test, several messages are typed continuously from the IM fixed client user interface and sent downstream to the mobile client. According to the Jabber protocol, the messages are first routed to the server where the source client is registered, and hence delivered to the intended recipient, after the proper processing. One of the messages forwarded by the server is delayed and does not reach the recipient client “immediately”, causing the server not receiving the TCP ack from the mobile client. It is not important to know where the message is delayed, if in the Internet (due to congestion for example), or in the wireless link; the only assumption in this test case is that the message gets delayed after that the server has delivered it to the recipient, the mobile client. The server does not deliver any subsequent message to the mobile client until it has received the ack for the delayed message.

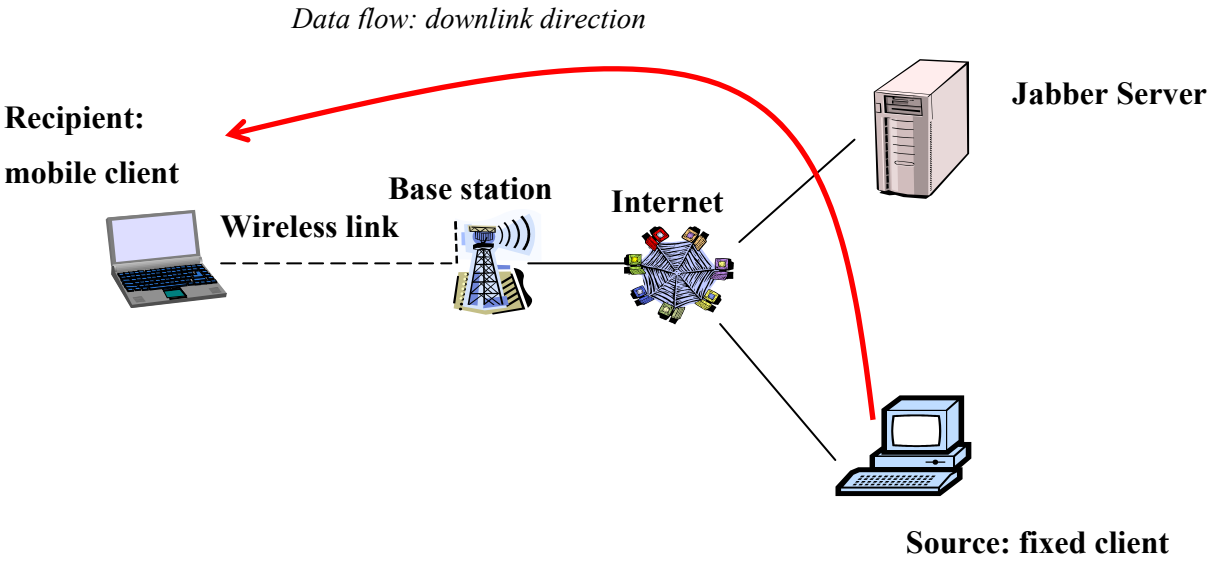



**Figure 16: Emulated network - test case 1**

In all the executed tests, Jabber messages consist simply in numbers, to be able to refer to them in a quick way. The resulting size of the XML-encoded message is of around 90 bytes (for one or two bytes of typed characters) for messages sent from the source client to the server. The server always adds a string indicating the source when delivering the message to the recipient, therefore producing a size of around 150 bytes. The overhead increases even more for test cases where the server delivers messages sent to a disconnected recipient; in this case, besides the source, a timestamp indicating the time of reception is added to the Jabber message, for a total size of around 230 bytes of overhead. This string is an optional feature, as tests effected with another server (openim, Java open source Jabber server, [javaserv]) show that in case of client reconnection the server does not add the timestamp string.

The rate at which messages are sent from the source client, in normal network operating conditions, is very low, around 1.2 Kbit/sec. The Jabber server allows configuring rate-limiting policies so that clients overcoming the configured rate had their message slowed down to respect such limits; we have disabled such policies in the server, so that all the delays were due to network problems and not to server limiting policies.

With reference to Figure 17, in **bold** are the number of the TCP level segments discussed while explaining the test. In the rest of the document we will refer to the bold numbered messages as *segments*, as they indicate TCP level messages carrying one or more Jabber messages or TCP acks.

Instead, with "message" we will indicate an application level message, i.e. Jabber. For example, "**6** – Message 3" refers to the sixth message shown in the flow chart, which carries Jabber message 3. Note that many message exchanges can be not shown for clarity, as they are not of particular importance in the context and that the sequence number is not shown for acks. Sequence numbering has nothing to do with TCP, it is only a monotonically increasing way of numbering messages.
In the following description, the terms source client and fixed will be used interchangeably, as they identify the same client; the same is valid for recipient and mobile client.

- Every Jabber message sent by the source (in one TCP segment) is received and acked (TCP ack) by the server (segment 1 and ack). Not shown in Figure 17, the source client keeps on sending messages for all the test duration. For all of them, the server will send a TCP ack. For the rest of the test, we will neglect such messages and we will concentrate on the recipient client side.
- Server delivers Jabber messages 2-3 to the mobile client (not shown).
- Jabber message number 4, in segment number 3, is the first delayed: the server does not receive ack from the recipient client and TCP timeout expires. The server does not send any following Jabber messages until the ack for Segment 5 is received.
- Server retransmits message number 4 in Segments 4/7. Meanwhile, it receives and acks messages sent by the source client. The first retransmission happens roughly 0.9 seconds after the original transmission. The others are backed off.
- The mobile client finally receives message 4 (Segment 3) and acks it. The ack is received by the server after sending segment 7. So far, the server has received from the fixed client 31 Jabber messages.
- The server sends several Jabber messages packed in a single TCP segment. The amount of application level data inside this segment is the TCP MSS (524 bytes). Since TCP is byte stream, Jabber messages are packed in TCP segments continuously: few complete XML Jabber messages and a part of one message fit in a single segment. For example, in Segment 8, Jabber messages 5-7 and a part of message 8 are packed. Two consecutive MSS sized TCP segments (8-9) are sent (the MTU is 576 bytes). The interval between these two transmissions is very low, not even 1 msec. The interval between the sending of segment 8 and the arrival of the ack to message 4 is < 1 msec as well. Thus we can infer that segment 8 sending was triggered by the arrival of the ack to message 4.
- Four duplicate acks for the retransmitted segments arrive at the server consecutively and are ignored by the server.
- The acks to segments 8 and 9 arrive. They are acknowledged separately because at the mobile client the ack for the segment 8 is sent before the arrival of segment 9. The actual

situation of segments arrival at the mobile client is much more interleaved than Figure 17 shows. However, for reasons of visual clarity it has been chosen this "sequential" ordering.

- The server sends other two MSS TCP segments (10/11) containing from the rest of 12$^{th}$ up to 18 completed messages, and a part of the 19$^{th}$. Again, the interval between transmissions is negligible, less than 1 msec.
- In Segment 13, the server sends the rest of message 19, and message 20, for a total of 299 bytes. No more Jabber messages are packed, even though the MSS has not been reached.
- The recipient client sends three acks for the remaining unacknowledged segments (not shown).
- From now on, the server sends each of the other Jabber messages received from the source (from 21 to 31) in a single TCP segment; each new message is sent after the reception of the ack for the previous one. The interval between two consecutive messages delivery is not negligible, as it happened for consecutive MSS transmissions: for each pair of transmitted messages is about 0.7 seconds. The interval between the arrival of one ack and the sending of the next data segment is variable: it ranges from values of less than 1 msec (when almost all the messages are delivered) up to hundreds of msec (at the beginning of the phase of one Jabber message sent in one TCP segment).

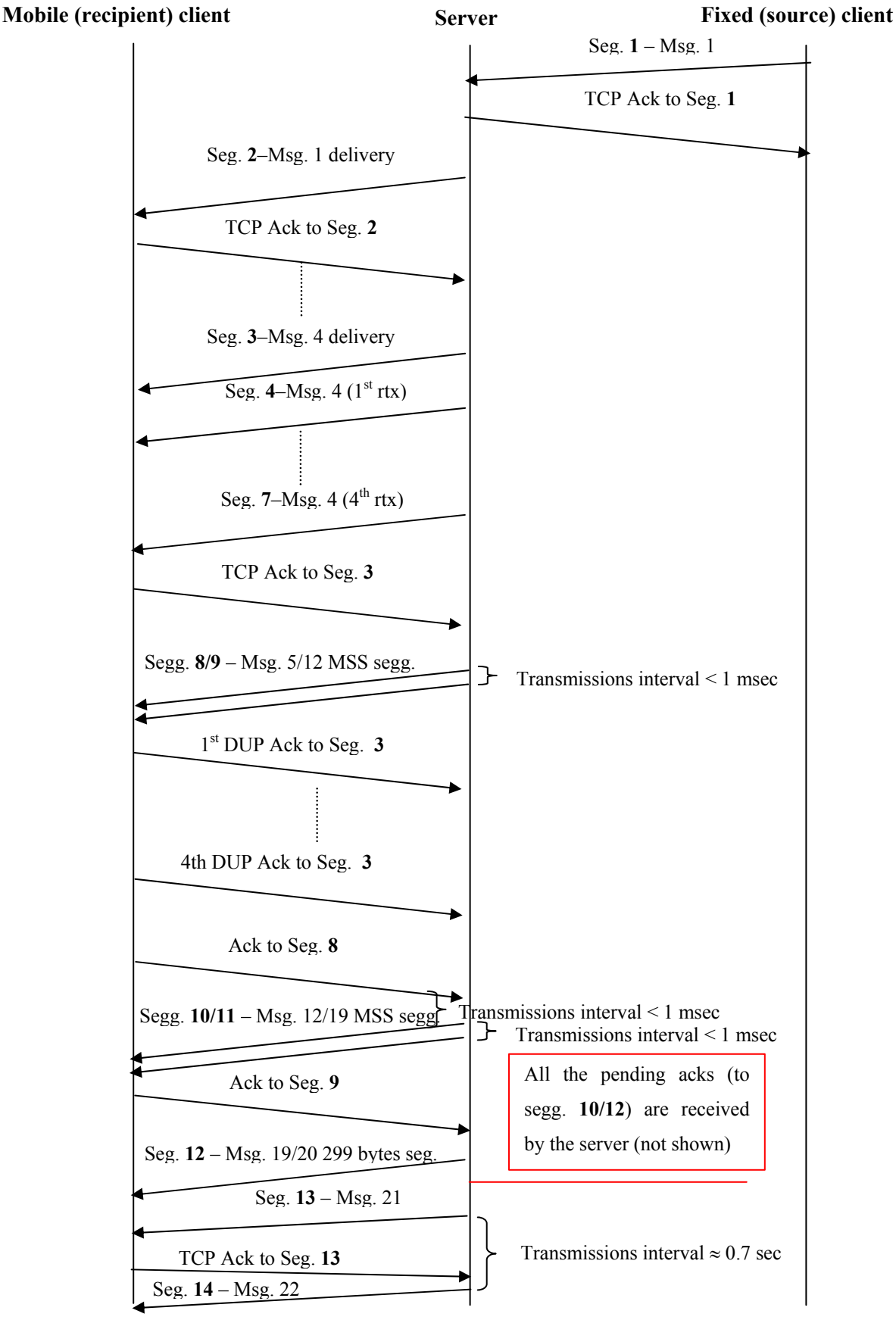



**Figure 17: Test case 1 message exchange flow chart – server interface view**

We will now calculate roughly the rate at which the server transmitted Jabber messages 5-20. The time of transmission of the TCP MSS segment 11 is 21.116272 seconds (from test beginning). Segment 20 is transmitted at 21.854503 seconds. The server has sent in this time interval 4 TCP MSS segments, and one 299-byte segment. To calculate the rates, we have to consider the actual number of bytes on wire, comprising also header overheads, which is 68 bytes (TCP timestamp option was enabled). The total number of transmitted data in this period is therefore 2735 bytes, that is, 21880 bits. The transmission time of segment 20 is

$$T_t = \frac{2936\,\text{bits}}{28800\,\text{bit/sec}} = 0.101944\ \text{seconds}$$

The total time elapsed for transmitting the messages is thus: T = (21.854503 – 21.116272 + 0.101944) seconds = 0.840175 seconds. The average server sending rate is thus:

$$R = \frac{21880}{0.840175} = 26{,}042\ \text{Kbit/sec}$$

Which approximates the wireless link bandwidth.

Another repetition was run for this test, obtaining again the same behaviour: the server packs Jabber messages until at some point, all the pending acknowledgments are received, and afterwards sends message singularly. This test was also repeated having two fixed source clients sending messages to the mobile recipient. The server stopped to send packed messages after that all the pending acks, from both the clients, were received. Refer to section 20.3 for further details on this test case.

NOTE: Jabberd2 server uses a database for storing messages and users personal information (in particular, we have used BerkeleyDB database [berkeley], but other choices, such as MySQL [mySQL] are possible).

The behaviour of the server is due to the Nagle algorithm, which is enabled in the Jabberd2 server. However, since the source client sends up to 31 Jabber messages before the ack for the delayed packet is received the server should send MSS data anyway instead of stopping after 19 messages. Moreover, in test case 3, described in Section 17, the server behaves in a way apparently opposite than this. In test case 3, a client sends messages to another client, when it is disconnected from the server. Upon

recipient client connection, the server begins to deliver all Jabber messages one by one, with evident inefficiency as the amount of data for each TCP segment is very low, around 230 bytes.

The same test case was run with the old server version. In this case, the server packed messages in TCP MSS segments until all the Jabber messages stored before the arrival of the ack for the delayed one were sent. Afterwards, the messages were sent in single TCP segments. The problem with this server was that it reordered Jabber messages. An example clarifies the concept: let's suppose that message 3 is delayed and that up to 5 Jabber messages fit in one TCP MSS. After receiving the ack for message 3, this server sent messages 4-8, which were received in the inverse order: say, 8,7,6,5,4 (packed up to fill TCP MSS). This caused at the recipient client, which does not know anything about the real order of Jabber messages, displaying the messages in the wrong order in the user interface. This behaviour was not present in case of reconnection of a client: in this case, the old server sent all the messages stored packing them in TCP MSS segments, preserving the order in which they were received.
In order to verify the effects of Nagle algorithm, this test case was repeated with a higher propagation delay, 1000msec, which led to a RTT five times higher than the previous. In this case, the server seemed to behave correctly, always packing Jabber messages to fill a TCP MSS. The RTT of the connection seems to play an important factor in this context, but what is desirable is that the server behaves correctly no matter of the duration of the RTT of the connection with a client.

# 16 Test case 2: Delay in the uplink direction

This test case is similar to the previous one, the only difference being that the messages are sent from the mobile client to the fixed one upstream rather than the other way round. Figure 18 below shows the emulated network for this test case. The real life situation this test resembles is that of a mobile device user sending Jabber messages to a buddy far away in the Internet; messages sent by the mobile user can likely experience delays on their way to destination, for examples due to bandwidth shortages of the wireless link or user mobility. The user is not aware that one of the messages has been delayed, and keeps on sending messages to his buddy. This is the situation studied in this test case.

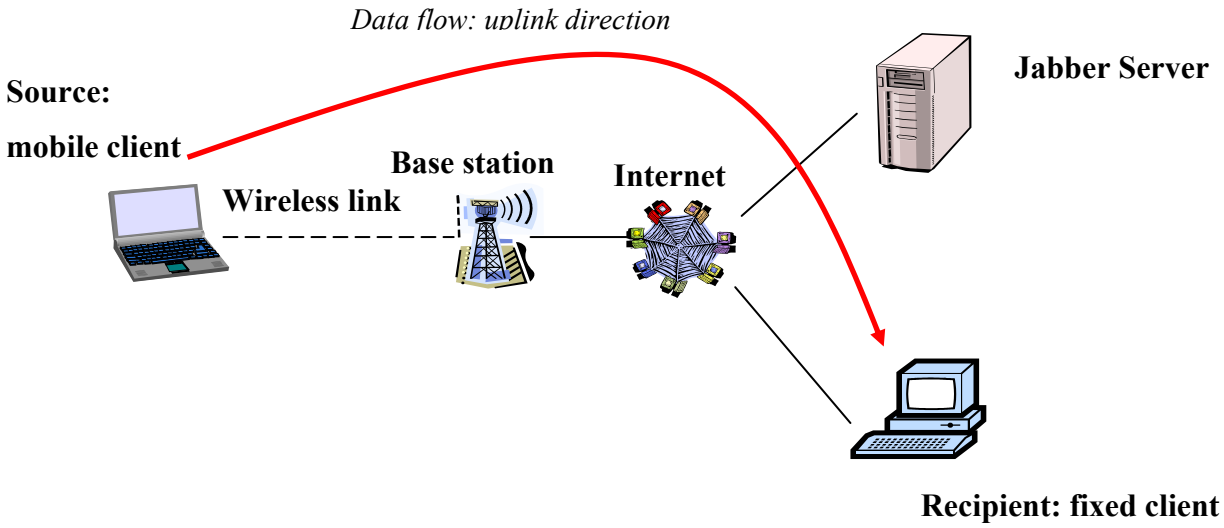



**Figure 18: Emulated network - test case 2**

TEST DESCRIPTION: In this test, several messages are typed continuously from the IM mobile client user interface. One message is delayed in the wireless link on its way to the server, causing the server not sending the TCP ack to the source. The source does not deliver any subsequent message to the server until it has received the ack for the delayed message. When the ack is received, the mobile client sends to the server all the pending messages packing them in MSS TCP segments.
Considerations about size of messages, sending rates, terminology and conventions are similar to what used in test case 1. However, the sequence chart, shown in Figure 19, was drawn according to the messages exchanged through the mobile client interface.

- At the beginning, the link condition is good: the source sends to the server two Jabber messages in two TCP segments, one for each Jabber message. The server acks them and delivers to the recipient fixed client (segments 1-2). The message exchange sequence is shown only for the first Jabber message.
- Jabber message 3 (in segment 3) is delayed. Particularly, it is sent about 4 seconds after the beginning of the test, and arrives at the server after about 19 seconds, delayed of 15 seconds.
- Since the source client does not receive the ack for message 3, it retransmits it. The retransmission is delayed as well and does not reach the server. The source client sends 3 retransmissions of message 3 (segments 4-6).
- The server sends the ack for message 3. This segment is received by the mobile client after the three retransmissions, as shown in Figure 19.
- Message 3 is delivered to the recipient in segment 7 and acknowledged. The retransmissions are correctly NOT delivered to the recipient.

- Jabber messages 4-16 are sent by the client in 2 TCP segments (8 and 9). Each segment carries MSS data. The time interval between these two transmissions is < 1 msec. The interval between the arrival of ack to message 3 and the sending of segment 8 is less than 1 msec. Segment 8 sending was triggered by this ack arrival.
- Three duplicate acks for the retransmitted message 3 (segment 5) are sent by the server.
- The server ack for segment 12, carrying Jabber messages 4-10, is sent by the server after the three DUP ACKS.
- Jabber messages 16-26 are sent by the client in 2 TCP segments (10 and 11). The first segment contains MSS data, while the other 423 bytes. The time interval between these two transmissions is < 1 msec. This means that the mobile client application has delivered up to 26 Jabber messages to the TCP level when the ack for the retransmitted message is received.
- All the unacknowledged segments sent by the mobile client, that is, segments 9/11 are acked by the server.
- From message 27 onwards (segment 12), the communication between source client and server happens regularly. The source client sends one Jabber message in one TCP segment.
- The server delivers to the recipient client all the messages it has previously received, one by one in a single TCP segment, comprised the Jabber messages received "packed" (4-26). The interval between two messages delivery is pretty unpredictable: it ranges from relatively short values of 0.3 seconds to surprisingly high values of almost 2 seconds. Note that the messages were already stored at the server, so the high interval between ack transmissions is not due to the fact that there is nothing to ack, but to inefficient server database access in situations of short RTTs.

The outcome of this test shows again an unacceptable behaviour of the server. It would be desirable in this situation that the messages are delivered in MSS TCP segments to the recipient client. In this test case, the server (mis)behaves differently from the previous one: before in fact messages were packed and hence sent one by one, while now they are always sent one by one. The explanation is in the RTT of the connection client – server: if it is too short, as in this test case (few msec of an Ethernet link between server and client), the server cannot access properly the database for retrieving messages.

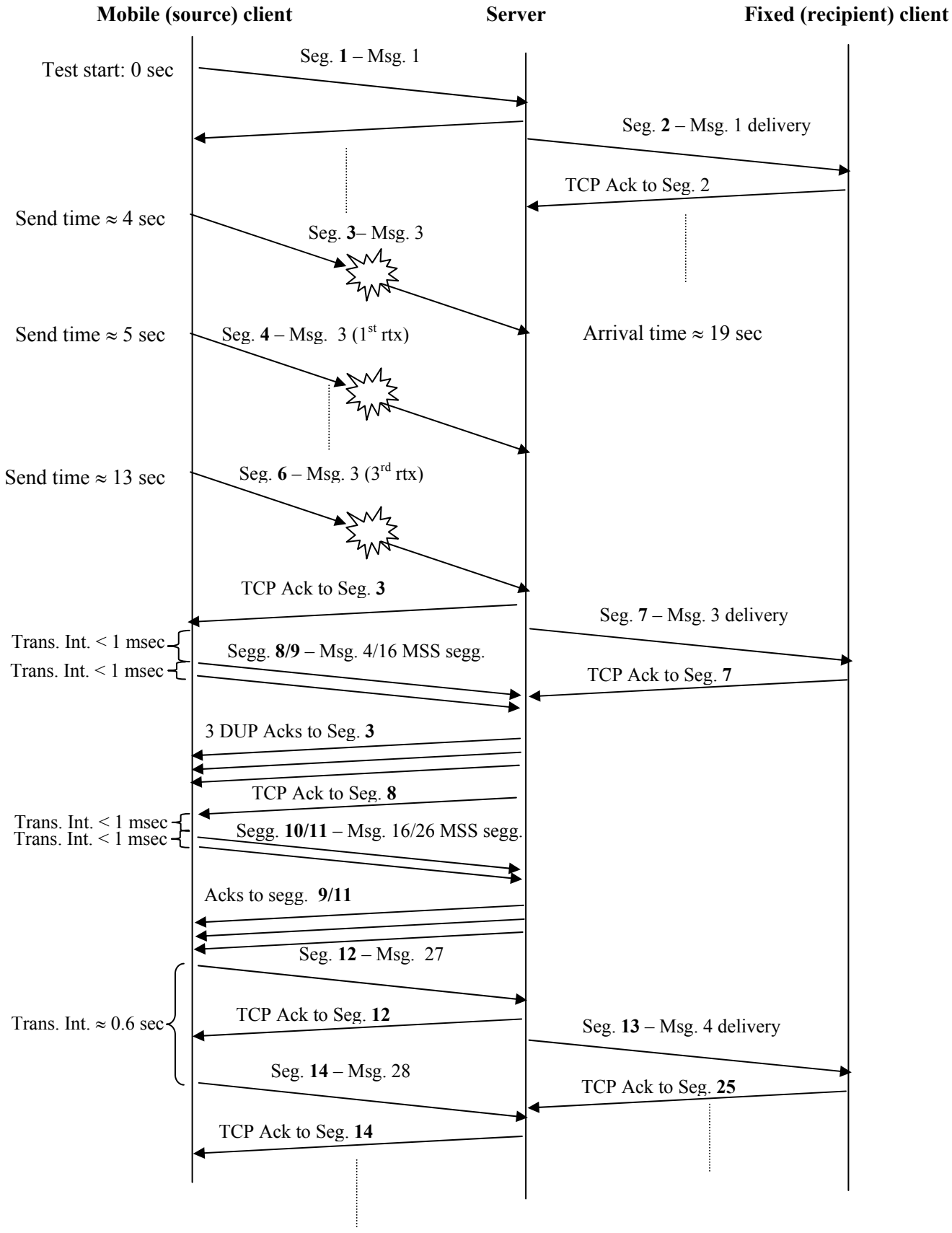



**Figure 19: Test case 2 message exchange flow chart - mobile client interface view**

# 17 Test case 3: Reconnection of a client

This test case emulates a situation when a Jabber user sends messages to a disconnected buddy. The server where the recipient user is registered stores the messages it receives for a disconnected client to deliver them after it has received a presence message from the intended recipient. In this test case, no messages were delayed; it thus applies to the most general possible network environment, under the assumption that the message flow is regular and no retransmissions are experienced. Figure 20 shows the emulated network for this test case.

TEST DESCRIPTION: This test case comprises both cases when messages are sent upstream or downstream. The recipient client (could be the mobile client or the fixed client) is disconnected at the moment of message sending. Upon connection, the server begins to deliver the previously stored messages to the recipient. The behaviour of the server for both directions of the streams is linear: it just delivers one Jabber message in one TCP segment.

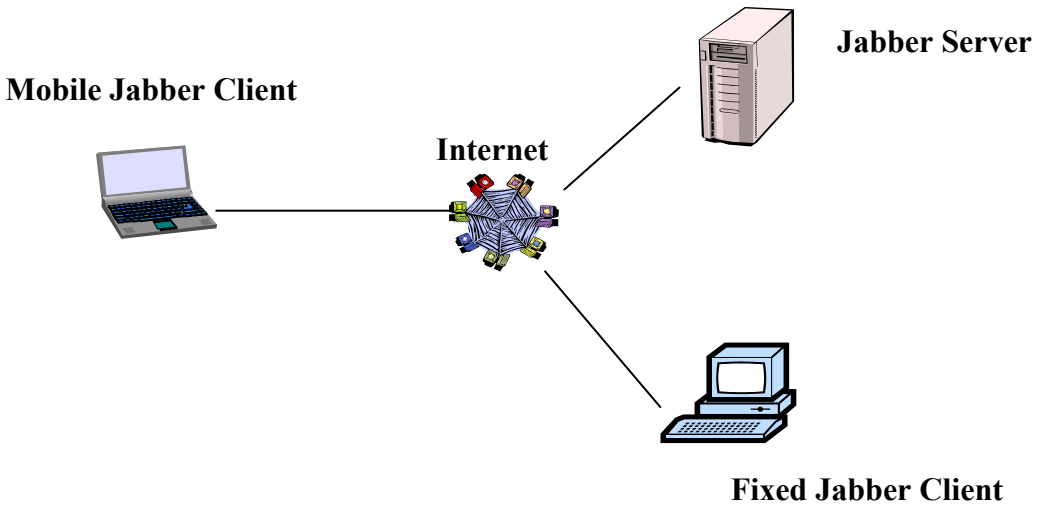



**Figure 20: Emulated network - test case 3**

The outcome of this test is the same, regardless of the fact that the fixed client sends messages when the mobile one is disconnected, or vice versa. The behaviour is very simple: the source client sends one Jabber message in one TCP segment. The server stores the received messages and, after the recipient client completes the authentication procedure, comprising communication of presence information, the server delivers the previously received messages one by one. The interval between two consecutive transmissions is of the order of the tens of msec.

Again, the behaviour of the server is not optimal. In order to deliver to the recipient the 20 messages offline stored, it takes 40 messages, precisely 20 TCP segments and respective acks. The total time elapsed in this message exchange is, from the first message sent by the server to the reconnected client, to the arrival of the ack for the last message, about 5.1 seconds. The same test case was run also with the old server, Jabber-1.4.2: besides the first message, sent in one TCP segment, all the others previously received by the server were sent in MSS TCP segments (1460 bytes), for a total of only 8 segments exchanged, comprised TCP acks, in just about 0.04 seconds, which is the 0.78% of the time the other server took.

This test case was repeated, the fixed client as source client and the mobile as recipient, with an increased value of RTT. The outcome was coherent to the results of test case 1, and the server packed the messages delivered to the newly connected client to fill TCP MSS segments. This is a desirable behaviour, and the server should conform to it even when the RTT of the connection is low.

# 18 Test case 4: Delay in reconnection. Downlink direction

This test case puts together the test cases 1 and 3, described respectively in Sections 15 and 17. It emulates the situation when a client connects to Jabber network, accessing from a wireless link, when the server has some messages to deliver to it. One of the messages (and thus all the following ones) sent by the server is delayed due to say, wireless link impairments or network congestion. The emulated real life network is the same shown in Figure 16 above.

TEST DESCRIPTION: In this test messages are sent downstream from the fixed client to the mobile one. The recipient client is disconnected at the moment of message sending. After its connection, the server begins to deliver Jabber messages. One of them is delayed and does not reach the client “immediately”. The test analyses how server delivers the messages to the newly connected client. The message exchange is analyzed at the server interface.

- The source client sends a total of 25 messages to the server, when the recipient is disconnected. Each message is sent to the server in one TCP segment and acked.
- After the completion of the authentication procedure and the reception of the presence message from the mobile client, the server first sends the first two Jabber messages

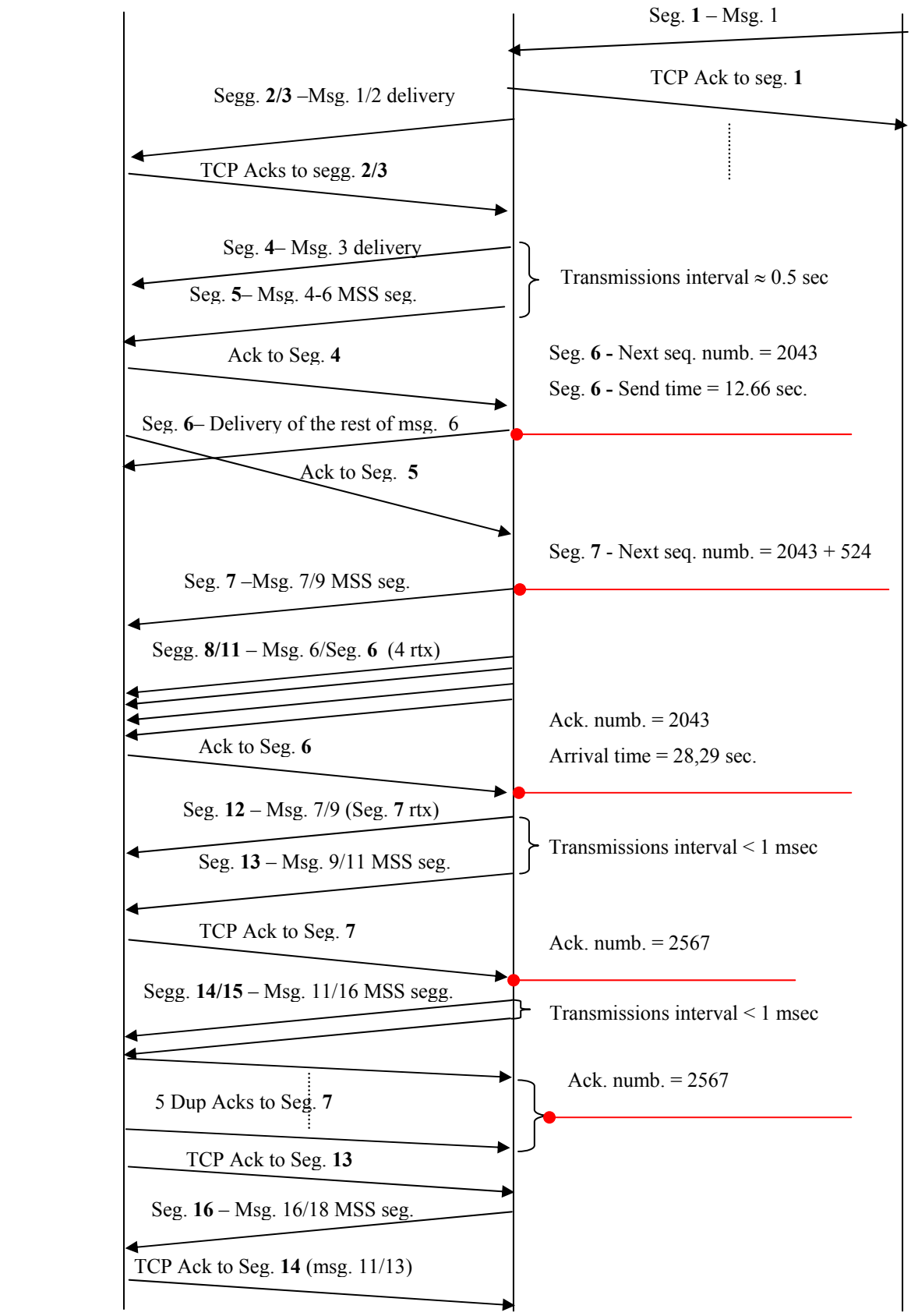
singularly in two TCP segments (2 and 3), receiving the correspondent acks from the mobile client in segments 4 and 6. In Figure 21 only the exchange for the first Jabber message is shown.

- The server sends a third Jabber message in a single TCP segment, number 4.
- However, before receiving the ack for message number 3, the server sends in segment 5 an MSS TCP segment containing Jabber messages 4,5 and part of 6. The time interval between transmission of segment 4 and segment 5 is about half a second. Segment 4 was triggered by the ack to segment 3.
- The ack for segment 4 arrives.
- In segment 6, sent after 12.66 seconds, the rest of message 6 is delivered. This is the segment that gets delayed. The relative TCP sequence number of this segment is 1874 (1874 bytes sent in that direction from the beginning of the test), data length is 169 bytes, and thus next sequence number is 2043.
- The ack to segment 5, the MSS sized one arrives at the server.
- Segment 7 is an MSS TCP segment containing messages 7, 8 and part of 9. Its next sequence number is thus 2043 + 524 = 2567.
- Segments 8 to 11 are server retransmissions, due to timeout expiration (length of timeout exponentially increased) of segment 6, the delayed one. The TCP sequence number of each retransmission is 1874. In the middle of retransmissions, not shown in Figure 21, the server sends presence information of the newly connected mobile client to the fixed client, which acks the received message.
- After 28,29 seconds, the ack to the delayed segment 6 is received by the server. Its acknowledgement number is indeed 2043.
- Segment 12 is a retransmission of segment 7: it is sent as soon as the ack for the previous unacked segment is received. Its sequence number is 2043.
- Segment 13 is an MSS TCP segment delivering messages part of 9, 10 and part of 11. The interval between transmissions of segments 12 and 13 is < 1 msec.
- The ack to segment 7 arrives at the server. Acknowledgement number is 2567.
- Segments 14 and 15 are MSS TCP segments containing messages 11 (partly) to 16 (partly). The interval between transmissions of segments 14 and 15 is < 1 msec.
- Five duplicates acks to segment 12 arrive. Note that they all are cumulative acks: their acknowledgement number is in fact 2567. They account for four retransmissions of segment 6 and one of segment 7.
- In segment 16 a MSS segment containing Jabber messages part of 16, 17 and part of 18 is sent.
- The ack to segment 14 arrives.

- Segment 17 is an MSS one containing messages part of 18,19, part of 20.
- The ack for segment 15 arrives.
- Segment 18 is an MSS one containing messages part of 20,21, part of 22.
- The ack for segment 16 arrives.
- Segment 19 is an MSS one containing messages part of 22,23, part of 24. Segment 20 is sent after less than 1 msec, and containing the rest of the messages that the source has sent to the recipient (up to 25), AND the presence information of the source for the recipient.
- The acks for all the unacked segments arrive.

The message exchange sequence was reported analyzing the messages exchanged through the server interface. The flow chart for messages exchanged through the mobile client interface would look different.

The behaviour of the server is different from the case of reconnection without delay. In this case, even before that the first segment is delayed the server delivers an MSS segment to the recipient (segment 8). In the other case, all the messages were sent one by one. This is because the test case emulates a wireless link between destination client and server, with a higher RTT between client and server.


Mobile (recipient) client
Server
Fixed (source) client
Seg. 1 – Msg. 1
TCP Ack to seg. 1
Segg. 2/3 –Msg. 1/2 delivery
TCP Acks to segg. 2/3
Seg. 4– Msg. 3 delivery
Seg. 5– Msg. 4-6 MSS seg.
Transmissions interval ≈ 0.5 sec
Ack to Seg. 4
Seg. 6 - Next seq. numb. = 2043
Seg. 6 - Send time = 12.66 sec.
Seg. 6– Delivery of the rest of msg. 6
Ack to Seg. 5
Seg. 7 - Next seq. numb. = 2043 + 524
Seg. 7 –Msg. 7/9 MSS seg.
Segg. 8/11 – Msg. 6/Seg. 6 (4 rtx)
Ack. numb. = 2043
Arrival time = 28,29 sec.
Ack to Seg. 6
Seg. 12 – Msg. 7/9 (Seg. 7 rtx)
Seg. 13 – Msg. 9/11 MSS seg.
Transmissions interval < 1 msec
TCP Ack to Seg. 7
Ack. numb. = 2567
Segg. 14/15 – Msg. 11/16 MSS segg.
Transmissions interval < 1 msec
Ack. numb. = 2567
5 Dup Acks to Seg. 7
TCP Ack to Seg. 13
Seg. 16 – Msg. 16/18 MSS seg.
TCP Ack to Seg. 14 (msg. 11/13)

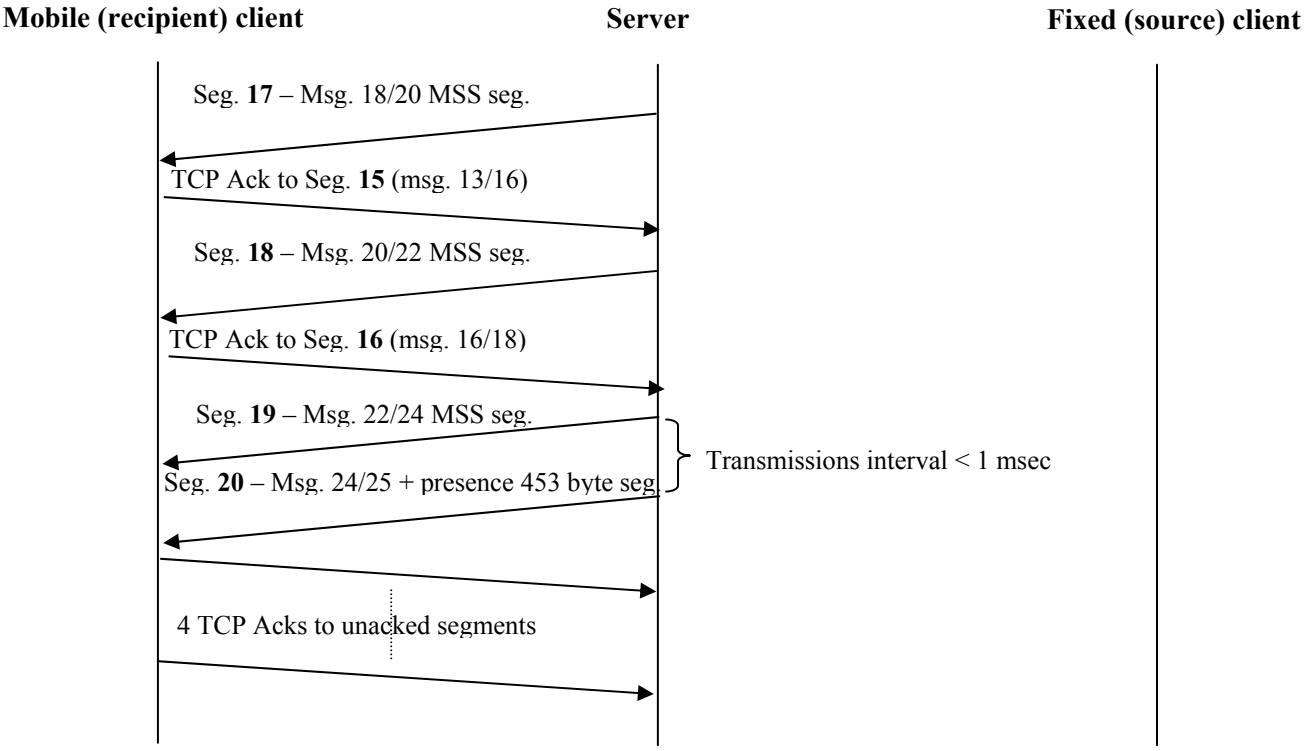



**Figure 21: Test case 4 message exchange flow chart – server interface**

# 19 Test case 5: Delay in reconnection. Uplink direction

This test case is similar to the number 4, but the messages are sent in the uplink direction; that is, the client that receives messages sent to it when it was not connected is the fixed one. This case presents two sub cases, for both of them the emulated scenario is depicted in Figure 18 above:

1. The recipient client (fixed) connects AFTER that all the messages have been delivered to the server from the source (mobile) client.
2. The recipient client connects DURING the time the messages are being delivered to the server from the source.

## 19.1 Connection of the fixed client after message delivery to the server

TEST DESCRIPTION: Messages are sent upstream from the mobile client to the fixed one. The recipient client is disconnected at the moment of message sending. Upon connection, the server begins to deliver messages to it. One of the messages sent by the source (when the recipient is still disconnected) is delayed and does not reach the server “immediately”. The user keeps on typing messages for the intended recipient, ignoring that delay is being experienced. The recipient client will connect after that ALL the messages will reach the server from the source. This test case is actually a mix of two other test cases: "delay_up" (Section 16 for the part of the mobile client sending messages to the server and "reconnect" (Section 17, for the part when the server delivers them to the newly connected client. The traffic is analyzed at the mobile client interface. The beginning of the flow chart shows how messages actually interleave in their arrival and departures at the server and at the mobile client.

- The first message arrives at the server and is acked (segment 1 and ack).
- The second message sent after about 0.5 seconds from test beginning by the mobile client is delayed and arrives at the server after about 15 seconds.
- Segments 3/5 are retransmissions of segment 2 (message 2) that the mobile client sends because the ack for the delayed message is not received.
- The ack for segment 2 is immediately sent by the server when it receives segment 2 (after about 15.5 seconds from test beginning). The ack arrives at the mobile client after about 16 seconds, which is roughly 0.5 seconds + 15 seconds delay + RTT.
- In segments 6,7, in MSS sized segments, Jabber messages 3/15 are packed, coherently to the behavior of the test delay_up. The interval between transmissions is < 1 msec.
- The three DUP acks for the three retransmissions of Seg. 2 are received by the mobile client.
- The acks for segments 6 and 7 arrive. Between such arrivals, two MSS TCP segments containing Jabber messages 15/27 are sent by the source client.
- Segment 10, containing 333 bytes of data, until Jabber message 30, is sent.
- All the pending acks, until segment 10, message 30, are received.

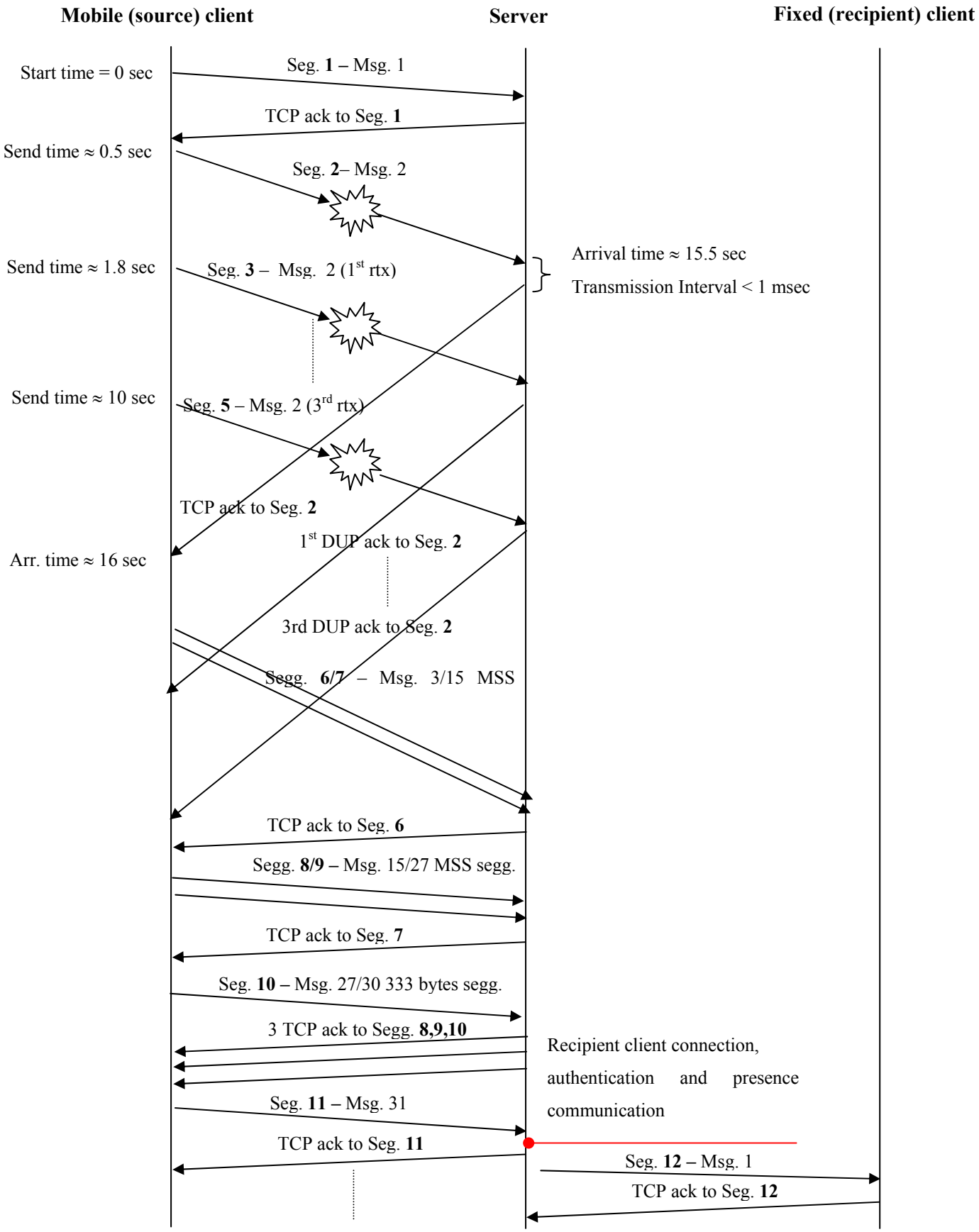



**Figure 22: Test case 5.1 message exchange flow chart – mobile client interface view**

- Beginning from message 31, the client sends the messages one by one, that is, one Jabber message in one TCP segment. This is due to the fact that after the segment containing message 30 has been sent, the 31st has not been yet delivered from the application to the TCP level for sending, because the user has not typed it yet. The timing of messages sending depend on the typing speed of the user.
- After that all the messages have been delivered from the source to the server, the recipient connects. After the recipient authentication phase and presence information communication to the server, the server delivers each message in one TCP segment, which is regularly acked by the client. This behavior is predictable, even though undesirable, as the second part of the test recalls the reconnect test.

This test puts together two previously executed tests, delay_up and reconnect. The conclusions that we can draw out of this test are the same we drew for such tests. This test case gives confirmations about the server internal working mechanisms, and about the way Jabber messages are mapped onto TCP segments.

## 19.2 Reconnection of the fixed client during message delivery to the server

TEST DESCRIPTION: Messages are sent upstream from the mobile client to the fixed client. The recipient fixed client is disconnected at the moment of message sending. Upon connection, the server begins to deliver messages to the recipient. One of the messages sent by the mobile client (when the fixed one is still disconnected) gets delayed and does not reach the server “immediately”. The user keeps on typing messages for the intended recipient, ignoring that delay is being experienced. The recipient client will connect DURING the time that the message is delayed, that is before it arrives at the server and while the other user is typing messages from the user interface. This test case is a modification to the test case "delay_up" (Section 16, where the recipient client connects during message sending from the source client to the server, instead of being already connected when the source begins to send messages.

- The traffic is logged at the server interfaces beginning with the first message received from the mobile client.
- The first three Jabber messages are sent by the source and acked by the server. Each message (beside of course the first) is sent after the reception of the ack for the previous one. The third ack is received by the server 1 sec after test beginning. This is less than 3

RTTs, but there is to consider that the test begins when the server receives the first message, so 1 propagation delay + 1 transmission delay time after the sending of the very first message.

- The first delayed Jabber message is number 4, in segment 4. For the server this lack of messages arrival from the source client looks like a normal period of inactivity (even though from the lack of received inactivity probes it could find out that something is wrong).
- During the time that message 4 is delayed, the recipient client connects to the server.
- After authentication and presence communication of the recipient client, the server delivers the three message it has stored in three TCP segments, and receives the correspondent acks (segg 5/7). The server has not still received message 4.
- The server sends presence information of each other to the two clients. Segment 8 is the presence information of the recipient to the source. The vice versa is not shown.
- The source client acks immediately the received presence message, but the ack is delayed as well. The server thus retransmits 4 times the presence message (segments 9/12).
- Message 4 at last arrives at the server, after 31.44 sec and is immediately acked.
- Immediately after message 4, three retransmissions of it sent by the source client arrive and are dup-acked by the server.
- Afterwards, the ack for the presence message, sent in segment 8, arrives.
- Two dup acks for segment 8 arrive from the source client, triggered by segment 8 retransmissions.
- A fourth retransmission of message 4 arrives at the server (segment 16), and the fourth DUP ack is sent.
- Two more DUP acks for segment 8 arrive. There were in fact 4 retransmissions of segment 8. Note that the time elapsed between the arrival of the delayed message 4 and the fourth retransmission of segment 8 is less than 5 msec.
- Analyzing traffic at the source client interface, it can be seen that the presence message arrives after three retransmissions of message 4. These retransmissions arrive at the server themselves after its retransmissions of the presence message (because they have been delayed). The client acks each retransmission, but the duplicate acks are not delivered to the server but still delayed. The DUP acks for message 4 retransmission arrive consecutively at the source client.
- Message 4 is at last delivered to the recipient client in segment 17 and acked.

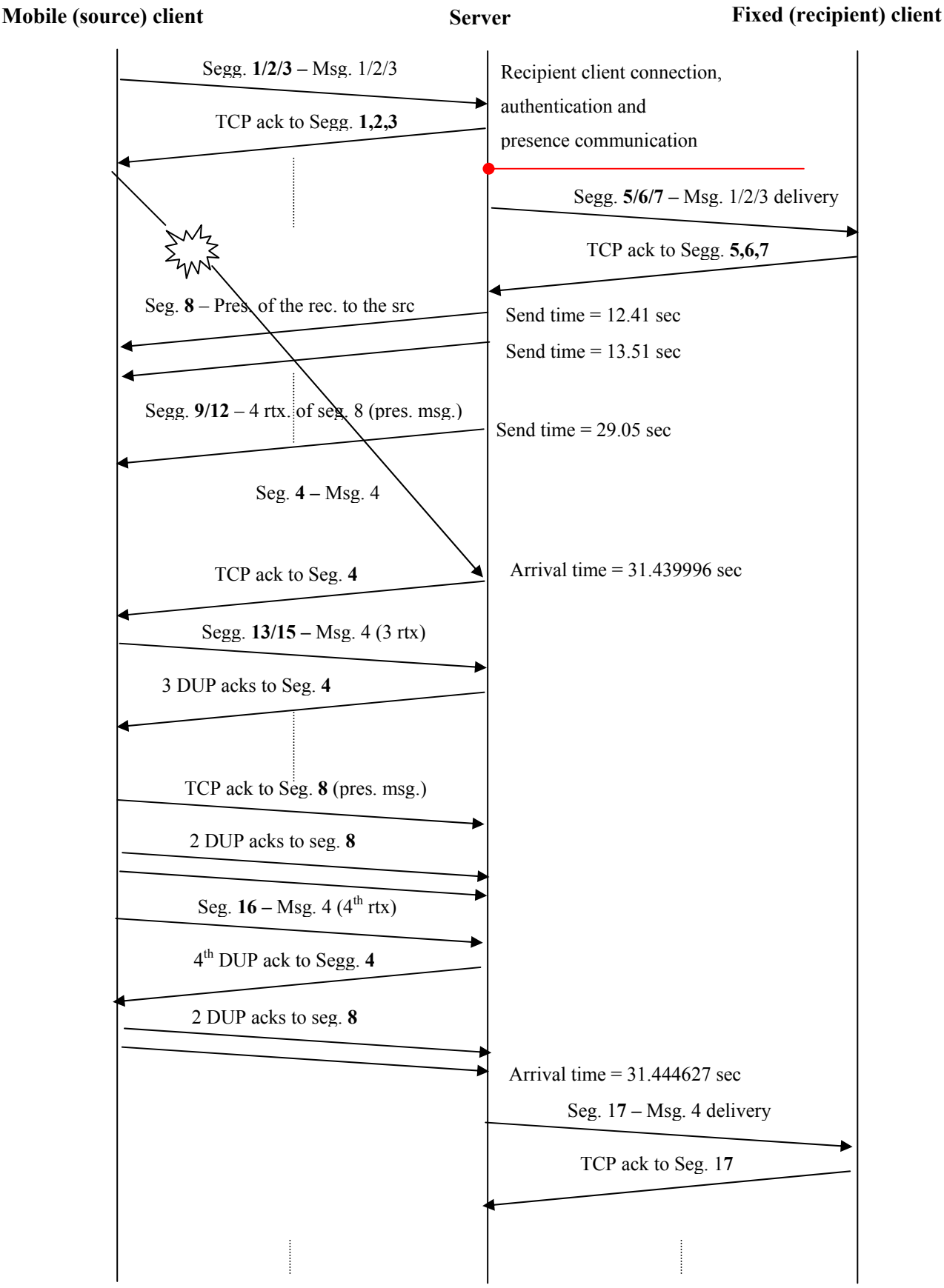



**Figure 23: Test case 5.2 message exchange flow chart – server interface view**

- From now on the source client sends Jabber messages packed in fully sized TCP segments, according to the results of the test case delay_up. The messages are delivered by the server to the recipient client according to the behavior described for the reconnect test. Note that messages 1/3, received by the server when the recipient was still disconnected, are delivered with the additional timestamp string. Message 4 (and the following obviously) are received by the server after that the intended recipient has communicated its presence, so they are delivered without the additional timestamp string.

The scenario discussed emulates a situation of temporary network failure, where all messages sent by the mobile client cannot get through and are delayed (or lost, what it matters is that retransmissions are triggered). A situation that could be interesting to emulate is the one where only one message is delayed and further messages sent by the mobile client can reach the server while the delayed one is still on the network. In such an environment, the outcome of the test described in this section would likely be that the ack to the presence message sent by the mobile client reaches the server, avoiding thus the retransmissions of the presence message and relative DUP acks.

If also one (the first, namely) retransmission of the delayed message can get through, the expected result is that the server is able to send the ack before, and thus the mobile client can send before the other messages that the application has passed to the TCP level. Besides, this situation would only lead to one DUP ack for the delayed message.

# 20 Other test cases

This section discusses other tests that were run to verify some of the issues raised by the main; preliminary tests run before the main ones to gather a general picture are discussed as well We will not go into the details of examining message exchange charts, but we will discuss them in general. Two typologies of preliminary tests cases were run using both the discontinued stable version of the server and the 2.0 version used in the main tests.

## 20.1 Tests with the older server version

Two basic kinds of tests were run with the older server, briefly mentioned in Section 15, involving the delay of messages delivered downlink to the mobile client, and the reconnection of a client that has

messages to be delivered to. We recall that this version of the server, Jabber-1.4.2 (the last release of this server, before its development is discontinued in favour of the Jabberd2 version, is 1.4.3) has nothing related with the Jabberd2 server, used in our main tests. This version was developed inside the Jabber community to be the official Jabber server, before that the Jabber protocol was treated by IETF and modified to be also compliant to the requirements posed by [RFC2779], and called XMPP. As a result, the server is not compliant to many of the operations and features of the XMPP protocol, and this is the main reason for having chosen for our tests the Jabberd2 version. Jabberd2 was developed independently from the first server version in the Jabber community with the precise goal to be compliant to the XMPP protocol specification.

The version 1 server could not handle efficiently sessions with clients connected through wireless links, or in general situation of delay in the network. If several messages accumulate at the server waiting for delivery that the ack for a previous sent segment arrive, the server "loses track" of the arrival order of the messages delivering them to the recipient in the exact inverse order. The recipient client is of course unaware of the exact ordering of messages, also because there is not application level sequence numbering in Jabber, and displays the messages to the user as they arrive, with consequent unpleasant results. However, one good point of this old server is that it always sent fully sized TCP segments, minimizing the number of RTTs needed to deliver data. Note that in an even older version of the server, the messages were reordered in a random way.

However, this server behaves optimally when it has to deliver messages to a newly connected client, as it fills a TCP MSS in order to send the stored messages preserving the order in which they had arrived. A first message is always sent in a single TCP segment, but all the remaining messages are delivered filling fully sized segments. The old version of server does not use an external database to store messages or user account information.

Another test was run with this server, aiming to study its behaviour when large files were sent in a single instant message; this scenario could happen if a user copies and pastes a large text file to send it to the remote buddy. The server can handle optimally this situation, as it delivers the message in fully sized TCP segments, in order, so that the user can display it regularly. We have also sent the big file when the recipient user was disconnected, and the server was able to deliver it correctly and efficiently.

## 20.2 Preliminary tests with the Jabberd2 server

Before executing tests with network failures, we run several test cases aiming to understand a general behaviour of the server in an environment without errors. These tests regarded especially the phases preliminary to the actual exchange of instant messages. We analyzed the various steps needed in the creation of a user account, seeing also how the server reacted in case of errors (such as provision of wrong credentials). We also tracked the steps needed to communicate presence information from the user to the server or for communicating to another user the presence of a buddy. Especially for the phase of server sending other users presence information there seems to be much XML redundancy, but we did not go into further details as these are all steps needed once in a session, and thus improving their realization would lead to a non considerable overall increase of performance, at least compared to improvements in the messages exchange phase.

We also studied the steps needed to add a user to the buddy list (roster in XMPP terminology); the protocol steps are in this case quite cumbersome, as to perform a complete exchange that would eventually lead to two users registered in each other roster, several messages are needed, according to the protocol specification. If user A asks to register user B in his roster, and B accepts, then the vice versa is not automatically true; another (heavy) similar exchange is needed for B to ask A to join his roster. If A could specify with the message that asks subscription to B that it allows to be registered in B's roster this would save several messages exchange. The protocol does not seem to allow this possibility, as it would be desirable as it refers to a common situation between instant messaging users, which often are friends and chat together and thus are in each other roster. Moreover, also the signalling is inefficient and could be improved, by cutting some of the messages needed for the purpose (see Section 8 of [imdraft] for details). However, roster manipulation operations happen once in a while, so their optimization is not a stringent issue, even though for users connecting say, from a GPRS mobile phone, reducing the number of messages needed for performing roster management would also lead to save money.

After analyzing these preliminary phases, we passed to the actual phase of messages exchange. We tried the best-case scenario where communication between clients occurred with no packet losses or delays, seeing that messages passed to the TCP level had the PUSH flag set and were immediately sent out, as it is desirable in an instant messaging application. These results were the basis of the following tests with delays during the message exchange phase.
A situation where a big text file was pasted into the instant messaging user interface was tested, and the Jabberd2 server behaved correctly (exactly as the old server did) filling fully sized TCP segments until the whole message was delivered to the intended recipient.

## 20.3 Tests with two source clients

A common use case in an instant messaging system is that of a user having at the same time more private one-to-one conversations with other remote users. In such a situation, it is likely that at the server arrive several messages from different sources directed to the same recipient. The ideal behaviour of the server would be to deliver messages the intended recipient regardless of the source user; in other words, it would be advisable that if there are two messages ready to send to the same recipient, but with different source, the server packs them in a single TCP segment. This would lead to a reduction in the delivery time, but especially to save bandwidth resources, as fewer messages are sent on the network; header overheads would be minimized due to the increased message payload.

The Jabberd2 server acts indeed correctly, following to the above described conduct. We repeated test cases 1, 2 and 3 having two source clients (the second source client, with reference to Figure 15, was Mind-1) and one recipient. In a first test we sent messages downstream from two fixed clients to the mobile client, alternating sets of ten consecutive messages from each source client. One of the messages got delayed, thus the server had to wait the arrival of the ack before delivering more messages. When the ack arrived, the server begun to send TCP MSS segments, respecting the chronological order in which the messages had been received from the source clients. After the tenth message from one source client, the server correctly appended messages from the other source until the segment MSS was filled.

However, the server presented the same behaviour described in Section 15, and stopped packing messages to send them one by one suddenly, with apparently no reason, precisely, after 19 messages from one client and 13 complete ones (plus part of $14^{th}$) from the other one were sent, for a total of 7 TCP MSS segments. After these 7 segments, the rest of message 14 was sent in a single segment, which therefore had even smaller size than one where a single whole message is sent. Other repetitions of the test case led to different number of messages when the server stopped to send fully sized segments but similar overall results. The common factor is that the server stops packing messages if at some points there are no pending acks. Increasing the RTT of the connection changes the behaviour of the server, which is able to pack more packets because the last unacknowledged segment is acked later in time.
Another test case emulated the situation where two user connected to a wireless link had an instant messaging session open with a user in a fixed network. These mobile clients, attached to the same base station, were the sources of the messages, and the recipient was obviously the fixed client. We

supposed that at certain point in time link connection went down and thus any one of the messages sent by the mobile clients could get through. When the clients gained connectivity, they began to deliver MSS segments to the server, which delivered them to the recipient one by one, respecting the chronological order of arrival. From the point of view of the server, one or two clients in this test case did not change much, it took the messages as soon as they arrived and sent to the recipient.

Finally, it was emulated a situation of two source clients sending messages to a disconnected user; predictably, the server just sent one by one the received Jabber messages upon client connection, respecting the order in which they had arrived.

# 21 Discussion on Jabber experiments

This section gathers the results that came out analyzing the various tests run, discusses them and gives pointers for future work.

## 21.1 The pacing of message sending

In several tests cases the server seems to wait for a TCP ack before sending further messages destined to the same recipient. The reason of this behaviour is that the Nagle algorithm is enabled. However, the situation is not plain, as the time needed to fetch messages from the database and the RTT of the connection with the client seem to influence the pace of message sending.

An inconsistency of server behaviour was observed in the test case 1 (Section 15, where at the server the time interval between an ack reception and another message sending was variable, after that all the messages in MSS segments had been delivered. This interval ranged from values of less than 1 msec, which possibly means that the ack triggered the transmission of the new message, up to value of hundreds of msec, which can be due to the time needed by the server to fetch a message from the database. In this case, the RTT of the connection was relatively low, around half a second; in test runs with higher RTT (around 2 seconds), the server received several messages from the source client, while still waiting for an ack (it has to wait 2 seconds, in the best case). When enough messages were received, the server packed them in TCP MSS segments and sent them. In the case that no enough messages to fill a TCP MSS were gathered, then the server waited for the ack arrival before sending new messages in a not fully-sized TCP segment.

The Nagle algorithm can be an obstacle for an instant messaging server; the messages exchanged are never so small (around 200 bytes) to make TCP overhead so much relevant and waiting for pending ack can lead to a loss of interactivity. A simple example: the server sent a message and is waiting for a pending ack and does not send any message until one of these three conditions is met:

1. The ack is received.
2. Enough messages to fill the TCP MSS are gathered.
3. The TCP timeout of the sent message expires, and the server retransmits the message.

The reasons for which the ack could be pending are that either the ack itself or the triggering message got delayed. While the server is waiting for the ack, it could send messages to the recipient. Because of network reordering, there is the possibility, in case the original message was delayed, that following messages arrive at the recipient before the delayed one. In this case the TCP of the recipient will not deliver them until the delayed message finally arrives, to achieve in-order delivery. Disabling the Nagle algorithm increases the probability that the receiver TCP has more data ready to deliver to the application when one segment arrives delayed.

## 21.2 Sending full-sized TCP segments

Another point of possible improvements is making sure that the server packs as many messages as possible (ideally, it should fill a TCP MSS) into a single TCP segment if it has enough Jabber messages ready to deliver to a client, instead of mapping one message onto one segment. This improvement would lead to deliver all the intended messages with minimal number of RTTs.

In several test cases we have observed inconsistent server behaviours; for example, in test case 1 (delay in downlink direction), the server first sent messages utilizing the full TCP MSS, and after all the pending segment were acknowledged it stopped doing this and delivered one Jabber message in one TCP segment. In test case 2, delay in the uplink direction, the server always delivered messages one by one to the recipient client. The difference between these two test cases is where the messages got delayed: when the messages were directed downstream, it was at the server, while for upstream messages the delay was suffered by the mobile client. In test case 1, the fixed client sent the messages one by one to the server, which forwarded them to the recipient, while in the other test case, the server received from the mobile client messages packed in MSS TCP segments, in a more bursty way, before

delivering them one by one to the recipient. In test case 3, the server sent the previously stored messages one by one to the recipient.

Our results show that the behaviour of server is strictly related to the time needed to retrieve messages stored in the database and to the RTT of the connection with the client. Same test cases, run with increased RTT values, show that the server is able to utilize the full TCP MSS when delivering Jabber messages to the application.

## 21.3 Summary and proposed improvements

Summarizing what evidenced in the two above sections, two are the most relevant modifications to how messages are mapped onto the TCP level suggested for achieving immediate performance, and both aim to reduce the time needed to transfer instant messages to a client. These modifications are particularly effective in scenarios where frequent delays occur, or more in general, in scenarios where the connectivity is bad, which can also mean disconnected networks, where the user loses and gain again connectivity, a situation likely to occur in cellular networks.

Particularly, if the latency is an issue, for example because a slow wireless link is on the end-to-end path, we suggest to send available messages as soon as possible even though an ack for a previous one is pending. This means disabling Nagle algorithm, which is the cause of the above mentioned behaviour. However, disabling Nagle is not the solution to all the problems, as Nagle is also strictly related to the second improvement proposal, that is, utilizing as most as possible the full size of the TCP MSS.

In fact, if Nagle were disabled, on the network we could observe a flow of small sized segments, all containing one Jabber message instead of TCP MSS segments (or at least of bigger size than those containing only one message). An example clarifies what stated: the server sends a message and while the ack is pending, the server receives more messages from the source client. With Nagle enabled, it would wait until one of the conditions 1 or 2 (in the example of the above section) would be satisfied. If condition 1 is verified and the ack arrives, the server would deliver as many messages as it has received from the client during the time spent for waiting the ack; if condition 2 is verified, then the server sends a TCP MSS segment. Note that if the timeout expires (condition three) the server will just retransmit the lost segment.

Thus Nagle implies, as well known, a trade-off between better network utilization and timeliness of data delivery: is it better to have single messages delivered immediately or waiting and deliver a burst of messages? This trade-off can be subject of further research.

A useful hybrid approach between having Nagle enabled and having it disabled, can be to keep track of the pace with which a user sends messages. If the pace is relatively fast, then it could be convenient for the server to wait until either enough messages to fill TCP MSS are gathered or a send timeout expires. The server must accurately select the send timeout, because a too long value would lead to a loss of the interactive nature of instant messaging, while a too low value would have the same effect of no timeout at all. The duration can be selected according to the average rate of message sending of the user, the average RTT measured for the connection between the server and the end-user and must of course respect the rate limiting policies if the server administrator has enforced them.

With regard to the improvement of the internal protocol working mechanism, a major point of improvement was detected in the subscription phase between two users. In the present document, we did not go into the details of the message exchange needed in this situation; nevertheless several messages are needed to complete the exchange and add a Jabber user to the roster of another one. The number of messages needed can be reduced by merging information carried by different messages in order to decrease the time and the network utilization needed to perform the exchange.

Operations of registering in each other roster do not occur so frequently, and this is the reason why we concentrated our analysis in improving the actual phase of message exchange, which takes most of the time in a Jabber session. However, optimizing the operation of registration in each other roster, even though may not give substantial improvements to the single end user, is a desirable feature for a network operator, as it leads to considering saving of bandwidth resources, especially as the number of Jabber users in the network increases, and thus proportionally the number of subscriptions processes (that is, signalling messages on the fly) initiated in the network.

# PART III: GUIDELINES FOR EFFICIENT IM SERVERS IMPLEMENTATION

# 22 Scope of the guidelines

Part III of the document puts together the considerations drawn previously and the experiments results to provide a set of guidelines for implementing an efficient Instant Messaging and Presence server, to be deployed in a wireless environment. The guidelines we give are valid in any network scenario, but are particularly useful for client – server communication in wireless networks. Also, even though in our experiments we have used the Jabberd 2 [jabberd] server, the considerations can be extended to any server providing IM or store and forward type of service, as are they are valid regardless of the protocol used.
We can summarize the guidelines as follows:

- Amount of data delivered to the TCP layer
- Pacing of segments sent to the network
- Handling multiple source clients
- Handling big messages
- Efficiency of database access
- Caching messages

The guidelines we provide are not and cannot be at any way exhaustive for an actual implementation of an IM server, both as number and as level of detail. They must be intended as traces that implementers should follow to build an efficient IM server. Some of the guidelines give suggestions for making the server more efficient, others instead enforce strong requirements that must not be neglected, as server performance would otherwise drop drastically.

We also present, in Section 24, a timeout based algorithm that servers can use to tune the pace of message delivery to an intended client, in order to maximize the efficiency and the timeliness of message delivery. Section 23.2 provides the motivations behind the proposal of such an algorithm.

# 23 Guidelines

This section presents in detail the guidelines outlined above. Each of the bulleted items in Section 22 is extended and the reasons for giving such a guideline are explained after writing it.

## 23.1 Amount of data delivered to the TCP layer

**Guideline 1***: An IM server MUST delivery stored messages to the TCP layer in large enough data blocks. Each data block written to TCP layer SHOULD have size of at least one TCP MSS, if possible. The server MUST minimize the number of RTTs needed to deliver messages to a client.*

Our experiments have shown that in several occasions the server, even though it had already received several messages from a source client, did not send as many Jabber messages as it could fit in a full sized TCP segment. In test case 3, reconnection of the recipient client, messages were destined to a disconnected client, being thus stored in the server for later delivery. When the client connected, the server delivered one Jabber message in one TCP segment, with obvious inefficiency in terms of RTTs needed for completing the delivery.

However, not always delivering all the messages at once is the desired solution, especially from the client side. A user may not want to be overwhelmed by all the messages stored while he was offline, because he is accessing from a slow wireless link, for example, in a way very similar to what IMAP allows, as illustrated in this document. A Jabber Enhancement Proposal (JEP) is being discussed in the Jabber community to allow a "Flexible Offline Message Retrieval" [JEP0013]. With this enhancement, a user reconnecting to a Jabber server can request to the server to deliver ONLY the header of the messages stored while he was offline, and specify, based on the received headers, which message body he wants to have delivered.

The key idea is that the server sends as much packed information as possible. Whether it is the messages stored, or just the headers, whether the client has just gone on line or messages were accumulated because of network delays, the server must deliver this information with MSS sized TCP segments, until all the data is delivered.

## 23.2 Pacing of TCP segments sent to the network

**Guideline 2:** *IM servers SHOULD have Nagle algorithm disabled. Doing this requires that the pace of message sending SHOULD be regulated by an application level timeout, whose length must be tuned so that the number of small segments on the network is limited to at most one per RTT.*

In normal operating situations, when the server receives a message for a given user, it immediately delivers it to the intended recipient, according to the IM philosophy. When the Nagle algorithm is enabled at the server, messages arrived at the server before a TCP ack for a previously sent message is received, cannot be delivered to destination, unless a full-sized TCP segment is sent.

On one hand, enabling Nagle is a way of conforming to Guideline 1, as it allows servers to gather many messages and send them at once. On the other hand, if the RTT between the destination client and the server is high and few messages are sent from a source client, but not enough to fill a TCP MSS, having Nagle enabled could be detrimental for the interactivity of the IM system, as the TCP layer at the server side will wait for the ack to arrive or the timeout to expire. It would be better if it had sent the received messages when still TCP congestion window allowed it, before timeout expiration, in contrast to Nagle. This approach would speed up message delivery both if the ack is delayed itself, and if it was the triggering segment, containing the Jabber message, which gets delayed.

Based on the above considerations, we argue that a middle way solution between enabling Nagle or disabling is desirable for an instant messaging server, which allows to achieve the efficiency of multiple message delivery in a single TCP segment, without suffering from timeliness problems. We give a high level description on how to tune and use timeouts for pacing message delivery later in this document, in Section 24.

## 23.3 Handling multiple source clients

**Guideline 3:** *Forwarding of messages to a given recipient client MUST be independent from the source client. The server MUST pack in a single TCP segment ready-to-delivery messages addressed from multiple source clients to a single recipient.*

Scenarios where a given IM user is exchanging instant messages with more than one peer are common in real life. In this case, the communication happens between multiple source clients and a single recipient client. An IM server should not base its forwarding policies on the source clients, but rather

on the destination client. According to the Guideline 1, the server should always deliver to TCP as much data as possible addressed to a given client, regardless of the source client. For example, if there are two messages ready to send to the same recipient, but with different source, the server should deliver them to TCP at once, so that they can be sent in a single segment. This would lead to a reduction in the delivery time, but especially to save bandwidth resources, as fewer messages are sent on the network; header overheads would be minimized due to the increased message payload.

## 23.4 Handling big messages

**Guideline 4:** *Servers SHOULD forward to TCP layer instant messages bigger than the TCP MSS as soon as they receive the first MSS bytes of data instead of waiting that the whole message is stored before beginning to forward it.*

Sending big instant messages is not so common, as the common behaviour for most of the user of IM services is to send quick and small messages; moreover, all the IM platforms offer to the users the possibility to transfer files, usually out-of-band. The Jabber protocol instead performs in-band file transfer. File transfer, since is an extension to the baseline functionalities of an IM server, is out of scope for this document.

Situations where users send big instant messages are possible, and servers must be prepared to handle them efficiently; for example, a user could copy and paste into the client user interface a big text file and send it as instant message. For big messages we mean those that do not fit into a single TCP segment as their size is higher than the TCP MSS for that connection (practically every IM protocol uses TCP for carrying messages). Servers should avoid a store and forward policy and begin to forward a big file as soon as the first TCP segment is received. The ways how servers can detect the end of file are protocol dependent; for example, in Jabber, the `<message>` and `</message>` tag indicate respectively the beginning and the end of the application level message. Other protocols may have some header field indicating the length of the message.

## 23.5 Efficiency of database access

**Guideline 5:** *IM servers MUST access the database where they keep stored message in an efficient way. They MUST minimize the number of accesses needed for executing their normal operations.*

**Guideline 6:** *IM servers MUST NOT fetch from the database one message per time but at least MSS data to give TCP the possibility to send full sized segments.*

The Jabberd 2 server used in our tests uses a database to store messages and information about registered users. In our experiments, database access has proven to be a bottleneck and influences server performance, especially under heavy load conditions. The reconnection case, executed when the RTT between server and recipient client was relatively low (under 10 msec) shows that the server was not able to pack more Jabber messages in one TCP segment, as it would be advisable according to Guideline 1. Repetitions of the reconnection test, with increased RTT values between client and server, reveal that the longer is the RTT the more server behaviour is optimal, and it is able to send full sized TCP segments. This indicates that if the server has more time for fetching and gathering together messages from the database, then it succeeds in respecting Guideline 1.

Repetitions of the delay down test with increased client – server RTT confirm this phenomenon. The server keeps on sending TCP MSS segments until all the data have been delivered instead of beginning to send them one by one when all the pending acknowledgements have been received. When the RTT is higher, it takes longer for the server TCP layer to receive all the pending TCP acknowledgements and the server has more time to fetch and gather together messages from the database.

The reconnection and delay down tests were executed having the Jabberd 2 server running in a more powerful machine than the one used throughout the main tests. In this case the server was always able to send Jabber messages packed in TCP MSS segments, even in situations of low RTT where the same server running with the older machine was not able to pack messages. We increased the load on the server (in the more powerful machine), running it in debug mode, which forced the server to write into a log file long lines of text whenever a significant event happened. We run tests recreating the same network conditions where the non-debug mode server packed messages and the outcome was that the server in debug mode was not able to do it. This result can be generalized moving the considerations about hardware to considerations about the overall load on the server, meant as number of concurrent users. The higher is the number of concurrent users, the higher is the possibility that the server will not be able to pack messages.

An optimal server implementation must guarantee an efficient database access, even under heavy load conditions. By efficient access we mean minimizing the number of interactions with the database, trying to fetch at least TCP MSS data at once; limitations on the amount of data to fetch and deliver to the TCP level can be given by TCP send buffers size for example. Other optimization ways are

possible, but we do not focus on them. The load on servers can be diminished by splitting server components over different machines, or duplicating server components so that they can share the load (both of these alternatives are possible in Jabberd 2).

## 23.6 Caching messages

**Guideline 7:** *IM servers SHOULD use a cache memory for storing messages of active destination clients, in order to retrieve them faster than it would be from the database. Messages addressed to offline users MAY be kept only in the database until their reconnection.*

Another way for minimizing the server accesses to the bottleneck database is of temporarily caching messages besides storing them into the database (duplication of messages is needed for failures recovery), when the recipient user is on line but the message cannot be delivered (i.e. connection problems experienced on the client – server connection). This is the case of the delay down test, where the source client kept on sending to a connected recipient client messages, which the server TCP layer could not deliver because it was waiting for a TCP acknowledgement. In such a situation, instead of storing the messages only into the database, it would be faster if the server could write them into a temporary cache memory as well. Accessing to the cache memory is faster than to a database, and this improves servers performance.

The size of the portion of cache memory reserved for each recipient client should be tuned according to the number of users on line; when a user goes off line, his portion of cache may be freed if needed. In general, when a user disconnects, his part of cache memory should be freed, however, if few users are online, servers could still maintain cache entries for the most recent disconnected users. If a new user comes on line, than a part of cache memory is reserved for his messages. When messages are fetched from the cache they are deleted and possibly replaced if the server has more messages for that client in the database. If a message for a user arrives, and does not fit into the cache, then it will be written into the database. When a user comes online, some of the messages stored in the database can be moved to its cache share while others are being delivered to be more quickly accessed and delivered.

How the cache memory should be partitioned among users is an implementation choice; we propose two alternatives. The first uses always all the available cache memory but is more complex; the server divides the whole cache memory exactly among all the connected users, but must redistribute the shares of memory in case of join/leave events. In this way, the amount of data that can be stored for

each user is higher, guaranteeing a faster data manipulation, but server operations are more complex because of repartitioning operations. Moreover, if the cache memory for a given user is full of messages and a new user comes on line, the server should remove the amount of data subtracted to that user's share copying it to the database (there is no need for copying if all the messages are originally duplicated to cache and database).

In the second method, the server does not divide all the memory among the online users, but leaves a part as reserve and when other users coming on line no repartition is needed as the necessary memory share is taken from the reserved area. Repartitioning operations are still needed if the number of online users grows and the reserved area runs over. The server could keep a history of the online users per period of day/week/month to have an idea of the average of online users in a given moment and on the amount of memory to reserve.

Another option is for the server to not reduce the share of memory for a user if the cache portion is occupied by messages; server should reduce the share only when memory is freed. The alternatives we have proposed are not the only solution, several are possible; discussing them in detail is out of our scope, which is to build on the above considerations for giving the following:

# 24 Timeout based sending algorithm for IM servers

The rationale behind the proposal of a timeout algorithm for instant messaging servers is that having the Nagle algorithm enabled causes servers TCP not to transmit small segments into the network but increases the risk that they remain stuck at the server side waiting for an ack to arrive. On the other hand, not enabling Nagle increases the number of small TCP segments on the network and the overhead of TCP/IP protocols header. The problem is that the application does not have the control on the sending policy of TCP; a solution where such a control can be obtained is desirable. Nagle algorithm should be disabled, but the application must pass to the TCP layer data only when either MSS data are available or when enough time has passed so that further postponing the delivery would cause loss of interactivity.

For an IM server, this consideration translates in gathering messages (possibly in the cache) addressed to a given recipient client, from whatever source client, until there is enough data (MSS bytes) to send or a timeout expires. This solution preserves the efficiency of data delivery implying less small sized TCP segments in the network and guarantees the interactivity of IM applications as messages would

never wait too much at the server before being delivered. The duration of the timeout must be carefully chosen, as too little value would lead to several small segments in the network, and too high value to an excessive delay in data delivery. In any case, the application level timeout must not exceed the TCP retransmission timeout.

The server should tune the duration of the timeout based on the RTT of the connection between the server and that given client. Of course, the server must have available at application layer information on the RTT of the connection with the client. An ideal solution would be for the server to use the RTT values already computed at TCP layer. When transport layer information on the RTT is not available, RTT must be computed at application layer.

If the IM protocol foresees application layer acknowledgments, this is easily done; however, one must cope also with protocols, such as Jabber, which do not have application layer acks. One solution is to measure the RTT of the connection in the phase of session establishment. This phase is constituted, as described in Section 12.3 by the exchange of several application level messages between client and server. One message sent triggers a reply from the other side; servers could measure the RTT based on the time elapsed between sending a message and receiving the answer from the client.

This solution is not optimal, as the server would not have information on the RTT for the remaining (and much longer) part of the session, where the actual instant messages exchange is performed. Servers could thus send PING messages to the client, at a reasonable rate in order not to cause congestion in the network, to measure RTT. This approach has the disadvantage that a client behind a firewall could never receive PING messages because of security policies. A third method is of using ECHO messages, to be answered with ECHO_REPLY messages, and measure the time elapsed between sending the ECHO and RECEIVING the reply. This solution implies modifying the IM protocol, if it does not have any functionality like this. Some protocol, like Jabber, have keep-alive probes that clients, idle for a long time, send to the server to signal that they are still connected. The protocol should be modified to allow server to send these probes to the client and having the clients answer with an application layer ack and not only a TCP ack.

## 24.1 Computing the timeout

The timeout value for a client A is computed every time a TCP ack for a delivered instant message is received by the server TCP layer, if it is possible to reuse transport layer information at application layer. If RTT is computed using application level information, than the RTT calculation must be

effected differently, as described above. For example, servers may send periodically test messages to a connected client (PING, ECHO, keep-alive) to measure a fresh value of RTT. In any case, the RTT to be used for calculating the IMTimeout must take care of the last measured value (SampleRTT) and of the average of the last measured values (Estimated RTT). The formula we propose is the same used at TCP level (α=0.125):

$$EstimatedRTT = (1-\alpha) \cdot EstimatedRTT + \alpha \cdot SampleRTT \quad (1)$$

$$IMTimeout = \gamma \cdot EstimatedRTT \quad (2)$$

γ is a parameter higher than 1 (a possible value is 1.5, but we do not enforce any value) which must tune IMTimeout for satisfying this inequality:

$$EstimatedRTT < IMTimeout$$

In other words, the IMTimeout should be higher than the EstimatedRTT, which means γ > 1. Further improvements to the algorithm may lead to refinement of this formula, for example, γ could be set dynamically according to the network conditions or other factors instead of being a fixed parameter. If such an algorithm is used, a server waits that a message for a destination client is received. It does not deliver the message to TCP immediately, but starts the timeout and waits. If more messages arrive for that given client and the server is able to deliver down to TCP MSS data, then the IMTimeout is stopped, data delivered. If the IMTimeout expires before MSS have been gathered, data are delivered anyway to TCP and timeout stopped.

After data delivery to TCP, if a new message arrives, the IMTimeout is initiated again; the RTT value to be used is the last computed one, according to one of the computation methods we have suggested above.

## 24.2 The algorithm

```
1 /* Initialization: client A connection */
2 Compute RTT with A based on the session establishment
phase;
3
4 /* IMTimeout calculation */
```

```
5 while (;;) {
6 if (client disconnects)
7 Execute client_disconnect();
8  if (the message for A arrives at the server is the first in queue){
9     Store message;
10   Compute IMTimeout according to the formula, use last computed RTT value;
11 Start the IMTimeout; }
12 while (IMTimeout has not expired && total size of messages in A queue < TCP MSS) do
13 {
14   if (client A disconnects)
15         Execute client_disconnect();
16   if (another message for A arrives at the server) {
17         Store it into database;
18         Compute the size of pending messages; }
19 }
20 /* Delivery operations */
21   if (IMTimeout has expired) {
22         Deliver data to TCP layer;
23         Reset IMTimeout for A; }
24   else {
25         while (is it possible to deliver MSS blocks of data to TCP) Deliver MSS data to TCP layer;
26   Recompute IMTimeout with the last calculated RTT value and restart it; }
27 }
28
29 Client_disconnect() {
30 Delete all the IMTimeout related state for A;
31 Deliver A's presence update for A's buddies (if any) to TCP;
32 If any, leave the undelivered messages for A in the database for offline delivery;
33 Exit; }
```

# 25 References


[berkeley] Berkeley DB database. http://www.sleepycat.com/

[CP02] Chakravorty, R.; Pratt, I., "WWW performance over GPRS". Mobile and Wireless Communications Network, 2002. 4th International Workshop on, 9-11 Sept. 2002 Page(s): 527 -531

[core] Saint-Andre P., Miller J., "XMPP Core". Internet Draft (Work in Progress). November 2003. draft-ietf-xmpp-core-20

[Gray95] Gray T., *Message Access Paradigms and Protocols*. Available from: http://www.imap.org/papers/imap.vs.pop.html

[ICQ01] *Overview of ICQ.* http://mobile.act.cmis.csiro.au/wu-tang/srs/icq.html.

[imdraft] Saint-Andre P., Miller J., "XMPP Instant Messaging". Internet Draft (Work in Progress). November 2003. draft-ietf-xmpp-im-19

[imap] Crispin M., *Internet Message Access Protocol – Version 4rev1*, March 2003. RFC 3501 (Standards Track).

[impp] *Instant Messaging and Presence Protocol (impp)* – IETF Working Group http://www.ietf.org/html.charters/impp-charter.html.

[Isak01] H. Isaksoon, *Version 5 of the ICQ Protocol,* ICQ protocol specification document April 2001. Available from: http://www.algonet.se/~henisak/icq/icqv5.html - CTC.

[jabber] *The jabber official web site*: http://www.jabber.org/

[jabberd] The jabberd 2 project. http://jabberd.jabberstudio.org/

[Jac88] Jacobson V., "Congestion Avoidance and Control", Proceedings of ACM SIGCOMM '88. Stanford, California, August 1988, pp. 314-329.

[javaserv] Java Jabber server: http://javajabber.net/en/

[JEP0013] Saint-Andre P., Kaes C., "Flexible Offline Message Retrieval". JEP-0013 (Standard Track). 22-01-2004

[JEP0096] Muldowney T., Miller M., Eatmon R., "File Transfer". JEP-0096 (Standards Track). 30-12-2003

[Min03] Mintz Mike. *MSN Messaging Protocol description*. Unofficial, second draft. Available from: http://mono.es.gnome.org/imsharp/tutoriales/msn/book1.html (last checked: 3.6.2003).

[mySQL] MySQL database. http://www.mysql.com/

[Oett01] R. Oettinger, *Total Time Spent Using Instant Messaging Jumps 110 Percent At Work And 48 Percent At Home Versus Last Year, Reports Jupiter Media Metrix*. Available from: http://www.jupiterresearch.com/xp/jmm/press/2001/pr_111401.html.

[OpenN] *OpenNap: Open Source Napster Server,* 04.06.2003. Available from: **http://opennap.sourceforge.net/**napster.txt

[Oscar] Jeremy Hamman, *AIM/Oscar Protocol.* Available from*:* http://www.geocities.com/smokeyjoe12345/OscarProtocol.htm

[pop] Myers J., Rose M., *Post Office Protocol - Version 3*, May 1996. RFC 1939 (Standard).

[Proto] *Protocol-gaim*: http://gaim.sourceforge.net/protocol.php

[RFC1459] J. Oikarinen, D. Reed, *Internet Relay Chat Protocol*, May 1993. RFC 1459.

[RFC1733] Crispin M., *Distribute Electronic Mail Models in IMAP4*, December 1994. RFC 1733 (Informational).

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

[RFC2779] Day M., Aggarwal S., Mohr G., Vincent J., *Instant Messaging / Presence Protocol Requirements,* February 2000. RFC 2779 (Informational).

[RFC2810] C. Kalt, *Internet Relay Chat: Architecture*, April 2000. RFC 2810.

[RFC2811] C. Kalt, *Internet Relay Chat: Channel Management,* April 2000. RFC 2811.

[RFC2812] C. Kalt, *Internet Relay Chat: Client Protocol*, April 2000. RFC 2812.

[RFC2813] C. Kalt, *Internet Relay Chat: Server Protocol*, April 2000. RFC 2813.

[RFC2822] Resnick P., ed., *Internet Message Format,* April 2001. RFC 2822 (Standards Track).

[SA01] Saint-Andre P., *Let there be Jabber*. Linux Magazine, August 2001. Available from: http://www.linux-mag.com/2001-08/jabber_01.html

[SASL] Myers J., "Simple Authentication and Security Layer (SASL)". RFC 2222 (Standards Track). October 1997.

[SILC1] Riikonen, P., *Secure Internet Live Conferencing (SILC), Protocol Specification*, Internet Draft, May 2002.

[SILC2] Riikonen, P., *SILC Packet Protocol*, Internet Draft, May 2002.

[SILC3] Riikonen, P., *SILC Key Exchange and Authentication Protocols*, Internet Draft, May 2002.

[SILC4] Riikonen, P., *SILC Commands*, Internet Draft, May 2002.

[Sin03] Liew Kwek Sing Nelson. *AOL ICQ vs MSN Messenger*. CM316 Multimedia Systems Coursework, University of Southampton, Dept. of Electronic and Computer Science. Available from: http://mms.ecs.soton.ac.uk/papers/16.pdf (last checked: 3.6.2003).

[STAT] *IRC network statistics*, http://irc.netsplit.de/networks/, 04.06.2003.

[TLS] Dierks, T., Allen C., "The TLS Protocol Version 1.0", RFC 2246 (Standards Track), January 1999.

[TOC98] *Version of TOC protocol* (Version1.0). Also available from: http://www.cs.berkeley.edu/~mikechen/im/protocols/aim/PROTOCOL.txt.

[XMPP] *Extensible Messaging and Presence Protocol (XMPP)* - IETF Working Group. Available from: http://www.ietf.org/html.charters/xmpp-charter.html

[yahoo] *The messenger official website*: http://messenger.yahoo.com.

[yahoop] *The Yahoo protocol*: http://www.howtodothings.com/showarticle.asp?article=491

[zep] A document about zephyr protocol: http://web.mit.edu/zephyr/doc/protocol