University of West Bohemia in Pilsen
Faculty of Applied Sciences
Department of Computer Science and Engineering

Diploma thesis

Instant Messaging in corporate networks

Pilsen, 2004 Petr Ferschmann

I hereby declare that this diploma thesis is completely my own work and that I used only the cited sources.

Pilsen, 21 May 2004, Petr Ferschmann, ...

Content

1 Introduction

There exists a lot of instant messaging systems. The most often used ones are free public systems. They are great for personal usage but for corporate usage they may not be the best choice. I will try to write down the requirements for such usage. I will also try to implement such system.

I will also try to figure out which protocol is better for corporate usage. Public services protocols are usually proprietary and not published. But in last year standardization organisation IETF (Internet Engineering Task Force) published two public protocols – SIP/SIMPLE and XMPP/Jabber. I will look on them.

One of the requirements is also scalability. I will try to find scalability issues in my proposed solution and suggest ways how to bypass them. To increase scalability it is required to configure target machine; I will do that for Linux and I will describe required steps. And finally I will make some measurements to prove my decesions.

TODO: dopsat úvod – HOTOVO?

2 Enterprise Instant Messaging

¨The usage of Instant Messaging (IM) system in corporate networks has specific requirements. In this chapter we will discuss the main requirements.

TODO – rozepsat požadavky – HOTOVO?

Today most IM systems are used as public Internet services. These systems are also used as corporate networks. But these systems are out of control of the company – when you forget password or your account is stolen you can not do anything. When you receive message from someone, you can not be sure that she is really she and not some attacker.

So, now, I will try list all requirements:

- **interoperability** – today IM systems cannot communicate together. When you use one system, you can not send message to other system. It is due closed protocols and closed policy – public IM providers usually fight against interoperability gateways – by protocol changes and even by blacklisting (i.e. centrally disabling) such participant. That is also the reason why everybody uses public IM systems instead of private ones (such as for E-mail). It is unacceptable for corporate customer to use IM that can not "talk" to customers, so everybody uses the public systems as customers do that too.
- **security** – corporate customers need higher security:
  - confidentiality – for sending confidential message – i.e. nobody can read it during the transmission except the receiver. For such cases we use end to end encryption (public/private key) and usuallly also transport encryption for even higher security (we sometime send unencrypted data).
  - authenticity – when we receive a message we should be able to verify that it comes from the right person. Again this can be done by asymetric encryption (public/private key). But because today the Public Key Infrustructure (PKI) is not widely adopted today we can not use asymetric encryption as we would like. So for us it is sometime acceptable to be just sure that the message sent someone from cooperating company – in such case verifying domain is enough – this is also required for fighting against spam.
- **directory services integration** – corporate users use some kind of directory service (LDAP,

Active Directory, etc). Such service usually provides a list of user accounts, groups, etc. This services are used in companies to simplify account administration across whole company network. Every computer service in company uses such system for user authentication and authorization.

- **authentication service integration** – this service is an integral part of a directory service. But sometime it can be separated. Example of this is usage is Kerberos with LDAP. Access to such services is usually handled by SASL libraries.
- **central administration** – we need server side configuration that is not available at all in public services. We need to add/remove user, specify access control policy, configuring running services, etc.
- **access control** – we should be able to limit communication of some users, so they can communicate only with our customers and/or our company employers..
- **central roster (contact list) –** it will be good, if we can insert new users into a roster – so we can enforce that everybody in team will have all other members in contact list. We can also do that every secretary will have support team member in contact list.
  Integration of searching with directory services is also required.
- **server side roster** – we need to store roster on server, so roaming users can connect from different computers and still hey will be able to see all users.
- **server side history** – for some customers it is required by law to store all communication history (i.e. lawyers, stock market corporations etc). It is also good for roaming users, so they can always look on messages they sent (as they can do that with e-mail – in case of IMAP).
- **fault tolerance and high availability** – because IM systems are used for business, it is viable for company that such service is still available. So the IM system should have some features that permits hardware failure without service outage – for higher availability IM systems should be deployed on clusters of machines.
- **multi node operation** – many companies have geographically divided departments. IM systems are often used for communication inside the company. When Internet connection fails, local IM system must be able to work. It is very similar to phones – the company cannot work without that. The cooperation inside company must continue even the

internet connection is broken. of course the communication to the world (other departmens or customers) will not be possible.

- **simple installation for a small office** – this is a usability requirement.

This are only the main requirements. But there are many other features that can be usable in corporate network:

- **alias support** – many companies have internal support - IT support, serviceman, etc. And employers usually communicate with them. They can use either phone or newly IM. But this service are provided by multiple people. So we can provide service that will help to choose who should handle the request. Every member of IT support team can activate the IT support role. When it activated the alias account will be visible online (eg. itsupport@example.com). When user send message to that address, the employer with activated role for IT support will receive such message. Of course who can activate roles should be limited by some kind of policy rules.
- **GSM redirect** – When you receive message that fits some group of rules (eg. high priority, sender user – eg. managing director, etc) and recipient user is just off line (not present at computer) we can deliver such message to GSM cell phone as SMS (Short Message Service).
- **Event messages** – in company's IS (information system) can happen many kind of events – from new order to support request. Sometime users want to be informed about such event. We can on such event send IM message and inform user immediately.

3 Protocol comparison

3.1 Closed protocols

These protocols are not public and cannot be used for new implementations. So I will describe them only shortly.

3.1.1 ICQ

This protocol was the first one. It created Instant Messaging market. For a long time, this protocol was the most frequently used, but today it has fewer users than Jabber (see [ZDNET01]). The ICQ was bought by AOL company – the company that provides AIM system. So now the ICQ program that can be downloaded on http://www.icq.com/ uses AIM protocol.

The User ID (UID) is a number (integer). When the user registers, he gets a UID.
ICQ protocol is based on UDP. ICQ server works as a directory that maps users to their IP addresses. Contact lists were stored only on client side. In the cases with firewall, users can send short messages through the server. Messages sent through the server are limited to 460 bytes.

TODO: dopsat ICQ

TODO: rozepsat prechod na AIM protocol.

3.1.2 AIM

America OnLine

TODO: dopsat AOL

3.1.3 MSN Messenger

Protocol from Microsoft.

TODO: dopsat MSN

3.1.4 Yahoo!

asd

TODO: dopsat YAHOO!

3.2 Open protocols

The only publicly accepted organization that do standardized IM protocols is IETF.

We can divide IETF IM specifications into following groups:

RFC 2778 and RFC 2779 – specifies requirements for IM protocols.

Common specifications by IETF – common specifications for all protocols (mainly profiles)

XMPP/Jabber – specifies XMPP (Extensible Messaging and Presence Protocol). It is successor of Jabber protocol.

SIP/SIMPLE – SIP/SIMPLE protocol. It is an IM protocol based on SIP (Session Initiation Protocol) used for VoIP (Voice over IP)

3.2.1 Common specifications

As part of IETF standardization effort was formed IMPP (Instant Messaging and Presence Protocol) Working Group and created common requirements for all instant messages protocols. Today both strongest players (SIP/SIMPLE, XMPP/Jabber) fulfill it.

It created 4 specifications (2 RFC and 2 Drafts):

- RFC 2778: A Model for Presence and Instant Messaging [RFC 2778]
- RFC 2779: Instant Messaging / Presence Protocol Requirements [RFC 2779].
- Common Profile for Instant Messaging [IETF-IMPP-IM]
- Common Profile for Presence [IETF-IMPP-PRES]
- Presence Information Data Format

3.2.1.1 RFC 2778 - A Model. for Presence and Instant Messaging

This specification defines roles for presence service and instant message service.

Presence service

This service has two types of „clients“: **presentities** (can set presence status) and **watchers** (watches presence status). A watcher can be either a subscriber (i.e. notified about future changes) or a fetcher (i.e. requests for actual presence status).

Instant message service

This service also has two types of „clients“: a **sender** and an **instant inbox**. A sender sends messages and instant message service delivers message to the instant inbox.

It also defines the required parts of presence information and instant message. For more info see [RFC 2778].

3.2.1.1.1 RFC 2778 – guaranteed delivery

This specification says that the IM system is not required to guarantee delivery. But in the case of enterprise usage, this is highly inappropriate. We MUST guarantee delivery even in the case of lower scalability or throughput.

Jabber fulfills this guarantee in the case of server to server delivery. In case of client to server delivery it is not guaranteed (see section „Proposed jabber improvements“).

SIP/SIMPLE guarantees delivery in the case of page-mode messaging. In case of Message Session, the session cannot be initiated when failure occurs and therefore messages cannot be sent at all (so the user knows about it).

3.2.1.2 RFC 2779 – Instant Messaging / Presence Protocol Requirements

This paper specifies common requirements and security considerations for IM protocols. It is a set of rules.

This paper is important only for protocol specification writers and not for protocol implementers.

3.2.1.3 Common Profile for Instant Messaging: message format

For complete text see [IETF-IMPP-IM]

This draft specifies common profile for IM message systems interoperability. It defines the interoperability message format.

It specifies the format for interoperability (based on MIME). The format specification is not useful. What is useful is the field description.

As both XMPP and SIMPLE contain these fields, this specification can be used as a base for protocol interoperability.

3.2.1.4 Presence Information Data Format

http://www.ietf.org/internet-drafts/draft-ietf-impp-cpim-pidf-08.txt [IMPP-CPIM-PIDF]

This draft specifies CPIM presence format. It is based on XML.

As both XMPP/Jabber and SIP/SIMPLE contain this fields, this specification can be used as a base for protocol interoperability.

3.2.1.5 Address Resolution for Instant Messaging and Presence

http://www.ietf.org/internet-drafts/draft-ietf-impp-srv-04.txt [IETF-IMPP-SRV]

This draft specifies how to map the URL to real services. It incorporates _im and _pres DNS SRV prefixes. It is used mainly on interoperability gateways and proxies.

3.2.1.6 Common Profile for Presence

http://www.ietf.org/internet-drafts/draft-ietf-impp-pres-04.txt [IETF-IMPP-PRES]

This paper specifies the common profile to allow interoperability between various presence services. It is intended for writers of gateways and proxies.

3.2.1.7 Common Profile for Instant Messaging (CPIM)

http://www.ietf.org/internet-drafts/draft-ietf-impp-im-04.txt [IETF-IMPP-IM]
This draft belongs to common specifications.

TODO – zkontrolovat zda ty odkazy jsou spravně

TODO – otázka zda mít tyto specky jako podkapitoly nebo jako výpis (tečkama) nebo rozepsat

3.2.2 Jabber/XMPP

3.2.2.1 Overview

Jabber is an open, XML-based protocol for near real-time (ie does not guarantee delivery times) messaging and presence.

The Jabber project was started by Jeremie Miller in early 1998. Behind this protocol stands Jabber Software Foundation (JSF). JSF created the protocol specification. The IETF working group was formed in order to standardize the protocol as IETF standard. This group is called XMPP WG and it standardized the XMPP/Jabber protocol.

XMPP/Jabber fulfills RFC 2779 (common IM specification [RFC 2779]) and also contains mappings to CPIM (Common Profile for Instant Messaging).

It is XML over TCP/IP protocol. It works very similarly to SMTP but it is more secure and more spam-proof. It uses URL format:

```
jabber://user@domain/resource
```

user – user identification part

domain – domain identification – it corresponds to the existing domain. The mapping from domain to server handling jabber service is done by DNS extension *SRV*. If it is not defined, it connects directly to the host represented by domain (A entry).
resource – allows one client to be connected multiple times (eg. mobile/ work, etc).

Jabber specifies two parts - client to server (c2s) and server to server (s2s) communication. The client always connects to its server and all communication to the other one is done through this server.

3.2.2.2 Jabber protocol specifications

3.2.2.2.1 Extensible Messaging and Presence Protocol (XMPP): Core

This draft describes the core of XMPP. It describes XML in TCP protocol. This part is not necessarily bound to IM system.
It specifies this parts:

- **Stream protocol** – stream level protocol.
- **Stream initiation** – how to initiate stream (version negotiation, choosing target, etc)
- **Feature negotiation** – this is very important for future extensibility – the client can learn about the supported features.
- **TLS binding** – it is very similar to STMP `startls` command. You can negotiate the features, select the virtual server and start the encrypted channel.
- **SASL authentication and SASL encryption** – SASL is used for authentication. SASL can also be used for the encrypting channel. It is allowed to use TLS and SASL encryption.
- **Resource binding** – this is the only IM specific part – it is used to negotiate the resource part in JID (Jabber ID). It is used after SASL authentication.

So an example connection can look like this:

```
C: <stream:stream
C:       xmlns='jabber:client'
C:       xmlns:stream='http://etherx.jabber.org/streams'
C:       to='example.com'
C:       version='1.0'>
-- important are fields 'to' and 'version'. If version is not present, value
-- '0.9' is used.
S: <stream:stream
S:       xmlns='jabber:client'
```

```
S:       xmlns:stream='http://etherx.jabber.org/streams'
S:       id='345'
S:       from='example.com'
S:       version='1.0'>
S:   <stream:features>
-- we support TLS
S:     <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'>
S:       <required/>
S:     </starttls>
-- we support SASL
S:     <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
S:       <mechanism>DIGEST-MD5</mechanism>
S:       <mechanism>KERBEROS_V4</mechanism>
S:     </mechanisms>
S:   </stream:features>
C:   <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
S: <proceed xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
-- TLS is started here – all future data are encrypted --
C: <stream:stream
C:       xmlns='jabber:client'
C:       xmlns:stream='http://etherx.jabber.org/streams'
C:       to='example.com'
C:       version='1.0'>
S: <stream:stream
S:       xmlns='jabber:client'
S:       xmlns:stream='http://etherx.jabber.org/streams'
S:       id='345'
S:       from='example.com'
S:       version='1.0'>
S:   <stream:features>
S:     <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
S:       <mechanism>DIGEST-MD5</mechanism>
S:       <mechanism>KERBEROS_V4</mechanism>
S:     </mechanisms>
S:   </stream:features>
-- start SASL authentication
C: <challenge xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
C: cmVhbG09InNvbWVyZWFsbSIsbm9uY2U9Ik9BNk1HOXRFUUdtMmhoIixxb3A9ImF1dGgi
C: LGNoYXJzZXQ9dXRmLTgsYWxnb3JpdGhtPW1kNS1zZXNzCg==
C: </challenge>
S: <response xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
S: dXNlcm5hbWU9InNvbWVub2RlIixyZWFsbT0ic29tZXJlYWxtIixub25jZT0i
S: T0E2TUc5dEVRR20yaGgiLGNub25jZT0iT0E2TUhYaDZWcVRyUmsiLG5jPTAw
S: MDAwMDAxLHFvcD1hdXRoLGRpZ2VzdC11cmk9InhtcHAvZXhhbXBsZS5jb20i
S: LHJlc3BvbnNlPWQzODhkYWQ5MGQ0YmJkNzYwYTE1MjMyMWYyMTQzYWY3LGNo
S: YXJzZXQ9dXRmLTgK
S: </response>
-- SASL response-challenge continuous
S: <success xmlns='urn:ietf:params:xml:ns:xmpp-sasl'/>
-- SASL was successful – we start again stream
C: <stream:stream
C:     xmlns='jabber:client'
C:     xmlns:stream='http://etherx.jabber.org/streams'
C:     to='example.com'
C:     version='1.0'>
S: <stream:stream
S:     xmlns='jabber:client'
S:     xmlns:stream='http://etherx.jabber.org/streams'
S:     id='345'
```

```
S:      from='example.com'
S:      version='1.0'>
S: <stream:features>
S:     <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>
S:     <session xmlns='urn:ietf:params:xml:ns:xmpp-session'/>
S: </stream:features>
-- client negotiates
C: <iq type='set' id='bind_1'>
C:      <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
C:         <resource>work</resource>
C:      </bind>
C: </iq>
S: <iq type='result' id='bind_2'>
S:      <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
S:        <jid>somenode@example.com/work</jid>
S:      </bind>
S: </iq>
C:</stream:stream>
S:</stream:stream>
```

TODO – výpis do příloh

When the client initiates encryption wrapping or SASL authentication, it sends again <stream:stream/>. Attributes 'version', 'id' and 'to' must be forgotten and sent again. These values should correspond to previous (i.e. when authenticated to domain example.com it should still be example.com).

Moreover, this draft describes how to handle stream level errors.

It also specifies stanzas from '**jabber:client**' name space.

- **<message/>** - it is a kind of push method used for sending messages.
- **<iq/>** - it is a pull method for asking questions.
- **<presence/>** - used for sending and receiving presence.

All the stanzas have these attributes:

- from – identifies the sender that sent this stanza
- to – identifies the receiver of this stanza
- id – uniquely identifies the stanza in this session.

3.2.2.2.1.1 <message/>

Used for sending messages.

Example stanza:

```
C:  <message from='juliet@example.com' to='romeo@example.net' xml:lang='en'>
C:     <subject>Hello</subject>
C:     <body>Art thou not Romeo, and a Montague?</body>
C:  </message>
```

3.2.2.2.1.2 <iq/>

IQ (Info/Query) is used for sending and receiving commands to/from server. iq contains attribute 'type'. That can have these values:

`get` – the client asks server for some kind of information (eg. content of roster)

`set` – the client sets some value to server (eg. add user to roster)

`result` – the server responds to the client that the action was successful (the in case of set) or sends data (in case of get).

`error` – the action cannot be finished – an error happened.

3.2.2.2.1.3 <presence/>

Publish/Subscribe framework for presence status.

Example stanza:

```
C:  <presence>
C:     <show/>
C:  </presence>
```

3.2.2.2.2 Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence

http://www.ietf.org/internet-drafts/draft-ietf-xmpp-im-22.txt [IETF-XMPP-IM]

This draft enhances XMPP-CORE for IM systems. There are specified <presence/>, <iq/> and <message/> stanzas. For more information see this draft.

3.2.2.2.2.1 End-to-End Object Encryption

http://www.ietf.org/internet-drafts/draft-ietf-xmpp-e2e-07.txt [IETF-XMPP-E2E]
Here is specified how to encrypt stanzas in XMPP/Jabber. For encrypting it employs S/MIME.
Example of e2e encrypted presence:

```
<presence to='romeo@example.net/orchard'>
     <e2e xmlns='urn:ietf:params:xml:ns:xmpp-e2e'>
   <![CDATA[
   Content-Type: multipart/signed; boundary=next;
                 micalg=sha1;
                 protocol=application/pkcs7-signature

   --next
   Content-type: application/pidf+xml
   Content-ID: <2345678901@example.com>

   <xml version="1.0" encoding="UTF-8"?>
   <presence xmlns="urn:ietf:params:xml:ns:pidf"
             xmlns:im="urn:ietf:params:xml:ns:pidf:im"
             entity="pres:juliet@example.com">
     <tuple id="hr0zny">
       <status>
         <basic>open</basic>
         <im:im>away</im:im>
       </status>
       <note xml:lang="en">retired to the chamber</note>
       <timestamp>2003-12-09T23:53:11.31Z</timestamp>
     </tuple>
   </presence>
   --next
   Content-Type: application/pkcs7-signature
   Content-Disposition: attachment;handling=required;filename=smime.p7s

   [signed body part]

   --next--
   ]]>
     </e2e>
   </presence>
```

3.2.2.3 High and low of protocol

High:

Very simple for implementing and understanding.

Existing open source and commercial implementations with very high quality.

Great number of extensions

Widely used

IETF standard

Low:

scalability – it uses TCP/IP and XML. This can be hard to implement when we are talking about scalability. But you can see later in this document it can be implemented with scalability in mind.

3.2.2.4 Important features for Enterprise IM

3.2.2.4.1 Client reconnection

When the server has multiple IP addresses (eg. group of machines) and one IP address should be removed (shutting down one machine), it must let the client know that he should reconnect to the same server. Jabber has a special stream error for this purpose.

3.2.2.4.2 Message delivery notification

As a base of XMPP/Jabber there is delivery notification, but it is not reliable. It means that if the server got an error it can inform you. But if the message is lost during sending there can be situations where sender is not informed about that. It is mainly when we write the data into TCP/IP stream and the operating system stores a part of the message in buffer for later sending. The application thinks that data were sent, but the receiver does not receive full data. This is especially a problem with mobile users where connection failures are frequent (eg roaming GPRS).

3.2.2.4.3 Server side message history

XMPP/Jabber does not specify how to fetch message history from server. So this protocol must be extended.

3.2.2.4.4 Client connection failure detection

Sometimes the client connection is ended without proper closing. The server can try to send data to the client or will deliver only a part of XML stanza. This is a big problem with roaming users. It can be fixed by using keep-alive – i.e. once in a while it sends some data to connection (eg. space character). If the connection is broken, timeout will occur very soon. This is also widely used.

3.2.2.4.5 Pushed roster

As a side effect of <iq/> protocol we can push contact list entries to the client. The client will show it in the contact list. This can be used for central managing of rosters – changes will be shown to the client in time.

3.2.2.4.6 SASL authentication

Because XMPP/Jabber uses SASL for authentication, it can be easily integrated into the existing Kerberos or NTLM infrastructure.

3.2.2.4.7 Directory integration

To search for users, we can use JUD (Jabber User Directory). It is a service designed for this purpose. This can be easily integrated into the existing directory service. SASL authentication can be also integrated into this service.

3.2.2.4.8 Interoperability

Nowadays there exists many components that can be used for accessing other networks. These can be used as internal or external components accessed by XMPP/Jabber.

TODO: dopsat Jabber

3.2.3 SIP/SIMPLE

TODO: dopsat SIMPLE

4 Enterprise Implementations comparison

4.1 SharePoint Services Collaboration platform

Closed protocol. It is part of MS Exchange Server.

TODO: dopsat

4.2 Lotus SameTime server

http://www.lotus.com/products/lotussametime.nsf/wdocs/homepage

TODO: dopsat

4.3 AIM

TODO: dopsat

4.4 odigo.org

Only MS Windows. Gateways (MSN, Yahoo, ICQ). Can communicate over HTTP.

TODO: dopsat

4.5 jabber.org

TODO: dopsat

4.6 12Planet Instant Messaging Server

web system – není možný jiný přístup
http://www.12planet.com/en/software/messenger/
LDAP, SQL, Gateways(MSN, Yahoo, ICQ/AOL), POP3, SMTP

TODO: dopsat

4.7 Microsoft exchange server

TODO: dopsat

5 Why to choose XMPP/Jabber

TODO – dopsat

Why is jabber better then SIMPLE – see [IMP01].

6 Scalability constraints

Jabber is based on XML over TCP/IP. This is main scalability issue in this protocol. Most of the operating systems are limited to number of TCP/IP connection possible to one machine and/or one program. This limits we can divide in to:

- operating systems limits
- application constraints

6.1 Operating system limits

These limits are mentioned here only for informational purpose. These limits are OS specific and their bypassing is also OS specific. These issues must be bypassed on every machine that will be part of the cluster.

From these limits you can see that practical maximum of TCP/IP connections on one machine is by default 1024 and can be increased somewhere between 200 000 and 1 000 000 (see Appendix A - Operating system limits on page 40).

6.2 Application constraints

In this section we will discuss scalability constraints for our application so that we can fulfill them later.

To decrease these constraints, we can divide users into sub domains – eg. pilsen.example.com, prague.example.com. So any synchronization and replication is only required inside this domain.

Most of the required databases are user-centred – the user must see consistent view. But two users across the system do not need to be completely synched (except opened session registry and presence subscription registry).

TODO – rozepsat víc omáčky

6.2.1 Database types

In this part we will describe databases that we need to fulfill our needs. We also describe the operation mode for them.

6.2.1.1 *Opened session registry*

In the case of a cluster we need to know who is connected where. We need to know that on the machine X is connected user `'romeo@example.com/pda'`. So when we receive a message for such a user, we can deliver it to the opened connection.

We need to quickly replicate this registry. When some part of the system goes down, all sessions that are connected to it are discarded. When the whole system goes down, the registry is empty.

If replication is delayed (ie not synchronized) we can get into trouble – we can accept a session with already used resource. We can also receive the message and because we think that the user is offline, we store it in offline storage for later. But the user already requested offline messages and does not ask second time for them.

In this registry we get an entry for every connected user. This database is usually stored in the memory.

6.2.1.2 *Presence subscription registry*

When we add somebody to the contact list we ask for subscription – if this subscription is accepted, we are informed about further presence changes.

This registry can be delayed in replications. These are rarely changed.

6.2.1.3 *Roster storage*

This service is used for storing the content of a roster. Data are accessed only by the owner. They must be strongly synchronized.

In some implementations, presence subscription registry and roster storage are merged together.

6.2.1.4 *User account directory*

This database is used for user authentication and search within JUD (Jabber User Directory). This database replication can be delayed without problems. Password changes and account addition should be replicated quickly.

6.2.1.5 *Private storage*

This storage is used for storing client configuration and should be synchronized for changes.

It is accessed only by the owning user.

6.2.1.6 Offline message queue

When the system receives a message for the user who is not connected, we need to store it for later delivery. When user connect we send all stored offline messages to him.
We do not need to replicate this queue. We only need to inform the queue that it should be dispatched to the client.

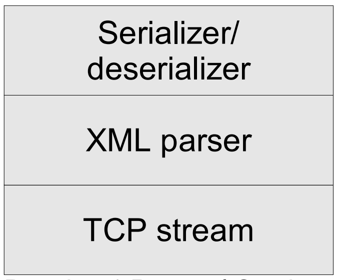
6.2.1.7 Message history storage

Because we need to store messages on the server side, we need storage for that. If history can be seen by the user, we need to replicate it quickly (but we do not need synchronization on that as we only add items). If the user cannot see the message history, it can be delayed.

6.2.2 XML stream handling

Another problem to solve is how to handle XML streams.

We have the following handling stack:



Drawing 1 Protocol Stack

We read/write data from/to TCP stream. We need to parse XML from byte stream. Later we need to create date structures from XML stanzas.

TODO – co je to stanza, serialize/deserializer najdete v další kapitole.

Example stanza:

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

As you can see, XMPP/Jabber uses namespaces intensively. This is also true when writing the stream. The whole stream is one XML document.

Let us look at this stream as series of chunks. We need to have one XML parser per connection that is the same for the connection lifetime.

At the beginning we get

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

This part is different from later stanzas because it is not closed. This is opened for the whole connection lifetime.

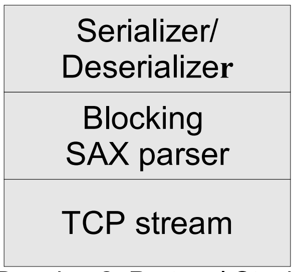
We have basically two ways how to handle XML streams. We can divide them by thread style usage:

- one connection per thread
- multiple connections per thread (reactor pattern, see [ACE-OVERVIEW])

See the next chapters for descriptions of each mode.

6.2.2.1 Connection per thread

This model is the simplest but has scalability problems. It is recommended for systems that support a maximum of 1000 concurrent connections.



Drawing 2 Protocol Stack

This solution is simple – we just create XML parser (`SAX` – event based parser) and provide low level handling for stream (i.e. we must override how to read data). When the parser needs more data, it just gets blocked on the socket and waits for the next data.

We must use such a parser that supports partial reading – we need to handle data as they come and we cannot wait for the rest of XML document.

For this purpose, we cannot use a generic `DOM` parser as it returns the whole document at once (and want to read till the end of the document). However, we can adapt `DOM` to work over `SAX` and return only one stanza at a time. But first we have to handle `<stream:stream>` differently. I recommend using `SAX` for reading, and when we accept stream initiation, we use some kind of `SAX` to `DOM` adapter (why to use `DOM`, read chapter 6.2.2.2.3 on page 26).

The issue with this model is mainly thread stack and operating system scheduling (some OS has problems with too many threads – eg. linux version up to 2.5.x). It is required to reserve virtual and also real memory for every thread. This can easily decrease available virtual memory for process (this occurs only on a 32 bit processor).

6.2.2.2 Multiple connections per thread

In this case we use reactor pattern (see [ACE-OVERVIEW]). This pattern uses a system known as `select` or `poll` (unix syscalls), `java.nio` (J2EE) or `WaitForMultipleObject` (Win32 syscall).

We have only one thread for handling multiple connections (usually all connections) waiting on them, and when there is some event, we handle it. On this thread there should not be too much work as it can decrease throughput (we should delegate demanding operations to other threads (thread pool)). We can think of this pattern as registering callback for receiving or writing data. The disadvantage of this mode is the requirement for non-blocking XML parser (see chapter 6.2.2.2.1 Non-blocking XML parser on page 25)

This mode can scale to millions of concurrent connections (see chapter Appendix B – Measuring implementation on page 43).

6.2.2.2.1 Non-blocking XML parser

We need to implement non-blocking XML parser. One way is to write a completely new parser. But software theory tells us to reuse as much as possible. So we can use some existing XML Pull Parser (see [XMLPP]) implementation.

Pull Parsers are very similar to SAX. The difference is who controls the application flow. In case of SAX our callbacks are called when some part of XML is found (eg. attribute, tag, etc). But in the case of Pull Parser, we call the parser and it returns to us the next event type and we can control the program flow.

TODO – dopsat strukturu non-blocking XML parseru

An example code using Pull Parser can look like this:

```
// walk trhu all attributes
for (int i = 0; i < parser.getAttributeCount(); i++) {
   // compare attribute name
   if (parser.getAttributeName(i).equals("type")) {
      String value = parser.getAttributeValue(i);
      // compare attribute value
      if ("get".equalsIgnoreCase(value)) {
            // do something
      } else {
            // do another thing
      }
   }
}
```

This program looks for an attribute named “type” and with value “get”.

But using only Pull Parser is not enough. We can still have only half of the stream and we cannot block (as we handle multiple connections from this thread). So we can employ `XMLStanzaDetector` (see chapter XML Stanza Detector on page 27). This is an "intelligent" buffer. We add to it all the data we receive. It counts "<" and ">" from the data and when it receives a complete stanza it returns it to us. Again there must be some kind of hack for opening `<stream:stream>`.

Again, we provide a wrapper for low level reading from stream for XML PullParser. But this time we do not read directly from socket but we read from `XMLStanzaDetector`. But, as might be guessed, the stanza detector is a simple parser. So it can easily happen that XML is not valid and XML Pull Parser will try to read more data than we have. In that case, I recommend to throw stream error, as XML is not valid.

6.2.2.2.2 Karma

As a protection against the denial of services (DoS) jabber.org's server implementation uses „karma". Every connection has its own karma. It represents how much data can be sent in a certain amount of time (eg. a minute). When this limit is exceeded, the socket slows down or gets completely blocked.

Using these mechanisms, it is possible to protect oneself against spoofing by receiving data at a very high speed. The same concept is used for limiting the number connections coming from one IP address.

6.2.2.2.3 Event based parsing versus Document Object Model

There are two basic ways to handle XML data. This is not about difference between SAX/XML Pull Parser and DOM but about the way of handling the data.

- **Event based** – We read only the data that we need. At first sight it can look excellent, but the opposite is true. This solution is more error-prone and that can be a problem. There is also the problem that every event can be read only once. In my implementation I used this way (XML Pull Parser), but after some experience with that I can recommend only DOM handling. In Pull Parser I sometimes made a mistake and there was a situation when I forgot to read till the tag end. Thus the parser was in a different state than I expected later.
- **DOM** – this way we read stanza (either using SAX or XML Pull Parser) into memory as

Document Object Model and later we browse through it. This solution allows us to browse DOM twice and when we get lost in parsing (eg exception is thrown) we always know where we are (as the parser does not have position in XML stream like in event based way). This is a recommended way based on my experience. For this case I recommend some kind of XMLPullParser to DOM adapter.

6.2.2.2.4 XML Stanza Detector

In the non-blocking case where we take the parser only for the whole stanza, we need to know when the stanza is complete. We a used XML stanza detector. It is a growing buffer with stored cursor position, used size, max size, xml node level, previous character and the state of the machine.

xml node level – is used for getting to know how deep in XML we are.

previous character – it is used to detect opening and closing tags.

state machine – we need to differentiate where we are. We can have the following states:

NORMAL – we are in normal text

TAG – we are inside the tag

TAG_ATTRIBUTE – we are inside tag attribute (in quotes)

In some cases using only NORMAL and TAG is enough. But if the characters „>“ and „<“ are incorrectly stored in attributes, we do not learn about it (but this is incorrect xml; the detector is allowed to just throw it away with error).

Again, we need to have some kind of hack for supporting opening `<stream:stream>` and `<?xml version=”1.0” ?>`.

In implementation details you can see one example XML stanza detector.

TODO dopInit – implementation details.

6.2.2.2.5 Serializations/Deserializations vs. direct processing.

Now we have parsed XML stanza in some kind of DOM. So let us think what to do now. There are two basic ways to handle the incoming stanzas.

One way is to directly pass the stanza to the processor (handler). This is the most natural way. The processor will find out what kind of stanza it is and what fields it contains.

But there is another, much better way - using serializers/deserializers. The basic idea is that we put stanza (XML node) to the serializer. This is a small program that creates some representing object from XML node. This way we separate the way we understand XML from the way we process it. This is very similar to MVC pattern (Model View Controller) – we separate the way how a packet looks, how it is represented in XML and how we handle it.

Let us take example.

This is the model:

```
class Message implements XMLStanza {
  String subject;
  String body;
}
```

This is how it looks in XML (view):

```
<message>
  <subject>Hello world!</subject>
  <body>Hello from Pilsen</body>
</message>
```

To the view part also belongs the way we translate from XML to model.

And finally the controller – this is the processor:

```
void onMessage(Message msg) {
      System.out.println(msg.getSubject());
}
```

Today there are two versions of Jabber protocol – XMPP/Jabber 1.0 and old Jabber 0.9. And we can just use a different view to communicate these two versions.

6.2.2.2.6 Clustering - gates

It can take an amount of time and bandwidth to establish 1 000 000 connections. So we should take care of closing them as little as possible.

One solution is to have gates/proxy to system. Gates „own" the connections and receive data and send them to the rest of the system. This allows us to upgrade nearly the whole cluster (except gate machines) without the user even noticing anything. If we want to remove one of gate machines we stop it and the clients will reconnect to another one (using <stream:see-other-host/> in XMPP stream).

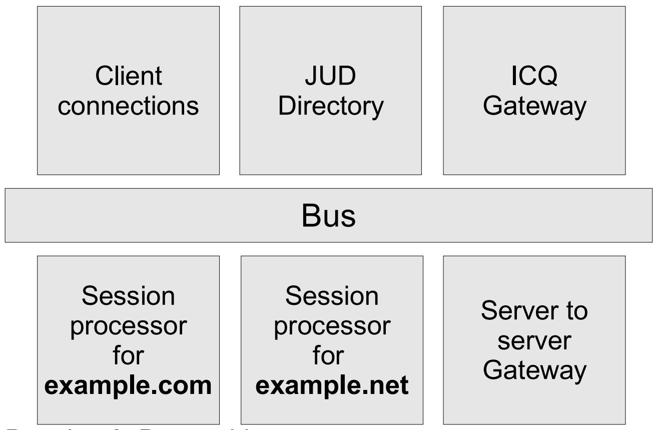
7 Proposed architecture

In this chapter we will discuss ways of implementing the IM system. We will start from a simple solution and end with the clustering solution. We will use constraints from the previous chapter.

7.1 Message Bus Architecture

In our system we will use message bus architecture. As you can see, the system is just plain old MOA (Message Oriented Architecture). We have multiple services that need to receive messages that are addressed to them. So every service subscribes to the message bus with addresses it wants to receive messages to. In our example we have two virtual domains: example.com and example.net. We also have icq.example.com (ICQ gateway) and jud.example.com (Jabber User Directory). We also have server to server gateway – that is the "default route". All messages that cannot be delivered locally are routed to external server using generic jabber protocol.

We also have "Client connections". That is a client gateway. Clients are connected to it and all messages from them are sent to session processor for the attached domain. Every connection at the beginning of communication specifies the domain it wants to talk to (eg. example.com or example.net). There must be negotiation if the given domain is handled on the message bus (as it can be on a different computer in the bus). If the message from the client is targeted to different service, the session processor just redirects the message to a proper receiver. There are reasons for that. We can easily implement interceptor support (eg. for logging messages, verifying by XML Schema, etc) or implement policy support (this user cannot send messages outside the company while others can).


Client connections
JUD Directory
ICQ Gateway
Bus
Session processor for **example.com**
Session processor for **example.net**
Server to server Gateway
Drawing 3 Bus architecture

All the magic about clustering support is hidden in the message bus. We can have multiple session processors for the same domain or multiple server to server gateway. Some intelligence in the message bus can do load balancing, fail over, heartbeat etc. All the replication staff is hidden by the service.

TODO – co to je session processor

7.1.1 Simple Bus implementation

In this case we will not use bus capable of communication across multiple computers. Bus is just message queue, with thread pool taking data from this queue and registry for services. Every service can be called simultaneously by multiple threads with multiple messages at once.

Slightly modified scenario I used in prototype implementation.

TODO – rozepsat vic

7.1.2 Clustering Bus implementation

In this case we will use bus that is capable to communicate across multiple computers. We can either use existing implementations (eg. JMS, TibcoRV, JGroup, etc) or implement our own.

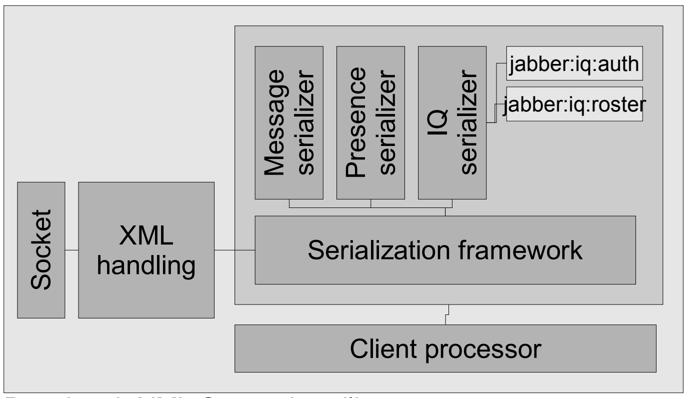
I will only describe what we can do with such buses.

TODO dodělat

7.2 XML handling

In the next two chapters we will discuss how to implement XML handling. This represents whether the system will be able to support 1000 or 200 000 connections.

For both cases we will have a scheme shown on Drawing 4 XML Stanza handling.



Drawing 4 XML Stanza handling

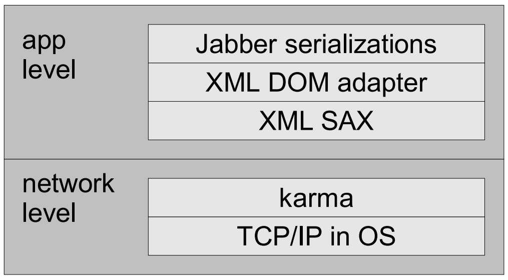
We read data from socket and pass it to XML handling – that parses DOM from it and calls Serializations framework. It implements View part from Model View Controller (see Serializations/Deserializations vs. direct processing. page 27). It converts XML to Model and passes it to a client processor.

A client processor is very simple. It sends all data over the bus to an attached session processor. The client processor also handles SASL and TLS commands (not encrypting in TLS/SASL but controls when to start it and put handlers between socket level and XML handling level).

Bear in mind that the difference between small and scalable solution is only on the lower levels. It means that you can change from small to scalable solution without the need to rewrite serialization framework or client/session processor.

7.2.1 Small implementation

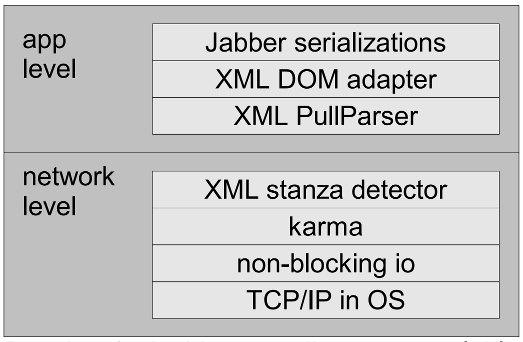
Here I will show the simplest solution that scales only to 1000 concurrent connections. The idea is to use only SAX parser and not asynchronous I/O operations. It is just simpler. We provide callbacks to SAX parser and we also provide low level reader from socket. Karma implementation is also simpler because when we read more data then allowed we can just put the thread to sleep. We also need SAX to DOM adapter that creates DOM only from part of XML document (one stanza). Again we need some kind of hack for `<session:session>` support.



Drawing 5 Small architecture diagram

7.2.2 Scalable implementation

This solution can scale from 100 000 to 1 000 000 depending on the used OS. This solution is harder to implement – we must use non-blocking I/O operations (what is always harder to handle), XML Stanza Detector and asynchronous karma. You must use asynchronous XML parser. How to handle non-blocking IO read either [ACE-OVERVIEW] or [JAVA-NIO].



Drawing 6 Architecture diagram - scalable solution

To know how to implement an asynchronous parser look see chapter Non-blocking XML parser page 25.

8 Implementation details

In this chapter we will see more details about implementation I made. The software is written in the Java and uses many external libraries. It requires J2SE 1.4 or higher except one part that requires J2SE 1.5 for asynchronous TLS support (in earlier version it was not available).

For compilation and running you will require program `ant` (it is java equivalent for `make`). For support of J2SE 1.5 you will need at least version 1.6 (packaged on attached CD). For more info how to run or compile this software see User guide on page 33.

8.1 Components

Here I will divide the system into parts.

TODO dodělat

8.2 Class diagram

TODO dodělat

8.3 Sequence diagram

TODO dodělat

9 User guide

Just start :-)

TODO dodělat

TODO – do příloh

10 Resume

blablabla

TODO dodělat

Abbreviations

- **User** – how user is uniquely identified in the IM system
- **ICQ** (http://networking.webopedia.com/TERM/I/ICQ.html) **-** An easy-to-use online instant messaging program developed by Mirabilis LTD. Pronounced as separate letters, so that it sounds like "I-Seek-You," ICQ is similar to America OnLine's popular Buddy List and Instant Messenger programs. It is used as a conferencing tool by individuals on the Net to chat, e-mail, perform file transfers, play computer games, and more.
- **IETF** (http://networking.webopedia.com/TERM/I/IETF.html) - Short for ***I**nternet **E**ngineering **T**ask **F**orce,* the main standards organization for the Internet. The IETF is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. It is open to any interested individual.
- RFC
- **Jabber** (http://networking.webopedia.com/TERM/j/jabber.html) - an open XML protocol for message and presence exchange in real time between two points on the Internet. Jabber's asynchronous instant messaging (IM) platform is similar to IM systems such as AIM, ICQ and MSN but is open source, extensible through XML, decentralized (allowing anyone to run a Jabber server), and any Jabber server can be isolated from the public Jabber network in order to increase security. The Jabber project was started by Jeremie Miller in early 1998.
- **LDAP** (http://networking.webopedia.com/TERM/L/LDAP.html) - Short for ***L**ightweight **D**irectory **A**ccess **P**rotocol,* a set of protocols for accessing information directories. LDAP is based on the standards contained within the X.500 standard, but is significantly simpler. And unlike X.500, LDAP supports TCP/IP, which is necessary for any type of Internet access. Because it's a simpler version of X.500, LDAP is sometimes called *X.500-lite.*

  Although not yet widely implemented, LDAP should eventually make it possible for almost any application running on virtually any computer platform to obtain directory information, such as email addresses and public keys. Because LDAP is an open protocol, applications need not worry about the type of server hosting the directory.
- **MMS** (http://networking.webopedia.com/TERM/M/MMS.html) - Short for ***M**ultimedia*

Message Service, a store-and-forward method of transmitting graphics, video clips, sound files and short text messages over wireless networks using the WAP protocol. Carriers deploy special servers, dubbed MMS Centers (MMSCs) to implement the offerings on their systems. MMS also supports e-mail addressing, so the device can send e-mails directly to an e-mail address. The most common use of MMS is for communication between mobile phones.

MMS, however, is not the same as e-mail. MMS is based on the concept of multimedia messaging. The presentation of the message is coded into the presentation file so that the images, sounds and text are displayed in a predetermined order as one singular message. MMS does not support attachments as e-mail does.

To the end user, MMS is similar to SMS.

- **Presence** (http://networking.webopedia.com/TERM/P/presence.html) - The ability to detect whether other users are online and whether they are available. Presence services are commonly provided through applications like Finger and instant messaging clients, although a number of companies are developing products in other areas that leverage presence, such as VoIP.
- **SIMPLE** (http://networking.webopedia.com/TERM/S/SIMPLE.html) - Short for *Session Initiation Protocol (SIP) for Instant Messaging and Presence Leveraging Extensions*, an application of the SIP protocol for server-to-server and client-to-server interoperability in instant messaging. SIMPLE is a step in bringing standardization to instant messaging.
- **SIP** (http://www.webopedia.com/TERM/S/SIP.html) - *Session Initiated Protocol*, or *Session Initiation Protocol*, a signaling protocol for Internet conferencing, telephony, presence, events notification and instant messaging. The protocol initiates call setup, routing, authentication and other feature messages to endpoints within an IP domain.
- **SMS** (http://networking.webopedia.com/TERM/S/short_message_service.html) - Short Message Service (SMS) is the transmission of short text messages to and from a mobile phone, fax machine and/or IP address. Messages must be no longer than 160 alpha-numeric characters and contain no images or graphics.
- **Store and forward** (http://networking.webopedia.com/TERM/S/store_and_forward.html) A technique

common in messaging services where a data transmission is sent from one device to a receiving device but first passes through a "message center." The message center is typically a server that is used by the message service to store the transmitted message only until the receiving device can be located, and it then forwards the transmission to the intended recipient and deletes the message from the server.
A common type of store-and-forward messaging is that used between mobile phones.

- **VoIP** (http://networking.webopedia.com/TERM/I/Internet_telephony.html) - A category of hardware and software that enables people to use the Internet as the transmission medium for telephone calls. For users who have free, or fixed-price Internet access, Internet telephony software essentially provides free telephone calls anywhere in the world. To date, however, Internet telephony does not offer the same quality of telephone service as direct telephone connections.
There are many Internet telephony applications available. Some, like CoolTalk and Net-Meeting, come bundled with popular Web browsers. Others are stand-alone products. Internet telephony products are sometimes called *IP telephony, Voice over the Internet (VOI)* or *Voice over IP (VOIP)* products.
- **NAT** () – Network Address Translation

11 Bibliography

- [IMP02] XMPP Gets Second Vote of Confidence

  http://www.instantmessagingplanet.com/enterprise/article.php/3311211
- [IETF-XMPP-01] IETF XMPP Working group goals

  http://www.ietf.org/html.charters/xmpp-charter.html

Bibliography

ZDNET01: Matthew Broersma, Jabber numbers overtake ICQ,
RFC 2778: , RFC 2778,
IETF-IMPP-SRV: , Address Resolution for Instant Messaging and Prese, ,
IETF-IMPP-PRES: , Common Profile for Presence, ,
IETF-IMPP-IM: , Common Profile for Instant Messaging (CPIM), , http://www.ietf.org/internet-drafts/draft-ietf-imp
RFC 2779: , RFC 2779, ,
IETF-XMPP-IM: , Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Pr, , http://www.ietf.org/internet-drafts/draft-ietf-xmp
IETF-XMPP-E2E: , End-to-End Object Encryption in the Extensible Messaging and Presence, , http://www.ietf.org/internet-drafts/draft-ietf-xmp
IMP01: , A Lack of SIMPLE Pleasures, , http://www.instantmessagingplanet.com/enterprise/a
ACE-OVERVIEW: Douglas C. Schmidt, Overview of ACE, , http://www.cs.wustl.edu/~schmidt/ACE-overview.html
XMLPP: , XML Pull Parsing Common API, , http://www.xmlpull.org/
JAVA-NIO: , New I/O Java APIs,

Index of Tables

Illustration Index

12 Links:

http://www.instantmessagingplanet.com/

http://www.jabber.com/

http://www.jabber.org/

http://www.icq.com/

TODO – kam směřují a co obsahují

13 Appendix A - Operating system limits

This limits are mentioned here only for informational purpose. These limits are OS specific and their bypassing is also OS specific. These issues must be bypassed on every machine that will be part of cluster.

From these limits you can see that the practical maximum of TCP/IP connections on one machine is by default 1024 and can be increased somewhere between 200 000 and 1 000 000.

TODO – určit nebo rozdělit podle OS – dělám hlavně pod linuxem a windowsům se věnuji jen okrajově.

13.1 1024 file descriptors limit

Some operating systems have a limit for opened file descriptors (and TCP/IP connection is also identified by file descriptor). For example, on linux there is by default limit 1024. This limit is for security reasons and can be easily increased by calling "ulimit"

```
ulimit -n 1000000
```

To increase this value, we must have at least kernel version 2.4. There is the next limit at the value of about 1 000 000 on 2.6.x.

Solution:

calling "ulimit" on linux. For this operation we must have enough permission (must be root) or we can increase the limit for user running program in system configuration. See system administration guide.

13.2 5 000 – 10 000 connections on Windows XP Home, Pro

All Microsoft Windows versions except server editions have software limits at low number. This limit is added by Microsoft to move users to server editions. This limit is somewhere between 5 000 and 10 000.

Solution:

Use either server edition or apply undocumented windows registry change that increases this value.

13.3 28 232 outgoing connections on Linux

When the program establishes a connection to the remote port and does not specify source address, source port is taken from limited pool. The size of this pool is specified in /

proc/sys/net/ipv4/ip_local_port_range.

To increase this value simply use

```
echo "1024  65535" > ip_local_port_range
```

Maximal value is 64 511. All these limits are per source address. You can increase this limit by using multiple outgoing addresses.

78 579 system wide opened files on Linux

There is default limit 78 579 for opened files in system. This can be easily increased using:

```
echo 512000 > /proc/sys/fs/file-max
```

13.4 10 000 limit of threads

Most of the existing operating systems have limits on existing threads in one application. There is usually a problem with increased complexity of large number of threads. MS Windows is good in handling large a number of threads (~ 10 000), linux >= 2.4 slow downs with more than 1 000 threads and linux 2.6 should be good even in handling 1 000 000 threads.

There is also the problem that every thread must have a hole in virtual memory for stack. This can easily consume all virtual memory (even physical memory is still free).

We recommend using non-blocking io operation with thread pool instead of thread per connection.

13.5 Connection memory consumption

The operating system must create some memory structure for every connection that is established. This is usually less than 1000 bytes. When you establish a large number of connections, it can eat a lot of memory. But this consumption is much smaller that we will use in jabber implementation.

13.6 Syscall "select" 1024 limit

The system call "select" is limited to 1024 file descriptors in one call. We recommend using "poll" instead of "select".

13.7 TCP/IP in clusters

In large enterprises it can be a problem that jabber uses TCP/IP. With TCP/IP it is not possible to have cluster of machines and let every received XML stanza be handled by a different computer. One way would be implementation of TCP/IP in user space (that is unusable – too error prone) and the other way is to use gate machines (ie transferring from TCP/IP with XML to some internal representation).

14 Appendix B – Measuring implementation

TODO dodělat

14.1 Machine configurations

The values were measured on this machines:

Machine 1 (client):

Noname machine

AMD Athlon XP 1800+, 512MiB RAM

NVIDIA nForce 2 chipset - K7N2G

Linux Debian 3.0 with vanilla 2.6.4.

SUN Java version 1.5.0-beta-b32c.

Machine 2 (server):

Machine IBM ThinkPad X31

Pentium M 1.4, 756 MiB RAM (CPU frequency scaling was disabled).

Centrino chipset

Linux Debian 3.0 with vanilla 2.6.3.

SUN Java version 1.5.0-beta-b32c.

Common parameters

- Java compiler configuration: optimization on, debugging off
- JVM configuration: -server -mx128 -ms64
- In class `cz.ferschmann.im.Config` I modified compile time switch for debugging logging to off.
- both machines were connected using 100Mibit ethernet addapter.

Machines were during measurements only CPU bound. We have enough memory and we tried to have enough network bandwidth.

14.2 Test description

The goal of this measurements is to prove that the prototype server is capable of handling up to 100 000 concurrent connections. For this purpose we created scalable client that is able to handle thousands of connections.

We just start TCP/IP connections, authenticate using JEP-78 and set presence. We do not ask for roster. We made measurements on the server (how many connections we can made at a time) and also on the client (how long took to create one connection).

14.3 Results

TODO dodělat

14.4 Resume

From measured values it can be easily seen that difficulty of handling connections grows linearly. This can be a problem with a very large systems. I think that the problem is somewhere between the operating system and the API (this time poll). OS must at each step walk thru all connections to see all changes. To lower this problem we can use multiple connection manager (i.e. threads waiting in poll) – eg. every one will handle about 1000 - 2000 connections. So with about 1000 threads we can get to 1000 000 active connections. But this solution was not verified.

TODO dodělat