

Institute of Computer Science
Department of Programming and
Software Engineering



Diploma Thesis

PeerStorm: Distributed Editing of Conceptual Structures

Thomas Müller

Assigned by: Prof. Dr. Martin Wirsing
Supervisor: Axel Rauschmayer
Submitted on: 31.05.2005

Erklärung

Hiermit versichere ich, daß ich diese Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 31.05.2005

.....
Thomas Müller

Abstract

In times of globalisation large enterprises have become distributed all over the world. Teams from different part of the world have to work with each other. They have a need for communication infrastructure as well as for a collaboration infrastructure. We present here PeerStorm, a solution based on the Hyena platform for RDF editing. It is an extension on Hyena with transparent collaboration through a publish/subscribe system and a realtime update mechanism. It also has a Instant Messaging system which uses Jabber as its protocol. Therefore it is capable of working with any Jabber account, even with the ones from GoogleTalk and gives the possibility of Multi User Chats.

Contents

1	Introduction	3
2	Background	6
2.1	How to Distribute	6
2.1.1	Networking	6
2.1.2	The Client / Server Approach	6
2.1.3	The Peer to Peer Approach	8
2.2	Networked Editing	10
2.2.1	Wiki Editing	10
2.2.2	Versioning Systems	10
2.2.3	Direct Editing Systems	11
2.3	Instant Messaging	11
2.3.1	Protocols	12
2.3.2	Jabber	13
2.3.3	Non standard extension	21
2.4	Conceptual Structures	21
2.4.1	Semantic Web	21
2.4.2	Resource Description Framework	22
2.4.3	RDF as Triples	23
2.5	Hyena or how to Handle RDF	24
2.6	Eclipse Plugins	24
3	Extending Hyena with Chat Functionality	26
3.1	The Chat Interface	26
3.1.1	The Userlist View	26
3.1.2	The Chat View	27
3.2	Smack Library	28
3.2.1	Connecting to an XMPP Server	29
3.2.2	Buddy Management	29
3.2.3	Presence Awareness	30
3.2.4	Single User Messaging	30

3.2.5	Multi User Chat	31
4	Extending Chat with Document Management	32
4.1	Extension to Jabber	32
4.2	The Protocol Inside	33
4.2.1	Remote Document Management	33
4.2.2	Remote Document Subscription	34
4.2.3	Remote Document Synchronisation	35
5	Examples	37
5.1	Example: Distributed Wiki RDF Editing	37
5.2	Use Case: Distributed Outline Editing	39
6	Future Research	41
	List of Figures	42
	List of Tables	43
	Bibliography	44

Chapter 1

Introduction

In times of globalisation large enterprises have become distributed all over the world. Especially in software development it is very important to have service people near your customers. So data is most often sent across the world and even development teams from different parts in different countries work together on the same project. Before using computerized support distributed editing already had been a use case for many companies. A workflow then could have looked like this. Some authors write a document, perhaps a use case analysis in a software engineering process, each a different part. When finished it will be composed by a person to one single document and sent to each responsible person (mostly not the authors) to revise it. After that the authors get a corrected version back and fix the document. Then again the document will be sent to each responsible person because they don't know about each others changes. This process will be looped until everybody agrees with the version. Then the document gets published. If you somewhen worked in a middle-sized or bigger company this perhaps looks very familiar because even today this is how it is done. However in times of electronic data processing the production of the document became faster since you don't have to rewrite the whole document. But nevertheless there are companies where these drafts are still printed out and sent via interoffice mail instead of using for example email. That is because of a missing digital workflow management system. But since the globalisation of companies and also of their employees who have to be more mobile and flexible than they used to be in the past, the development starts to use the internet more. Current development outside the open source and university software works into direction of workflow management. IBM for example as others has a document management environment inside their current version of WebSphere 5.1. It is an integrated versioning system combined with a rights management to make it possible to check against given workflows.

Another example is open source software where the community is most often spreaded all over the world. Most open source software projects use a versioning system like CVS or the newer ones use SVN. Those are described more in detail by section 2.2.2. While it is not enough to simply have the code in a versioning system where many people can work with you also have to do a lot of communication. The discussion about the goals to achive in a new milestone for example is most often done by email or to be precise in mailing lists. But many projects also use chat systems like Internet Relay Chat or even Instant Messengers with Multi User Chat. How Instant Messaging works is described in section 2.3.

For software engineering purposes it is interesting to use the Ressource Description Framework (RDF) as a universal data structure which stores information in a graph. Its nodes are URIs which can describe real world "things" as well as web based links. Through edges one can correlate two nodes and name the correlation. Another advantage is the possibility to embed XML so it is possible to manage semi-structured data as well. More about RDF you will find in section 2.4.2. What we want to do and actually did is to create an extension on Hyena, a special RDF editor platform for software engineers, which gives the possibility of collaboration and communication in one product. Both is done based on the Jabber protocol (see 2.3.2) which is the same protocol GoogleTalk is based on. If simply using the chat functionality you even can use your GoogleTalk account or speak from any other Jabber account to people using GoogleTalk. It is even possible to do some Multi User Chat as described in 3.2.5. Two or more people using Hyena with the PeerStorm extension can do collaborative work by one publishing a document and the others can subscribe to it via Jabber. Then they can work on the same document specified by a Unique ID in a transparent manner. Each event will be caught by PeerStorm and transfered to the main document where the actual change is done. Then all the subscribed clients will be notified. So it feels the same as you would start Hyena in standalone mode. For example if you want to write a paper outline with a colleague you can both start Hyena with PeerStorm and the Outline vodule. You can discuss via the chat module with each other, discuss issues and you can edit the outline with the advantage that instantly each other sees the changes. If you both agree on the version you can both save the state. Editing RDF is partially a bit tricky when it comes to special features like blank nodes. You can't simply use the RDF XML representation since blank nodes are just counters there. So we needed to get them unique IDs to make them transferred correctly.

There is a part of the Remote Procedure Call extension implemented as well. It is adapted for use in Java and Hyena. This makes sure that further improvements in Hyena can easily be added to PeerStorm. There are some improve-

ments which are not done yet but perhaps will be in the future. One of them is an authentication for document subscription, another is the use of Jingle, a JEP for making Peer to Peer communication available. But first let us have a closer look of what is all needed to achieve our goals.

Chapter 2

Background

First we have a look at distribution and distributed editing. Thereafter we talk about what we want to distribute and edit.

2.1 How to Distribute

Many ways exist to get a message broadcasted. On the one hand you have the standard one way of publishing like newspapers, magazines, books as well as one way broadcasts like television, radio or mobile push services. On the other hand there is the internet as a bidirectional way of communication. Users just pull the messages they want to see or hear out of the web and a publisher can give them the possibility to comment or discuss the points.

2.1.1 Networking

The first computers were very large and had no connection with each other. This changed in the 1960s when first master-slave connections were established. From then there were the big central machines with plenty of power and the small terminals connected to them. Until the early 1980s this architecture remained the same even if the terminals evolved to PCs by becoming visual display units with growing processing power and dial-up connections.

2.1.2 The Client / Server Approach

Because of the change in capacity and capability of the PCs the architecture itself changed. The masters more and more became servers and the remote devices became clients [Cla04]. On clients you could find special client software which requested services from the server software on the servers. A simple example is an email client requesting the content of the mailbox on an email

host. The term server is used for either the hardware, the software or the whole system of hardware and software [Wik06a]. We prefer some piece of software which runs continuously, is reachable through a network and waits for connections to be a server because we want to talk about software not hardware. If a connection occurs it starts to exchange messages with the client which requested the connection.

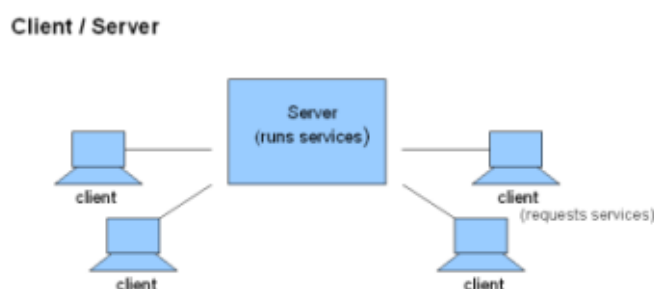


Figure 2.1: Client Server Schema: Here you can see the central server with its communication lines (may be physical but mostly logical) to the clients.

A client as well can be related to either software, hardware or both. You also need to know that there are two sorts of clients. On the one hand there are thin clients which are used in a similar way as the remote terminals in the master-slave architecture. They have less power compared to thick clients and almost no application logic. As minimal clients they request a little boot image from a remote server and connect to an application server after using this boot image for startup. On the other hand thick clients use their full computing power to do the computings and use the server mostly just as a huge multi user data storage. In the 1990s client / server became the mainstream and still remains so. Nevertheless there are some major issues with this architecture as you can see in table 2.1.

robustness	service depends on a single server (or server farm) running this service
scalability	adding clients means reducing data transfer for all users, adding servers is very expensive because the number of users servable is
vulnerabiliy	denial of service attacks are very common, but masquerading and data pollution attacks occure as well

Table 2.1: Significant Problems with Client / Server Architecture

2.1.3 The Peer to Peer Approach

To address the major issues of the client / server architecture the peer to peer approach has been reinvented. The first occurrence of the peer to peer model was not the current filesharing clients. There are several examples like ARPANET, USENET or even the email forwarding system via SMTP which all were in use before 1980 and which all are sort of peer to peer systems. There are three major types of peer to peer network architectures:

Pure P2P:

- Peers act as clients and server
- There is no central server managing the network
- There is no central router

Hybrid P2P:

- Has a central server that keeps information on peers and responds to requests for that information.
- Peers are responsible for hosting the information (as the central server does not store files), for letting the central server know what files they want to share, and for downloading its shareable resources to peers that request it.
- Route terminals are used addresses, which are referenced by a set of indices to obtain an absolute address.

Mixed P2P:

- Has both pure and hybrid characteristics

Figure 2.2 should paint a clearer picture of what the difference is.

Pure peer to peer is rare because the overhead of asking peer nodes for their content is for example bigger than asking supernodes or central index servers. That is why there is mostly one or more central instances for indexing. But the transfers or computings are done between peers without the central instance. An example for a pure peer to peer network is the original Gnutella network. You just needed one entry point and from there you got additional peer addresses if the current connected peer could not fulfill your request. Nevertheless the performance was not as good as for example the FastTrack network which is an example for a hybrid peer to peer network. As the name intends it is really faster because of its web of supernodes doing indexing for a group of clients. So one hop in FastTrack would mean several

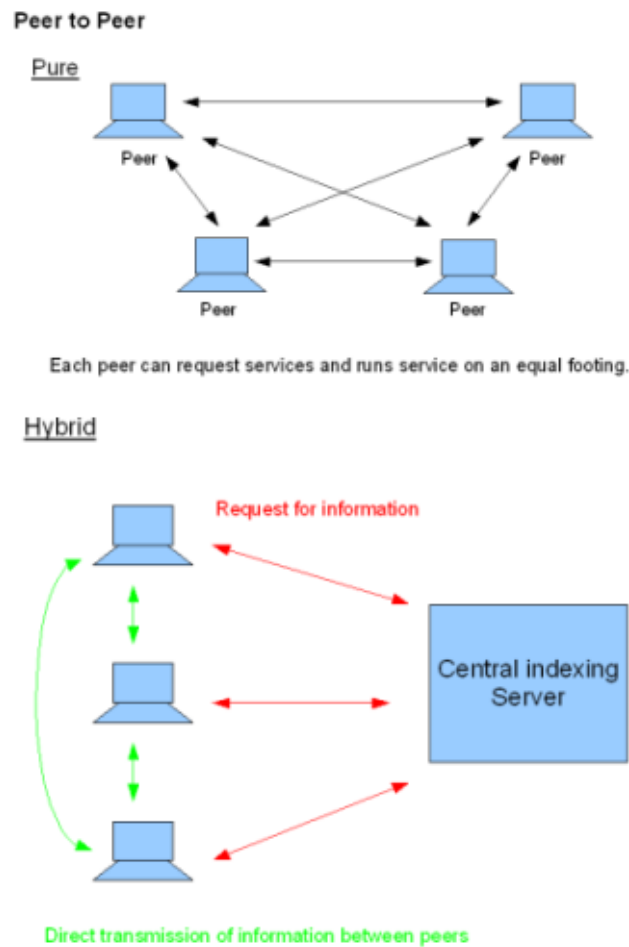


Figure 2.2: Peer To Peer Schema: This Picture shows the logical difference between pure P2P and Hybrid P2P.

hops in Gnutella.

Important characteristics need to be satisfied in order to have an advantage before the client / server approach. The first is that many devices must perform server-functions to overcome the dependency on one single server and the bottleneck of the connection to the subnet. Another important thing is that all of the devices which perform server-functions have to be able to perform all server-functions on the network. Otherwise you still would have the bottleneck just reduced for some of the functions and not for the overall reliability. As well important is that peers with server-functions assist devices which act as clients to find more servers.

Characteristics of applications which would gain performance out of this architecture would be applications with a really huge demand on processing power.

This could be some brute force key discovery process or some pattern searching like SETI@home. On the other hand there would be applications which use the snowball effect of widespread digital objects like files. This could be some update service which uses the connection while downloading the patch for uploads as well. Azureus is a BitTorrent client which uses such a service already.

2.2 Networked Editing

There exist several attempts to support editing in network environments. Some of them will be presented here. In order to have a distributed team work on the same project some major points have to hold. First is the importance of communication. Without communication the project will surely fail. There has to be an agreement about the tasks and their assignment. Everybody has to know his task. Furthermore there has to be a platform everybody uses to have consistent data. This is a very difficult point since it is possible that different people work within the same document. That is where the following things can help.

2.2.1 Wiki Editing

Wiki editing is a very popular way of creating articles on the web. Wikipedia grows and becomes more and more the most comprehensive encyclopedia on the net. Wiki is a synonym for easy changes on almost every page by anyone. A wiki enables documents to be written collectively in a very simple markup language using a web browser. A single page in a wiki is referred to as a "wiki page", while the entire body of pages, which are usually highly interconnected via hyperlinks, is "the wiki"; in effect, a very simple, easier-to-use database [Wik06c]. Through a versioning system a page can be reverted to any of its previous stages easily. Nevertheless it is by now only used for web pages so its primary target is online documents. However there exist an approach to use wiki as a semi structured presentation layer for structured data as well. See [AR] for further information.

2.2.2 Versioning Systems

There are several versioning systems in use. These systems provide a way to create a history of a document (mostly development source files), to tag versions as well as create different branches with different attempts to a problem. Any change should be commented so that every user working on the project knows what you have done. Another advantage in using versioning systems is

that in most cases the system can merge changes on the same file automatically. This also works for every file using pure text. Concurrent Versioning System (CVS) is such a system. It is a client / server system which works after the "copy-modify-merge" model instead of the "lock-modify-unlock" model. The key advantage is that you don't have to wait for others to finish their modifications before you can do your own modifications. Subversion (SVN) is another project similar to CVS but with some major advantages in directory versioning, file renaming and file moving. It also supports atomic commits so interrupted commit operations do not cause repository inconsistency or corruption.

2.2.3 Direct Editing Systems

Direct editing systems are used to collaborate with others on one single document and to see changes on the fly (if the network is fast enough). The current available systems are for example SubEthaEdit, MoonEdit and others. They have in common that they are used for pure text files only.

SubEthaEdit is a collaborative editor for Mac only. Its main features are:

- syntax highlighting
- symbol navigation
- auto-completion
- indenting
- built-in support for most common programming languages
- HTML export

You can work on the same file in realtime. MoonEdit is a more basic collaborative editor. Therefore it is a multi-platform editor which is its major point. You can use it with Windows, Linux or FreeBSD.

2.3 Instant Messaging

"Instant messaging is the act of instantly communicating between two or more people over a network such as the Internet." [Wik06b] This is the shortest but in my opinion best description of Instant Messaging.

To use Instant Messaging you need the following things:

- An Instant Messaging Service Provider (IMSP)
- An Instant Messenger (a software client to connect to the chosen IMSP)

- An Account with your IMSP
- In general an internet connection

Your communication partner needs this as well. Since there are many IMSP which are most often not compatible with each other, you both should have the same IMSP. The time you both logged into the client software you can add each other to your personal buddy list or contact list. Then you can start talking to each other through a "private chat". This is the basic featureset of an Instant Messenger. Nevertheless since its first occurrence the instant messengers evolved to full featured chat client where you can do Multi User Chat (MUC) as well as Voice Chat or Video Chat. You can send files to other users or even SMS to other devices like mobiles.

Each IMSP has its own Client with most of them using own proprietary protocols. At the moment there are the following mainstream clients: AIM, Google Talk, ICQ, Microsoft Messenger for Mac, MSN Messenger, NateOn, QQ, Skype, Windows Messenger and Yahoo! Messenger. There are multi protocol clients as well for example gaim or trillion.

Jabber(XMPP) is a protocol which has to be talked about a little more in detail. It is a free standard maintained by the Jabber Foundation. It is an XML protocol with the possibility to expand the protocol by introducing Extensions. So if you want to support a new format you just add an xml wrapper around and make the clients aware of the new format. Old clients just ignore these packages. So it is always downwards compatible.

2.3.1 Protocols

There are almost the same amount of Instant Messaging Protocols as there are Instant Messengers. As a matter of economy none of them is open standard except one. The only standardized open protocol is by now the Jabber (or XMPP as it is called since standardization) protocol which is described in the next section. Here we describe a version of the MSN Messenger protocol based on some reverse engineering documentation available on the internet. This should just give a short introduction how the proprietary protocol works in common. On MSN there are two kinds of servers you connect to. To simply be available and see the availability of your buddies you connect to the notification server. This servers purpose is mainly the presence notification, but it also let you create or join switchboard sessions. The switchboard servers are the second type of servers. They handle video, audio or chat sessions as well as file transfers and other services. A standard connection looks like in figure 2.3 shown. First thing is we connect to the dispatch server to negotiate the protocol version used for the connection (1., 1.reply). MSN Servers are capable

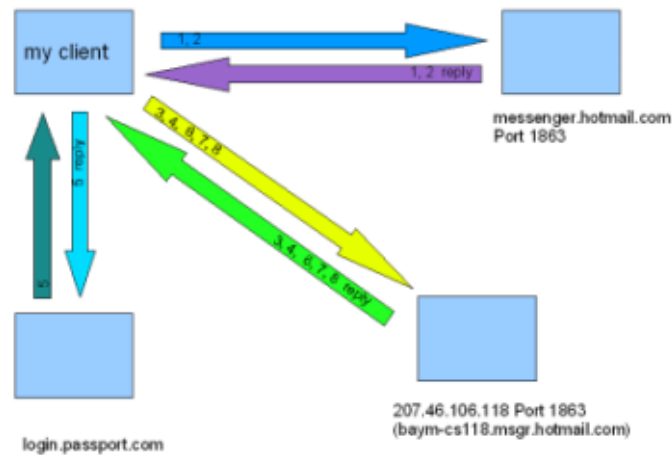


Figure 2.3: MSN Messenger Example Connection: For description please have a look into the text

of using more than one protocol versions even if they are not compatible. What now happens is that we are redirected to one of the notification servers. This is done for load balancing reasons (2., 2.reply). That is where we have to do a protocol negotiation again since it is another server we connect to (3., 3.reply). Now we want to login. The server returns a long string for use in Passport authentication (4., 4.reply). Then we do a MSN Passport authentication. With the reply of the Passport server we get an ID (5., 5.reply) which we then can use for our MSN Messenger session login (6., 6.reply). Once we are logged in the local buddy list will be synchronized with the server stored one (7., 7.reply). Last thing is that the initial presence will be sent (8., 8.reply). For gathering more information about the MSN Protocol you should have a look at [MM].

2.3.2 Jabber

The Jabber protocol or Extensible Messaging and Presence Protocol (XMPP) as it is called now is an open, secure, ad-free alternative to consumer IM services like AIM, ICQ, etc. The people from Google Talk use the Jabber Protocol as well for their client. This is why they support the Jabber foundation with information about their product to create a bridge between these similar techniques (see jabber.org JEPs). XMPP is a set of XML protocols. The XMPP Core Protocol [SA04c] is about the xml streaming technology, the SASL and TLS security options as well as the stanza semantics. The XMPP IM protocol [SA04a] is an instant messaging extension to the core protocol for basic instant messaging and presence handling. Further on there is a protocol for mapping

XMPP to the IETF's¹ CPIM² specifications [SA04d]. This is used to build bridges between Jabber and other IM services. The last RFC created by the XMPP workgroup [SA04b] is about end to end encryption and defines the possibility to encrypt messages and presence stanzas as well as objects for an end to end secured communication. There are plenty of additional extensions defined in Jabber Enhancement Proposals (JEPs) which are managed by the Jabber Software Foundation.

2.3.2.1 The Core Protocol

Client Server Connection Clients normally connect via a TCP connection directly to a server. It is possible to connect with the same account but different resources. A resource is part of a Jabber Identifier (JID). See also the paragraph about Jabber Identifier.

Net of Servers The XMPP system consists of a net of servers. This means each server has the (optional) possibility to communicate with other servers running a XMPP service. This is a straightforward extension of the client server communication. This is similar to for example the Simple Mail Transfer Protocol (SMTP) and is a peer to peer connection since a server can virtually act as server as well as act as a client.

Jabber Identifier In XMPP every network endpoint has a Jabber Identifier(JID). This JID is its address. The syntax for JIDs in Augmented Backus-Naur-Form is

```
jid           = [node '@'] domain [ '/' resource]
domain        = fqdn / address-literal
fqdn          = (sub-domain 1*('.' sub-domain))
sub-domain    = (internationalized domain label)
address-literal = IPv4address / IPv6address
```

[SA04c]

XML Stream "Definition of XML Stream: An XML stream is a container for the exchange of XML elements between two entities over a network. The start of an XML stream is denoted unambiguously by an opening XML <stream> tag (with appropriate attributes and namespace declarations), while the end of the XML stream is denoted unambiguously by a closing XML

¹IETF is the Internet Engineering Task Force

²CPIM stands for Common Presence and Instant Messaging

</stream> tag. During the life of the stream, the entity that initiated it can send an unbounded number of XML elements over the stream, either elements used to negotiate the stream (e.g., to negotiate Use of TLS (Section 5) or use of SASL (Section 6)) or XML stanzas (as defined herein, <message/>, <presence/>, or <iq/> elements qualified by the default namespace)” [SA04c].

XML Stanza ”Definition of XML Stanza: An XML stanza is a discrete semantic unit of structured information that is sent from one entity to another over an XML stream. An XML stanza exists at the direct child level of the root <stream/> element and is said to be well-balanced if it matches the production [43] content of [XML]. The start of any XML stanza is denoted unambiguously by the element start tag at depth=1 of the XML stream (e.g., <presence>), and the end of any XML stanza is denoted unambiguously by the corresponding close tag at depth=1 (e.g., </presence>). [...] The only XML stanzas defined herein are the <message/>, <presence/>, and <iq/> elements qualified by the default namespace for the stream” [SA04c].

The <message/> stanza is used to push some information from one entity to another one addressed by the to attribute.

The <presence/> stanza is a notification broadcast which may be sent to a single entity by giving the to attribute. However if the attribute is not set it will be broadcasted over the servers of the XMPP web. This means every server is notificated of the presence of a user but a client only gets notified if he subscribed for this special user presence.

The <iq/> stanza is used for a request/response mechanism similar to HTTP. One entity sends a request with type get or set to another entity that has to answer with type result or error. The stanzas of the request and the corresponding response must have the same id attribute. Here is an example iq communication (we assume that the stream already has been started in both directions):

```
<iq type='get'
  to='name@server.de'
  from='name2@example.com/pda'
  id='probing1'>
  <query xmlns='jabber:iq:version' />
</iq>
<iq type='result'
  to='name2@example.com/pda'
  from='name@server.de'
  id='probing1'>
  <query xmlns='jabber:iq:version'>
```

```
<name>name</name>
<version>1.4.1</version>
<os>Linux 2.6.9</os>
</query>
</iq>
```

This example sends a request to name@server.de to check which version he uses. The response tells that the version is 1.4.1 and he uses a Linux Kernel 2.6.9.

2.3.2.2 The Basic Instant Messaging Protocol

There are still only the three stanzas defined before but additional constraints are given to attributes and/or child elements .

The type attribute of the <message/> stanza is constrained. The attribute is declared as recommended. If it is not given or the client does not understand it the client should assume that the type is normal.

chat	The message is sent in a one to one communication context.
error	An error occured.
groupchat	The message is sent in a groupchat context (many to many communication)
headline	The message is a no reply message (perhaps by automated services)
normal	The message is without a context. Replies are possible.

Table 2.2: Possible values for the type attribute of message stanzas

The <message/> stanza may contain four child elements where the <error/> element is already defined in the Core protocol. The <subject/> element specifies the topic of the underlying message. The <body/> element specifies the text of the message. The <thread/> defines the context of the message. This is used in groupchat or chat messages.

The type attribute of the <presence/> stanza is constrained. The attribute is declared as optional. If it is not given the <presence/> stanza is interpreted as a online and available notification of the sender.

The <presence/> stanza may may also contain four child elements where the <error/> element is already defined in the Core protocol. The optional

unavailable	The entity is not available for communication any more.
subscribe	The sender entity wishes to subscribe to the recipient's presence notification.
subscribed	The sender entity allowed the recipient to receive his the presence notification.
unsubscribe	The sender entity unsubscribes from another entity's presence notification.
unsubscribed	The subscription request has been denied or a existing subscription has been cancelled.
probe	Request for an entity's current presence.
error	An error occured either in processing or delivery.

Table 2.3: Possible values for the type attribute of presence stanzas

<show/> element is used to specify a particular availability status and can have the following values:

away	The entity or resource is temporary away.
chat	the entity or resource is actively interested in chatting.
dnd	The entity or resource id busy (dnd = "Do Not Disturb").
xa	The entity or resource is away for an extended period (xa = "eXtended Away").

Table 2.4: Possible values of the show element

[SA04a]

The optional <status/> element gives additional plain text information about the availability mode e.g. why the entity or ressource is away.

The optional <priority/> elements value must be an integer between -128 and +127. This value is used to determine which resource should get the message if the message has no resource in its to attribute. The highest priority wins. However no priority is handled as zero priority.

2.3.2.3 Selected JEPs

JEP-9: Jabber-RPC Jabber-RPC defines a way to transport Remote Procedure Calls over the Jabber Protocol. It is similar to XML-RPC (see [Win03]).

XML-RPC unlike Jabber-RPC defines HTTP as the only valid transport mechanism which created the need to label the transport over Jabber as Jabber-RPC. An `<iq/>` stanza of type `set` is used to transport the request and a stanza of type `result` for the response. The `<iq/>` stanza contains a single query child element. The direct child of the query element is either one of `methodCall` or `methodResponse` for a request or a response. The encoding of the Jabber-RPC payload is UTF-8.

A typical request and the corresponding response [Ada06]:

```
<iq type='set'
  from='requester@company-b.com/jrpc-client'
  to='responder@company-a.com/jrpc-server'
  id='rpc1'>
  <query xmlns='jabber:iq:rpc'>
    <methodCall>
      <methodName>examples.getStateName</methodName>
      <params>
        <param>
          <value><i4>6</i4</value>
        </param>
      </params>
    </methodCall>
  </query>
</iq>
<iq type='result'
  from='responder@company-a.com/jrpc-server'
  to='requester@company-b.com/jrpc-client'
  id='rpc1'>
  <query xmlns='jabber:iq:rpc'>
    <methodResponse>
      <params>
        <param>
          <value><string>Colorado</string</value>
        </param>
      </params>
    </methodResponse>
  </query>
</iq>
```

The body of `methodCall` and `methodResponse` are defined through the XML-RPC standard. To check whether or not a resource supports Jabber-RPC you can use Service Discovery as described below.

JEP-30: Service Discovery Service Discovery is used discover information about a Jabber entity as well as to discover information about items associated with a Jabber entity. This is because Service Discovery was designed in order to get information about entities reachable through JIDs as well as to gather information about items not reachable through an own JID.

In order to obtain information about an entity you have to send an `<iq/>` stanza of type `get` with an empty query child element qualified by `http://jabber.org/protocol/disco#info` namespace to the target entity. You may also give a node attribute for the query element. As a reply you get an `<iq/>` stanza of type `result` which as well has the query element. The query element has one or more identity elements and one or more feature elements. The identity elements give a category and a type of the entity through a category and a type attribute. The feature element has a var attribute which gives a protocol namespace or an other feature offered by the entity [JH06].

```
<iq type='get'
  from='romeo@montague.net/orchard'
  to='plays.shakespeare.lit'
  id='info1'>
  <query xmlns='http://jabber.org/protocol/disco#info' />
</iq>
<iq type='result'
  from='plays.shakespeare.lit'
  to='romeo@montague.net/orchard'
  id='info1'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    <identity
      category='conference'
      type='text'
      name='Play-Specific Chatrooms' />
    <identity
      category='directory'
      type='chatroom'
      name='Play-Specific Chatrooms' />
    <feature var='http://jabber.org/protocol/disco#info' />
    <feature var='http://jabber.org/protocol/disco#items' />
    <feature var='http://jabber.org/protocol/muc' />
    <feature var='jabber:iq:register' />
    <feature var='jabber:iq:search' />
    <feature var='jabber:iq:time' />
    <feature var='jabber:iq:version' />
  </query>
```

</iq>

On the other hand to discover the items associated with a Jabber entity you have to do a request through an <iq/> stanza of type get with an empty query child element qualified by `http://jabber.org/protocol/disco#items` namespace addressed to the target entity. As a response you get an <iq/> stanza of type result which as well has the query element but also item children each item element with a `jid` attribute and optional a `name` attribute and a `node` attribute. If there is a `node` attribute given it may be an entry point for further nodes (e.g. a category for music ID nodes). So you can have a hierarchy of nodes. You can query this hierarchy by giving the `node` attribute with the blank query element at request time [JH06].

```
<iq type='get'
  from='romeo@montague.net/orchard'
  to='catalog.shakespeare.lit'
  id='items3'>
  <query xmlns='http://jabber.org/protocol/disco#items'
    node='music'/>
</iq>
<iq type='result'
  from='catalog.shakespeare.lit'
  to='romeo@montague.net/orchard'
  id='items3'>
  <query xmlns='http://jabber.org/protocol/disco#items'
    node='music'>
    <item jid='catalog.shakespeare.lit'
      node='music/A'/>
    <item jid='catalog.shakespeare.lit'
      node='music/B'/>
    <item jid='catalog.shakespeare.lit'
      node='music/C'/>
    <item jid='catalog.shakespeare.lit'
      node='music/D'/>
    .
    .
    .
  </query>
</iq>
```

This example shows a typical request/response communication where a node is requested and a result with further nodes is replied.

2.3.3 Non standard extension

To make it possible for users to publish documents we had to create a protocol extension which manages a publish / subscribe system on the client side. We could not use the JEP 0060 (PubSub) extension because this would have caused an involvement of the server implementation which in this case was not intended. Another point against is a still ongoing discussion about major changes since the document is still draft stage.

Our publish/subscribe system consists of three kinds of requests and responses. First of all there is the PubSubInfo part which gathers the currently published documents information of a specified client. When you have this information you can subscribe to such a remote document which is handled by the PubSubSubscribe part. The last part is the PubSubEvent part to distribute changes. The details of this system are shown in [4.2](#).

2.4 Conceptual Structures

This chapter is about the advantages of structured data.

2.4.1 Semantic Web

The term "Semantic Web" in context of the internet was first used by Tim Berners-Lee in his visionary document "Semantic Web Road map" published on the w3c homepage in 1998 [[BL98](#)]. Tim Berners-Lee is the inventor of the World Wide Web as we know it today. He is the Director of the World Wide Web Consortium. The introduction section on the official Semantic Web homepage located at w3.org describes the Semantic Web informal: "The Semantic Web is about two things. It is about common formats for interchange of data, where on the original Web we only had interchange of documents. Also it is about language for recording how the data relates to real world objects. That allows a person, or a machine, to start off in one database, and then move through an unending set of databases which are connected not by wires but by being about the same thing." [[EM06](#)] So the semantic web approach tries to put data in a standardised form like the original web did it for documents. The second thing is to link the information between data with same meaning (semantic). In order to achieve these goals "the Semantic Web approach [...] develops languages for expressing information in a machine processable form" [[BL98](#)]. These languages are the Resource Description Framework (RDF) on the one hand and the Web Ontology Language (OWL) on the other hand published as W3C Recommendations.

2.4.2 Resource Description Framework

The Resource Description Framework (RDF) is a framework for representing information on the Web. It is world-open so anyone can make statements about any resource.

RDF uses the key concepts listed in Table 2.5.

- Graph data model
- URI-based vocabulary
- Datatypes
- Literals
- XML serialization syntax
- Expression of simple facts
- Entailment

Table 2.5: Key Concepts of RDF [GK04]

So there is a RDF/XML Syntax to specify RDF graphs with a specific semantic. Similar to usual graph definitions, an RDF graph consists of

- Nodes: There are two kinds of nodes.
 - Resources: these are the main building blocks of the graph. If a node should be publicly addressable, it gets a URI as an identifier. If not, it remains anonymous and is called a *blank node*.
 - Literals: Resources can also be viewed as potentially *containing* URIs. In order to add arbitrary data to RDF, one attaches a literal to a resource. Literals can only exist in that role, as the target of an edge. They contain a text string.
- Edges: edges are binary and directed, connecting a source and a target node. Every edge has a node as its type; a label, if you will.

[Rau05] These conceptual structures are transferred into concrete syntax through the RDF/XML syntax. RDF data in XML looks like this:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
```

```

    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:te="http://hypergraphs.de/terms/"
<rdf:Description
  rdf:about="http://hypergraphs.de/mypage"
  dc:title="My Page">
  <te:publisher>
    <rdf:Description ex:fullName="My Name">
      <te:homePage
        rdf:resource="http://hypergraphs.de/mypage2" />
    </rdf:Description>
  </te:publisher>
</rdf:Description>
</rdf:RDF>

```

Conceptual this means:

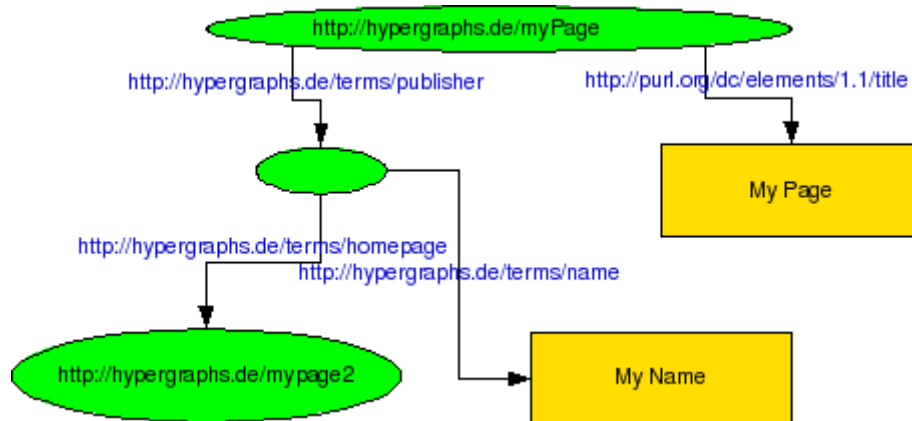


Figure 2.4: RDF Example Syntax as Graph (Conceptual): This is how a graph can be build. The rectangles are literals, the nodes are shown as ellipses.

A node is shown as an ellipse and a literal as a rectangle. The edges are named with their corresponding URIs. Internally RDF Graphs are triples as you can see in the next section.

2.4.3 RDF as Triples

Internally, RDF is stored as a set of *triples*. Each triple is an edge and has the components *subject*, *predicate* and *object*. The subject is the source of the edge, the object its target and the predicate its label. Subject and predicate must be resources, the object can either be a resource or a literal. It is interesting to note that, contrary to typical mathematical graph definitions, RDF does

not explicitly list the available nodes. A node is “brought into existence” as soon as it is either source or target of an edge. In practice, this is not a problem, because RDF is so fine-grained that single resources are rarely enough for expressing anything meaningful. Even the type of a node is defined by a triple.

2.5 Hyena or how to Handle RDF

Hyena is a platform for RDF-based software engineering. It uses a tree of RDF graphs to hold the data, each RDF graph can manage different kinds of software engineering related models within the same graph. Graph editing is handled by a graphical user interface based on the eclipse plugin architecture. Another way of editing is the possibility to do changes via a web interface. The core hyena engine consists of three parts:

- Components
- Vodules
- Web Service API

Components provide services like presentation, editing or node visualization. The Components are registered in a special registry and initialized on demand. Vodules are similar to plugin architectures. They provide a way to add support for different kind of models. They can be switched on and off and register contributions with the components. The Web Service API is used to define an interface for web editing and automatic web presentation. It is based on ReST. Hyena tries to reduce some overhead generated through the use of RDF. It avoids the use of Namespaces for example by using `rdfs:label`. This is to hide URIs which are used internally from the presentation layer.

2.6 Eclipse Plugins

First of all we decided to implement the distributed editing tool into the Eclipse plugin Hyena.

”Eclipse is an open source community whose projects are focused on providing a vendor-neutral open development platform and application frameworks for building software. The Eclipse Foundation is a not-for-profit corporation formed to advance the creation, evolution, promotion, and support of the Eclipse Platform and to cultivate both an open source community and an ecosystem of complementary products, capabilities, and services.” [Ecl06].

The Eclipse Platform is "an open source, robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools and rich client applications" [Los06]. So it is an Integrated Development Environment (IDE) but it is also a framework for standalone or cooperating products at the same time. This is gained through the Plugin structure which makes it possible to simply switch on or off the parts of the platform you need. In fact there is just a very small and simple workbench core which is used to manage plugins. Everything else is plugged into this core to create the IDE. "A plug-in in Eclipse is a component that provides a certain type of service within the context of the Eclipse workbench" [Bol]. If you want to find out more about the plugin architecture of Eclipse feel free to have a look at [Bol].

Chapter 3

Extending Hyena with Chat Functionality

First part was creating a simple and basic instant messenger for Eclipse. It should be possible to be aware of presence changes, to write single user messages and to join and create Multi User Chatrooms.

3.1 The Chat Interface

Our instant messaging client is divided into two Eclipse Views called Userlist view and Chat view. For each chat (single or multiuser) there is a separate Chat view where you can type your messages and have a look at the log. You can connect to a single Jabber server with a single ID via the Userlist view. Since it is not possible to connect with more than one account it is only one Userlist view available at a time.

3.1.1 The Userlist View

The Userlist view (or buddy list) represents the connection to an instant messaging server which supports the Jabber (XMPP) protocol. When you start the view you will not be connected. To connect to a server you use the "Connect to server..." entry from the views menu. Then a wizard will appear asking you for your login data. You have to give a username, a password, a server to connect to, the jabber server your username belongs to and the port you want to connect to. The last option is a checkbox where you can select if the connection should be encrypted using SSL/TLS. After pressing the Finish button you are connected and your buddy list (or roster as seen in [3.2.2](#)) will be shown (see figure [3.1](#)).

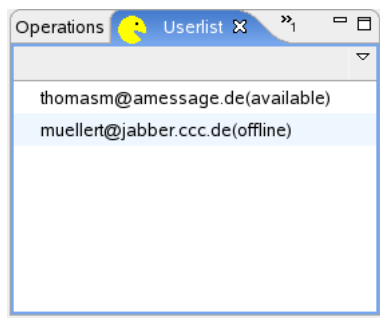


Figure 3.1: Userlist View: Here you see the Userlist View which enables you to manage your connection. The Views menu can be seen as a arrow pointing downwards on the upper right.

If you have no buddy list by now, you can add buddies by choosing Add Buddy... from the Userlist view menu. Then you just have to fill in the JID of the buddy you want to add and press the finish button. The buddy will appear in your buddy list at once. Whenever your buddy logs in the buddy will be tagged with (available). If you want to delete a buddy simply select it by pressing the right mouse button once on its JID. From the context menu select Delete Buddy and it will be removed from your buddy list.

To start a conversation with one specific user just double click on the users JID and a chat view will appear. Then you can use it as described in the next section. To close a chat it is enough to close the view. Whenever you get a message from another user it will be sent to the corresponding chat view. However if there is no chat view for this specific user one will be opened.

A Multi User Chat can be started by pressing the right mouse button on the Userlist view and choosing Start MUC ... from the context menu. Then a wizard opens where you have to specify the chatroom you want to use. If you are the first one connecting to this room it will be created and opened for everyone. However you will join the room then and others are free to join the room as well so group chatting will be possible.

3.1.2 The Chat View

A chat view is used to send plain text messages to one or more participants. The view is divided into two areas. On the upper left you can see the chat log. Whenever someone sends a message it is printed here with the JID followed by a colon and the message he or she sent. Beneath you can find the message composition box. You can type your message there and then send it by pressing the Return key. Figure 3.2 shows the view with some sample data.

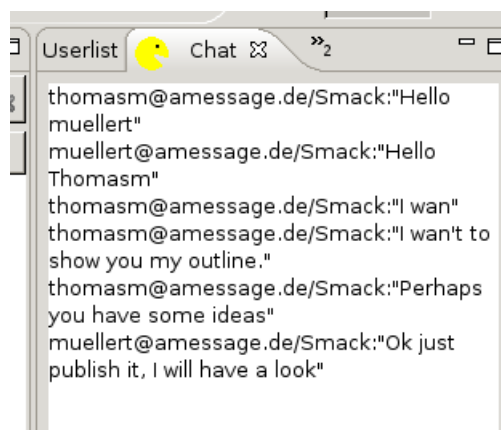


Figure 3.2: Chat View: Here you see the Chat view with a sample chat between thomasm and muellert, both registered with amessage.de

3.2 Smack Library

We choose Jabber as our protocol because it is an open standard protocol with a very active maintenance from The Jabber Foundation. Besides the other open protocol which we could have chosen was SIMPLE which as of 8.12.2005 did not meet our requirements especially the support of Multi User Chat which we wanted to have (see 2.3.2). Our primary focus was on a functional chat with a buddy list paired with presence awareness (first version without grouping) and single or multi user capability. Jabber had everything we needed including multi user chat (as a draft, but works). To get the chat functionality we introduced the smack library to Hyena. Smack is an open source API library for connecting to and using Jabber instant messaging systems.

The Key Advantages are [Sof] :

- Extremely simple to use
- Use of higher level constructs instead of packet level
- Provides easy machine to machine communication
- Automatic TLS and SSL encrypted connection
- Open Source under the Apache License (You may use it also in commercial products)

For more information on the Jabber protocol see 2.3.2.

3.2.1 Connecting to an XMPP Server

The following code creates a SSL encrypted XMPPConnection *con* to a server *host* at port *port* and logs in using a username *user* and a password *pass*. The *service* argument is the hostname of the JabberID. This is needed because the login servers hostname can possibly differ from the JIDs hostname (e.g. Google Talk: hostname to connect to is talk.google.com, hostname for JID is gmail.com or googlemail.com). For Google Talk you always need a googlemail or gmail account since their mail system and their instant messaging system work together. Smack supports Google Talk as well as other Jabber servers but no voice connection yet. However to create a connection the following code is needed.

```
XMPPConnection con = new SSLXMPPConnection(host, port, service);
con.login(user, pass);
```

If an error occurs (Timeout, Authentication, etc.) you get an XMPPException with a specific error code. You can then create an appropriate error notification message with your client. If the connection could be established you have different possibilities to go on.

3.2.2 Buddy Management

Smack has the possibility to group, add or delete buddies. We implemented just the add and delete part but in further versions grouping will be possible. The buddy management is handled by a roster object. The buddy list is saved on the server side and the roster object is filled on login. Then you can simply use the roster object to add or delete buddies. This looks somehow like this:

```
public void addBuddy(String jid) {
    if(con.getRoster().contains(jid)) return;
    try {
        con.getRoster().createEntry(jid, jid, null);
    } catch (XMPPException e) {
        handleRosterAddError(e.getXMPPError());
    }
}
```

For now it is only checked if your roster already contains the given JID. However there is no check if the buddy (JID) you add really exists with the server. In future versions this will be perhaps possible.

3.2.3 Presence Awareness

A big advantage of instant messaging systems is the presence awareness. When you log into a Jabber server your online status will be distributed so that everybody whose buddy list has an entry with your JID will be notified. To get the presence of your buddies you have to register a RosterListener with your Roster object:

```
roster.addRosterListener(new RosterListener(){
    public void presenceChanged(String jid) {
        System.out.println("User " + jid +
            " changed presence to " + roster.getPresence(jid));
    }
    ....
}
```

The RosterListener can inform about other changes to the roster like added or removed buddies as well.

3.2.4 Single User Messaging

Single User Messaging is quite simple. You create a chat object with the connection where you give the target of this chat. The target is a valid JID. It is not necessary that this target is located on the same Jabber server you use because all Jabber servers (since 17.01.2006 Google Talk as well [[Goo](#)]) are connected to each other. So sending a message from a@amessage.de to b@gmail.com is possible. To send a message you just invoke sendMessage on the chat with the message as a String argument. The following code shows how this is done.

```
Chat chat = con.createChat(target);
chat.sendMessage("Hello");
```

In order to receive messages your communication partner sends you should register a PacketListener for this chat. You add it as a MessageListener which gets notified whenever a text message is sent to you to this chat. You can then print out the message on a console like in this example or use it in a graphical user interface.

```
chat.addMessageListener(new PacketListener(){
    public void processPacket(Packet pak){
        Message message = (Message)packet;
        System.out.println(message.getFrom() +
```

```
        " says: " + message.getBody());
    }
});
```

3.2.5 Multi User Chat

A Multi User Chat works a little different as a single user one. You start with a MultiUserChat object instead of the normal Chat object. As target ID you choose a chatroom ID. A chatroom ID is composed like any other JID. Instead of the username you type in the roomname. The hostname normally has an additional subdomain for multi user chats. At amessage.info for example multi user chats are hosted at chat.amessage.info. So a chatroom ID for a multi user chat could look like myroom@chat.amessage.info.

Before you can join a room you have to check if the given chatroom ID matches one of the hosted rooms. If this is not the case you have to create the room and join it afterwards. If it already exists you can simply join it in case it is not locked.

```
MultiUserChat chat = new MultiUserChat(con, target);
for(HostedRoom hostedRoom : hostedRooms){
    if(hostedRoom.getJid().equals(target)){
        found = true; break;
    }
}
if(!found){
    chat.create(nickname);
    chat.sendConfigurationForm(new Form(Form.TYPE_SUBMIT));
}else{
    chat.join(nickname);
}
```

The sendConfigurationForm method can be used to define some access restrictions for the room but in our case it just creates an open room since it should be a simple client. Nevertheless this is something for future versions since you can add some more security with it. Sending and receiving works similar as with single user messages (see 3.2.4). In order to receive messages you just register a PacketListener with the chat. To send messages you call the sendMessage method with the message to send as a string argument. But there is a difference between both because you as a sender will receive the message you sent as well as every other person in the virtual room. This is important since it can lead to problems in distributed editing as we will discuss below.

Chapter 4

Extending Chat with Document Management

The first idea was to create a chat client as Eclipse plugin on the one hand in order to discuss about a document and to have on the other hand a completely separated distributed document editing tool to actually change collaborative the document. But after the first prototype and a closer look at the Jabber protocol it was pretty clear that there is a possibility to combine both use cases. For us there were no need for direct editing. In fact changing RDF Nodes through direct editing would change the RDF Graph behind with every single keystroke resulting in a big overhead for nonsense data. Nevertheless as you can read in chapter 6 about future research there are some ideas to combine both approaches.

Jabber is an easily extendible protocol with an integrated extension system based on xml. So in order to make documents available we decided to implement a protocol similar to the PubSub approach (JEP 0060) but without using a Jabber server to host the document. It uses <iq> statements to request the published document IDs of an entity as well as for the change transmissions.

4.1 Extension to Jabber

Extensions to Jabber are made through an extension system. You can simply use additional XML namespaces to extend the existing elements defined in the core namespaces (see RFC3921-2.4). Jabber can send any extension (as xml stream) with every message. Smack provides a way to write your own extensions and make them discoverable if you wish so. You just have to tell the extension how to become part of the xml stream and vice versa. This mechanism seemed to work for us in the first place but was replaced by the iq stanza mechanism hence we decided to make the document management

independent from single or multi user chats. Figure 4.1 shows the concept of our publish / subscribe system. The following preconditions must hold:

- Publisher and subscriber are connected to their Jabber servers.
- The publisher marked his document to be published.

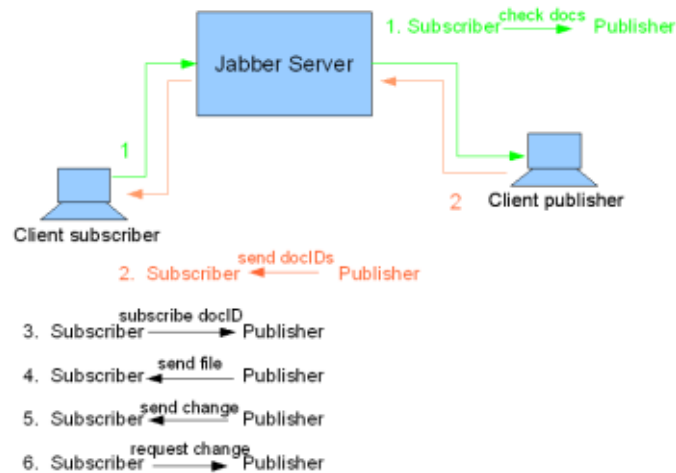


Figure 4.1: Own Jabber Extensions: For explanation please have a look at the text.

First thing happening is that the subscriber checks for published documents at the publisher (1.). As a reply he gets a list of document IDs (2.). These are the published documents. The subscriber selects one and subscribes for it (3.). As a reply he gets the whole document (4.). From that point every change on the publisher will cause a notification of the subscriber (5.). The subscriber himself can do changes which leads to a change request at the publisher client (6.). If the change is allowed it will be performed and the subscriber will be notified as before (5.). Otherwise the change will be discarded. If there is no notification the change will be discarded at the subscriber side as well.

4.2 The Protocol Inside

4.2.1 Remote Document Management

In order to get access to a remote document your peer has to make it available by putting it into published state. This is done by marking it in the Local Documents view. Thereafter you will be able to use the check documents button in the context menu for the jabber ID of your peer. Your client then sends a DocumentPubSubInfo request like this:

```
<iq id="D1GTn-4" to="muellert@amessage.de/Smack"
  from="thomasm@amessage.de/Smack" type="get">
  <query xmlns='http://hypergraphs.de/peerstorm/pubsub#info'>
    </query>
</iq>
```

and your peers client will answer as the result with a complete list of all his published documents:

```
<iq id="D1GTn-4" to="thomasm@amessage.de/Smack"
  from="muellert@amessage.de/Smack" type="result">
  <query xmlns='http://hypergraphs.de/peerstorm/pubsub#info'>
    <documents>
      <documentID>
        muellert@amessage.de/Smack#wikked-example.rdf
      </documentID>
    </documents>
  </query>
</iq>
```

The message ID is used to match query and result so it is not changed. The type of a query is always get as well as the result type is always result.

4.2.2 Remote Document Subscription

Once you can see the documents of a peer in your Chat Documents view (how to do this you see in 4.2.1) you can select one by marking it for distributed editing. This is when your client sends a DocumentPubSubSubscribe request:

```
<iq id="D1GTn-5" to="muellert@amessage.de/Smack"
  from="thomasm@amessage.de/Smack" type="set">
  <query xmlns='http://hypergraphs.de/peerstorm/pubsub#subscribe'
    documentID='muellert@amessage.de/Smack#wikked-example.rdf'>
  </query>
</iq>
```

As a result you get the whole file in the response:

```
<iq id="D1GTn-5" to="thomasm@amessage.de/Smack"
  from="muellert@amessage.de/Smack" type="result">
  <query xmlns='http://hypergraphs.de/peerstorm/pubsub#subscribe'
    documentID='muellert@amessage.de/Smack#wikked-example.rdf'>
    <file>
```

```
<wholedocument>
  <subGraph
    id='http://hypergraphs.de/hyena#SettingsSubGraph' >
    ...
  </file>
</query>
</iq>
```

4.2.3 Remote Document Synchronisation

Each change on the remote document will be done by the publishing client. This means if you are just the subscriber the changes you make will be sent to the publisher. There they are checked and possibly (mostly) performed. Then all the registered subscribers get a notification of the change. When you are the publisher your changes are performed directly and sent to all the subscribers. This is all handled by the DocumentPubSubEvent. An example for a request of such a change could look like this:

```
<iq id="300c81b5_10b6f0ca61c_-7fcd"
  to="muellert@amessage.de/Smack"
  from="thomasm@amessage.de/Smack" type="set">
  <query xmlns='http://hypergraphs.de/peerstorm/pubsub#event'
    documentID='muellert@amessage.de/Smack#wikked-example.rdf'>
    <change>
      <removetriplechange
        graphid='http://hypergraphs.de/hyena#MainSubGraph' >
        <triple>
          <ressource>http://page2.com</ressource>
          <ressource>
            http://hypergraphs.de/wikked/rdf.html#title
          </ressource>
          <literal>Page Two</literal>
        </triple>
      </removetriplechange>
    </change>
  </query>
</iq>
```

The publisher would do the change and reply with the following message

```
<iq id="300c81b5_10b6f0ca61c_-7fcd"
  to="thomasm@amessage.de/Smack"
```

```
    from="muellert@amessage.de/Smack" type="result">
<query xmlns='http://hypergraphs.de/peerstorm/pubsub#event'
  documentID='muellert@amessage.de/Smack#wikked-example.rdf'>
  <change>
    &lt;removetriplechange
      graphid='http://hypergraphs.de/hyena#MainSubGraph'&gt;
      &lt;triple&gt;
        &lt;ressource&gt;http://page2.com&lt;/ressource&gt;
        &lt;ressource&gt;
          http://hypergraphs.de/wikked/rdf.html#title
        &lt;/ressource&gt;
        &lt;literal&gt;Page Two&lt;/literal&gt;
      &lt;/triple&gt;
    &lt;/removetriplechange&gt;
  </change>
</query>
</iq>
```

All the other subscribers would simply be notified through a similar message with a set type and reply with a simple "ok" in the query item.

Chapter 5

Examples

First of all you have to make sure Hyena and PeerStorm are installed. Then Hyena should be started. By selecting the PeerStorm Perspective you get additional views on the screen. On the right you find the Userlist View which in other products is sometimes called buddy list. At the moment it should show the "Not connected" message. On the upper right of the view you can see a small arrow pointing downwards. This is the menu arrow for the view. Each view with a menu shows this arrow. If you press it you get a list of menu items. One is labeled "Connect". When you use it a New Connection Wizard appears asking you for username, password and hostname to connect to. The port should "normally" be one of 5222 or 5223. The latter is used for SSL Connections. In offence Google for example uses port 443 as well for SSL connections to their GoggleTalk service. Once you are connected your screen should look similar to the one in figure 5.1.

5.1 Example: Distributed Wiki RDF Editing

To start a distributed session for editing a RDF document you need a minimum of two clients using Hyena and PeerStorm. One client needs an open RDF file which both want to edit. The Local Documents view gives this one the possibility to publish his document so that others can subscribe to it. When the document is selected in the Local Documents view it instantly becomes available. After that, the second one needs to get the information about the document. This is done by pressing the right mouse button on the entry of the publisher to see the context menu (see figure 5.2). There you can select "Check Documents".

Right after you should see a new entry in the Remote Documents view. By selecting the little square in front of the documents ID you can subscribe for it. A new file wizard will appear, where you can create a new file for the

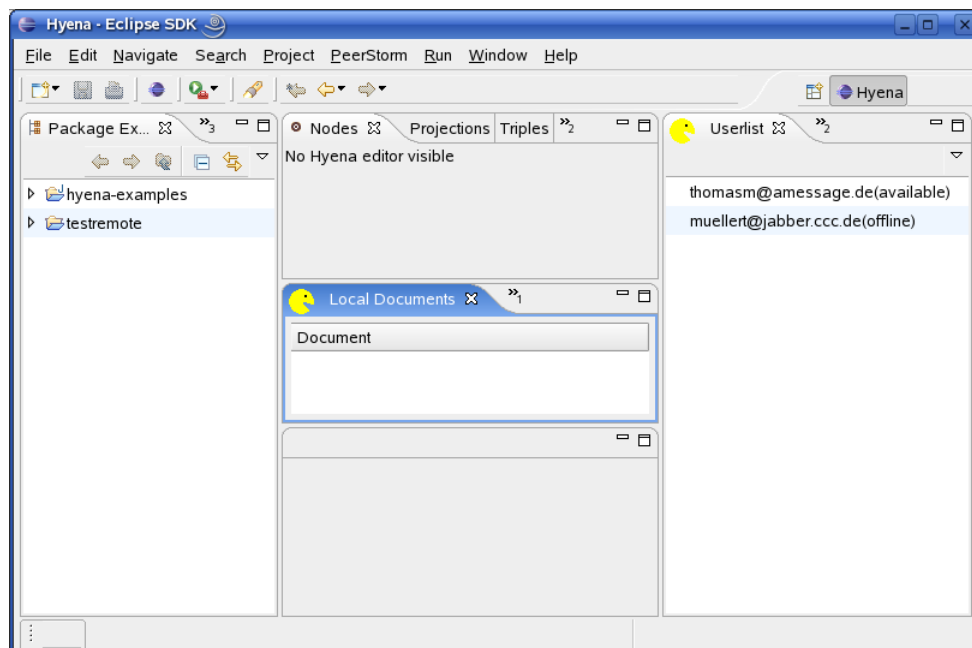


Figure 5.1: An Overview of Hyena with activated PeerStorm: Here you see Hyena with activated and connected PeerStorm. The important PeerStorm views are the Userlist view in the right and the Local Documents view as well as the Chat Documents view in the middle. Here the Userlist shows an available buddy called thomasm and an offline buddy called muellert. Currently there are no open documents so the Local Documents view is empty.

remote document(see figure 5.3).

After finishing that wizard the whole document will be transferred onto the second client and all the events from this time on will also be sent to the subscribed client. From then every change the subscribed client does will be stopped and transmitted to the publishing client which does the change and notifies all subscribed clients of the change. So if you are the subscriber you can simply click a node in the Nodes view. The editor will show the nodes details. Double click on the entry you want to change and do your changes. As soon as you confirm the change the publisher will be requested for this change and perform it if possible. Your client will be notified when the change is performed.

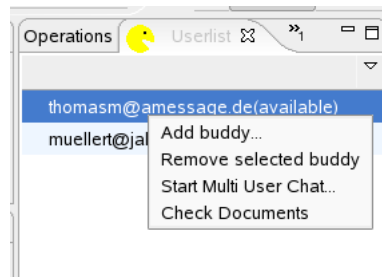


Figure 5.2: Context menu of a buddy: This figure shows a context menu of a buddy. You can add a buddy, remove the selected buddy from your list, start a multi user chat or check if the buddy has published documents.

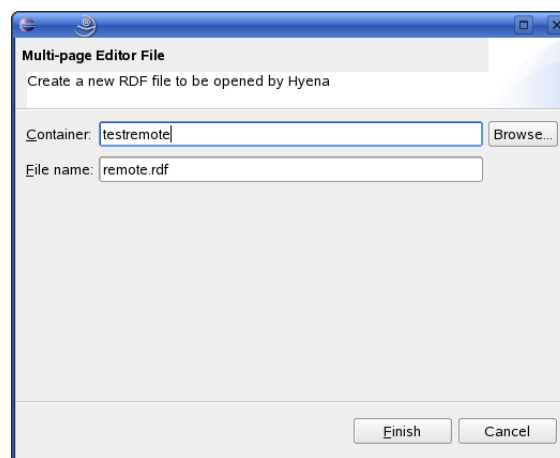


Figure 5.3: The remote file wizard: The remote file wizard lets you define which project should contain the data and how the file should be called locally.

5.2 Use Case: Distributed Outline Editing

Distributed editing works pretty similar to direct RDF editing. You need a minimum of two clients as well. One should open the Outline RDF and make it available by pressing the publish square in the Local Documents view. The other can then subscribe to the file. In the Outline Editor you can do the following editing:

- Create new entry: ctrl-return (Mac: command-return)
- Indent entry: tab
- Outdent entry: shift-tab
- Edit entry: return

- Delete entry: del

These editing commands and their results are transmitted as well. It is almost transparent for users who work with each other except the fact that you have to publish and subscribe. Working feels almost the same. So one starts with adding entries called Introduction, Implementation, Conclusion and while he is adding Future Research to finish the body, his coworker adds already the subentries for Implementation because this will be his part. He then does the indentation for his part. This is what they talked about before by using the Chat module of PeerStorm. A very interesting thing for writing papers because you see actually what your coauthor is doing and you can adapt perhaps your own words if the style is somehow different. Another advantage over writing parts separate and merge them later is that you can talk about not solved issues instantly when they appear via the chat environment while editing. In the end the result may look like shown in figure 5.4.

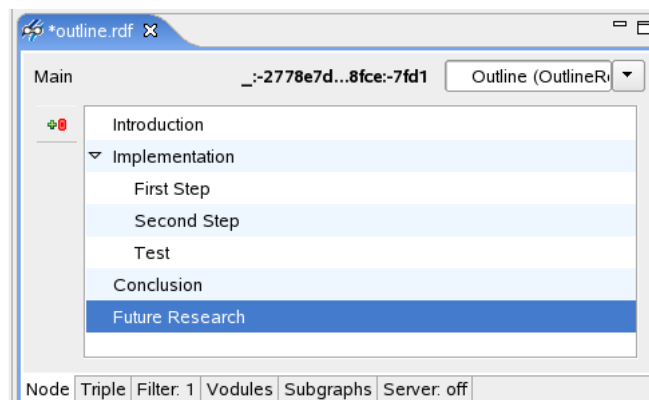


Figure 5.4: Result of the outline example: Here you can see the result how your document could look like when you do the example.

Chapter 6

Future Research

Interesting parts appeared while working on the product which could not be covered and need future research. For the integration part it was not intended to do direct editing for literals. But perhaps this could be implemented in further researches. The document management is very basic and could be improved. Some secure authentication for document subscription would be very nice to have. Security, except the secure communication itself, has not been a part of this work and could be very useful. Since Google Talk is as well Jabber based there is a slight possibility that future releases also cover Voice over IP which would be very interesting. Most people prefer to talk instead of using a chat especially if they are working. This product is for now complete but there are interesting aspects not covered yet. A very interesting point can be the Jingle JEP, a peer to peer extension which could drastically increase performance since you just need the server connection to find a peer and in firewall environments for whole punching. After that you can do a direct connection between the clients which would surely outperform the detour over the server and which could increase security since you can be sure to do communication just with the IP Adresses you want to as well as preventing the server from logging your communication in any way. Another fine feature for the future would be to make every client a backup server. Since all the clients hold the whole document in their memory, if the server fails for whatever reason it should be possible to set the server flag in another client and just go on as if there would have been no failure.

As you can see the product has potential to be an ultimate weapon for collaborative work in software development and other parts of document creation. Hyena is really good for single user editing and with PeerStorm we will increase productivity and teamwork for all software engineering models added to Hyena.

List of Figures

2.1	Client Server Schema	7
2.2	Peer To Peer Schema	9
2.3	MSN Messenger Example Connection	13
2.4	RDF Example Syntax as Graph	23
3.1	Userlist View	27
3.2	Chat View	28
4.1	Own Jabber Extensions	33
5.1	An Overview of Hyena with activated PeerStorm	38
5.2	Context menu of a buddy	39
5.3	Remote file wizard	39
5.4	Result of the outline example	40

List of Tables

2.1	Significant Problems with Client / Server Architecture	7
2.2	Possible values for the type attribute of message stanzas	16
2.3	Possible values for the type attribute of presence stanzas	17
2.4	Possible values of the show element	17
2.5	Key Concepts of RDF [GK04]	22

Bibliography

- [Ada06] D. Adams. *JEP-0009: Jabber-RPC*. <http://www.jabber.org/jeps/jep-0009.html>, February 2006.
- [AR] W. C. Kammergruber A. Rauschmayer. *A Wiki as an Extensible RDF Presentation Engine*. To be published. http://wiki.ontoworld.org/index.php/A_Wiki_as_an_Extensible_RDF_Presentation_Engine.
- [BL98] T. Berners-Lee. *Semantic Web Road map*. <http://www.w3.org/DesignIssues/Semantic.html>, September 1998.
- [Bol] A. Bolour. *Notes on the Eclipse Plug-in Architecture*. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.
- [Cla04] R. Clarke. *Peer-to-Peer (P2P) - An Overview*. <http://www.anu.edu.au/people/Roger.Clarke/EC/P2POview.html>, November 2004.
- [Ecl06] Eclipse. *Eclipse.org : About Us*. <http://www.eclipse.org/org/>, May 2006.
- [EM06] D. Brickley B. McBride-J. Hendler G. Schreiber D. Wood D. Connolly E. Miller, R. Swick. *Semantic Web*. <http://www.w3.org/2001/sw/>, January 2006.
- [GK04] B. McBride G. Klyne, J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. <http://www.w3.org/TR/rdf-concepts/>, February 2004.
- [Goo] Google. *Google Talk Federation*. http://www.google.com/intl/en/press/annc/googletalk_federation.html.
- [JH06] R. Eatmon P. Saint-Andre J. Hildebrand, P. Millard. *JEP-0030: Service Discovery*. <http://www.jabber.org/jeps/jep-0030.html>, January 2006.

- [Los06] P. Loshin. *The Value of Eclipse for the Open Source Community*. <http://www.b-eye-network.com/view/2334/>, February 2006.
- [MM] A. Sawyers M. Mintz. *MSN Messenger Protocol*. <http://www.hypothetic.org/docs/msn/index.php>.
- [Rau05] Axel Rauschmayer. *A Short Introduction to RDF for Software Engineers*, 2005.
- [SA04a] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. <http://www.ietf.org/rfc/rfc3921.txt>, October 2004. Network Working Group.
- [SA04b] P. Saint-Andre. *End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)*. <http://www.ietf.org/rfc/rfc3923.txt>, October 2004. Network Working Group.
- [SA04c] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core*. <http://www.ietf.org/rfc/rfc3920.txt>, October 2004. Network Working Group.
- [SA04d] P. Saint-Andre. *Mapping the Extensible Messaging and Presence Protocol (XMPP) to Common Presence and Instant Messaging (CPIM)*. <http://www.ietf.org/rfc/rfc3922.txt>, October 2004. Network Working Group.
- [Sof] Jive Software. *Smack Overview*. <http://www.jivesoftware.org/builds/smack/docs/latest/documentation/overview.html>.
- [Wik06a] Wikipedia. *client-server*. <http://en.wikipedia.org/wiki/client-server>, May 2006.
- [Wik06b] Wikipedia. *Instant Messaging*. http://en.wikipedia.org/wiki/instant_messaging, March 2006.
- [Wik06c] Wikipedia. *Wiki*. <http://en.wikipedia.org/wiki/wiki>, May 2006.
- [Win03] D. Winer. *XML-RPC Specification*. <http://www.xmlrpc.com/spec>, June 2003.