Nabu: A Semantic Archive for XMPP Instant Messaging

Frank Osterfeld

Projektarbeit

Technische Universität Kaiserslautern
Fachbereich Informatik
AG Wissensbasierte Systeme
Prof. Dr. Andreas Dengel

August 2005

Betreuung:

Prof. Dr. Andreas Dengel
Dipl. Inform. Malte Kiesel, DFKI GmbH
Dipl. Inform. Sven Schwarz, DFKI GmbH

Contents

1 Introduction **4**

2 Handbook **6**

2.1 Installation 6
2.2 First Steps 8
2.3 Search the Archive 9
2.4 The Nabu Ontology 11
2.5 Annotations 14
2.6 Log Sharing and Privacy Settings 16
2.7 User Observation 18
2.8 The Nabu Protocol 20

3 Development **27**

3.1 Design 27
3.2 Internal RDF Schema 28
3.3 URI Schema 30
3.4 Implementation 32
3.5 The Client API 38

4 Conclusion **42**

Bibliography **43**

A Response Codes **45**

Chapter 1

Introduction

The importance of Instant Messaging (IM) for private and organizational communication increased over the last years. IM, the instant sending and receiving of (mostly short) text messages between two or more users, became one of the most used communication channels on the internet and more and more valuable information is exchanged via instant messages. In contrast to the increasing amount of information exchanged, the IM implementations lack support to archive the information and retrieve it later when needed. Instant messaging is still transient and information gets lost. While E-Mail can be archived longterm server-side using the IMAP standard, there is no standard for archiving IM conversations. Chat logs are mostly stored locally on the client machine, using IM-client-specific file formats. This has several disadvantages: Storing the archive locally on the client computers is inconvenient when using more than one computer, archives are spread over different installations, are out of sync and get easily lost. Using proprietary, client- and protocol-specific formats to store the information makes it hard to manage and search the stored information using other interfaces than the client UI.

This project thesis addresses these issues by developing Nabu[1], a system providing server-side logging of instant messages. Nabu is implemented for the XML-based Jabber/XMPP protocol[2]. Unlike other proprietary IM protocols from major providers such as Yahoo!, MSN or AOL, Jabber/XMPP is an open standard. Most server and client software is available under open source licenses, which makes it possible to add Nabu's features as a plugin for an existing server implementation. Jive Messenger[3] was chosen due to its well-designed and well-documented code base and easy extensibility.

Nabu tries to integrate instant messaging into the efforts made in the Semantic Web [1] community to store and retrieve information in a unified way. It uses the Semantic Web standard RDF[4] to describe the stored information on the

[1] http://nabu.opendfki.de/

[2] http://www.jabber.org

[3] http://www.jivesoftware.org/messenger/

[4] http://www.w3.org/RDF/

server. For retrieving the stored information, it supports the SPARQL[5] query language, which is currently in an ongoing standardization process at the W3C.

Besides the logging of chat messages, further features of Nabu are:

- Users can add further metadata to the logged messages by adding their own RDF statements. That way information can be categorized and structured, making retrieval of relevant information easier.
- Nabu supports sharing of logged messages between users, to make e.g. a conference log available to other group members. Privacy is ensured by a strict privacy model, restricting access to explicitely authorized users.
- Nabu integrates instant messaging into the context elicitation framework of EPOS [6], sending message notifications to the EPOS user observation (when enabled by the user).

Related Work

Related to Nabu, a storage format and protocol for server-side message archives [3] was proposed as Jabber protocol enhancement. The proposal suggests a simple protocol and storage format for message archiving. While being simple, it defines its own format and does not use existing standards. In my opinion using existing technology as RDF and SPARQL makes the our approach more powerful without having to reinvent the wheel. Also, Nabu can be more easily integrated with other information sources, and offers additional features as log sharing and adding custom metadata.

The *Haystack project*[6] builds a client for information management by integrating various information sources in one frontend, using an infrastructure based on RDF. A messaging model was developed [4] to represent conversations from various communication channels, such as e-mail, news groups and instant messaging, in a unified way. In contrast to Nabu, Haystack is a client-based solution.

The *BuddySpace* [2] research project extends the presence concept in Jabber (simple offline/online/busy states) and adds information such as geographical location, current work focus etc. to the presence state. Furthermore it investigates how such additional semantics can be used to facilitate collaboration over networks. The BuddySpace Jabber client[7] demonstrates the concepts.

The following chapters discuss Nabu's functionality, concepts and implementation in detail. The Handbook chapter (p. 6) describes Nabu's features and is targeted towards users. Design and implementation are discussed in the Development chapter (p. 27). It also explains how to make use of Nabu by integrating it into software. Being an open-source project, this documentation is created and maintained online on the Nabu homepage.

[5] http://www.w3.org/TR/rdf-sparql-query/
[6] http://haystack.lcs.mit.edu/overview1.html
[7] http://buddyspace.sf.net/

Chapter 2

Handbook

2.1 Installation

2.1.1 Install the Nabu plugin

To use Nabu, you need Jive Messenger >= 2.1.3. You can download it on the Jive Messenger download page[1].

- If you haven't done so yet, install and setup Jive Messenger (see Jive Messenger documentation[2]).
- Download the Nabu plugin from the Nabu homepage
- Copy nabu.jar to the Jive plugin directory (jive_messenger/plugins).

2.1.2 Configure Nabu

To configure Nabu, go to the admin console, and choose *Nabu Configuration* in the *Server* tab. Here you can edit the Nabu configuration options. When finished, choose *Save Properties*. Note: At the moment, changes won't take effect before server restart.

Database Options

The options DB User, DB Password, DB URL, DB Type and DB Class are the database settings used for the underlying RDF store. Nabu passes them directly to Jena. For more detailed information how to setup the database and configure these settings, see the Jena documentation[3].

Example:

[1] http://www.jivesoftware.org/downloads.jsp
[2] http://www.jivesoftware.org/builds/messenger/docs/latest/documentation/
[3] http://jena.sourceforge.net/documentation.html

Your database user used for Nabu is 'nabuadmin' with password 'foobar'. The DB server is running on localhost:5432 and the database is 'nabustore':

When using PostgreSQL:

```
DB User: 'nabuadmin'
DB Password: 'foobar'
DB URL: 'jdbc:postgresql://localhost:5432/nabustore'
DB Type: 'PostgreSQL'
DB Class: 'org.postgresql.Driver'
```

When using MySQL:

```
DB User: 'nabuadmin'
DB Password: 'foobar'
DB URL: 'jdbc:mysql://localhost:5432/nabustore'
DB Type: 'MySQL'
DB Class: 'com.mysql.jdbc.Driver'
```

name of model

The name/identifier of the main model where conversation logs, account information etc. will be stored. You can choose any name here, usually there is no need to change this though, and the default "defaultModel" should do. If you change this, make sure that main model and internal model have different names.

name of internal model

The name of the internal model where Nabu stores internal server settings and privacy policies. As for the normal model, you can stick to the default here.

namespace for instances

The RDF namespace Nabu will use for stored messages, accounts etc. If the namespace is e.g. http://foo#, the account alice@jabber.foo.org will have a URI like "http://foo#Accounts/jabber.foo.org/alice". To guarantee globally unique URIs and to avoid collisions, you should use your own domain as prefix. If your server is jabber.foo.org, you might choose "http://jabber.foo.org/nabustore#" as instance namespace.

admin accounts

Add accounts here that should have special admin permissions for Nabu. These accounts have full read access to the Nabu archive, including the internal model. Privacy policies are ignored. Use with care. Examples: "admin@jabber.foo.org", "admin@jabber.foo.org;bob@jabber.foo.org"

2.2 First Steps

To use Nabu, a few steps are necessary to setup Nabu and activate logging. By default, logging is disabled completely, so Nabu does not log information ”behind the back” of your users. To activate Nabu, every user who wants to use Nabu must take the following steps:

2.2.1 Add the Nabu bot to your roster

To enable logging and to access the Nabu archive, the first thing to do is to add the Nabu Bot to your roster. Add nabubot@nabu.yourhost (where yourhost is the jabber server Nabu is installed on) to your roster, like you add every other contact. After mutual authorization the bot should appear online.

The Nabu bot is the interface between you and the Nabu archive: You can e.g. control logging, query the archive and manage your privacy settings by sending requests to the Bot (read here (p. 20) for a complete list).

2.2.2 Enable logging

For privacy reasons, message and presence logging is disabled by default. Nabu does not log anything sent by you without being explicitely told to do so. (This does not prevent that other users log messages you receive from them though).

Message logging

If you want Nabu to log your messages, you must enable it by sending a command to the bot:

```
LOGMESSAGES ON
```

Nabu should return

```
252 message logging enabled
```

You can check whether logging is enabled or not via

```
LOGMESSAGES STATUS
```

To turn logging off again, send

```
LOGMESSAGES OFF
```

Presence logging

Similar, you can control whether Nabu should log your presence changes (online/offline status, Away, Free for chat etc.).

Enable presence logging using

```
LOGPRESENCE ON
```

As you might have guessed, you can turn it off again via

```
LOGPRESENCE OFF
```

and check the status of presence logging by sending

```
LOGPRESENCE STATUS
```

After activating message logging, all the messages sent by you are logged as RDF (By default, logged messages can only be read by you and the receivers of the messages) To learn about searching the archive and other Nabu features read the following sections.

2.3 Search the Archive

Once Nabu logs your conversations, you probably want to search them at some point (otherwise archiving would not make that much sense). For querying the archive, Nabu uses the SPARQL query language[4]. SPARQL is a language for querying RDF stores, similar to SQL. Being powerful and versatile, it allows arbitrary complex queries (Unfortunately it's also quite complex for everyday use, so a GUI for the most common queries would be desirable).

Example: You want to search for all messages containing "Nabu". use the following command (it's splitted over multiple lines for readability, but must be sent in one message):

```
QUERY SPARQL
PREFIX nabu: <http://www.opendfki.de/ont/nabu#>
DESCRIBE ?msg
WHERE { ?msg nabu:body ?body . FILTER REGEX(?body, "Nabu", "i") }
```

('QUERY SPARQL' is the Nabu command used to send SPARQL queries. 'PREFIX nabu: ... REGEX(?body, "Nabu", "i")' is the actual query in SPARQL format). The query returns all messages ?msg that have a body ?body matching the regular expression "Nabu" ("i" makes the search case-insensitive). A bit too complex for a simple string search? Well, for this specific case, searching a string in message bodies, a shortcut exists:

[4] http://www.w3.org/TR/rdf-sparql-query/

```
QUERY SEARCHMSG Nabu
```

executes the same query with much less typing. This is the only shortcut though; for anything more complex SPARQL is the way to go.

Nabu will return the messages matching the query as RDF/XML. For instance, Nabu might return one message, containing ”Me thinks, Nabu rocks big time”:

```
210 <rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:nabu="http://www.opendfki.de/ont/nabu#">
  <nabu:Message rdf:about=
  "http://foo#instances/Messages/kafka/frank2/kafka/frank/20050714-094210.520">

   <nabu:body>Me thinks, Nabu rocks big time!</nabu:body>

   <nabu:previousMessageInRoom rdf:resource=
       "http://foo#instances/Messages/kafka/frank2/kafka/frank/20050710-143802.712"/>
   <nabu:inRoom rdf:resource=
       "http://foo#instances/P2PRooms/kafka/frank/kafka/frank2/"/>
   <nabu:subject></nabu:subject>
   <nabu:messageType>chat</nabu:messageType>
   <nabu:streamID></nabu:streamID>
   <nabu:sender rdf:resource=
       "http://foo#instances/Accounts/Jabber/kafka/frank2"/>
   <nabu:receivers rdf:resource=
       "http://foo#instances/Accounts/Jabber/kafka/frank"/>
   <nabu:datetime>2005-07-14T09:42:11.054+0200</nabu:datetime>
 </nabu:Message>
</rdf:RDF>
```

Note: **Nabu returns only RDF data that is accessible by you**. By default, this includes all messages you have sent yourself or received from others. Normally you won’t see messages exchanged between other users. If a user decides so, he can grant you access to a conversation log (for example, co-workers might decide to share the log of an online meeting with other team members). Read in the ’Log Sharing and Privacy Settings’ section (p. 16) for more detailed information on privacy settings.

Another example:

Return a list of all messages sent by schwarz@km-intern.opendfki.der since the beginning of the year:

```
QUERY SPARQL
PREFIX nabu: <http://www.opendfki.de/ont/nabu#> SELECT ?msg ?date  WHERE { ?msg
nabu:datetime ?date . ?msg nabu:sender
<http://km.dfki.de/xmpp/nabu#instances/Accounts/Jabber/km-intern.opendfki.de/schwarz>.
FILTER (xsd:dateTime(?date) >= xsd:dateTime("2005-01-01"))
```

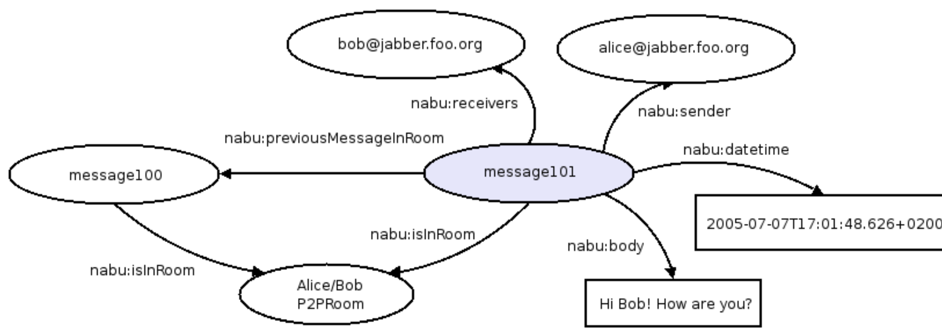
Read about the Nabu ontology (p. 11) to learn more about the data structures Nabu uses for logging.

2.4 The Nabu Ontology

This section explains the most important parts of the ontology, i.e. the RDF schema Nabu uses for logging. It shows you how messages, accounts and presence changes are represented in RDF. This covers only the most important classes and properties, you can find a complete description including the RDF Schema on the Nabu reference page[5].

2.4.1 Message

The most important class in the Nabu ontology is *Message*:



nabu:body is a literal containing the message text.

nabu:datetime is the time stamp added by Nabu when logging the message. The format is xsd:datetime[6] (e.g. "2005-07-13T11:51:34.781+0200" for "July 13th 2005, 11:51:34.781 CET"). Note that this is the time when the message was logged by Nabu, and not necessarily the message was sent or received. If sender and receiver are on different servers and every server has Nabu installed, the time stamp will be different. So identical messages cannot be matched using the timestamp.

nabu:sender links to the account that sent the message.

nabu:receivers: The accounts that received the message. In an one-to-one chat, this is a single account, the chat partner. In multi-user-chat, these are all accounts that received the message, i.e. the accounts that were in the MUC room when the message was sent. Note that the temporary nick names users have in a MUC room are ignored; anonymous MUC rooms are not supported. Nick names are resolved to the corresponding accounts. Also note that resources are omitted in both one-to-one and MUC logs.

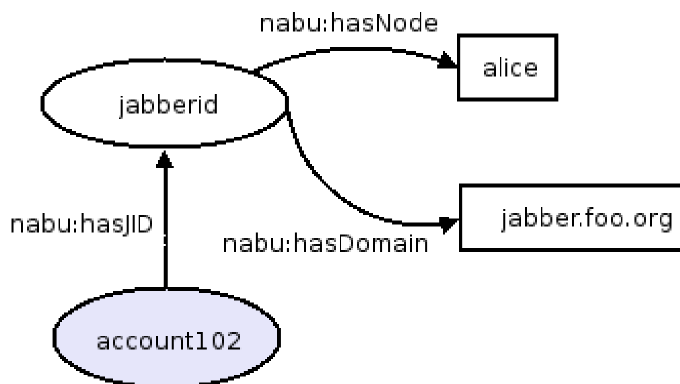
nabu:inRoom: The room the message was sent in. For details how rooms are defined see below.

[5] http://nabu.opendfki.de/cgi-bin/trac.cgi/wiki/Reference
[6] http://www.w3.org/TR/xmlschema-2/datatypes.html#dateTime

nabu:previousMessageInRoom: Links to the previous message in the room. Useful for tracking conversations and for exploring logged conversations with a specific chat partner over time.

2.4.2 Accounts

Account represents a user account. Every user account has a *Jabber ID*, like "alice@jabber.foo.org". Nabu does not store more information like person<->account associations, for privacy reasons. If such a mapping is wanted, one can store such information externally, e.g. using FOAF[7].



Every account has a Jabber ID representing the account. Not every Jabber account represents an account though (see e.g. MUC rooms), so Jabber ID and account are not identical.

2.4.3 Rooms

A room is a virtual place where two or more users meet and chat with each other. Every message has one room associated, and messages in a room are linked to make conversation tracking easier. Two types of rooms exist, depending on the chat type:

In one-to-one chats, the room is defined by the two persons chatting: If Alice chats with Bob, all messages sent by Alice to Bob and vice-versa are in "Alice-Bob-Room". The *nabu:previousMessageInRoom* property links all messages sent between Alice and Bob, making it easy for Alice to navigate through all logged messages she sent to or received from Bob. In the Nabu ontology, this kind of room is called *P2PRoom* (*Point-to-Point-Room*). In RDF, the "Alice-Bob-Room" might look like this:

[7] http://www.foaf-project.org/

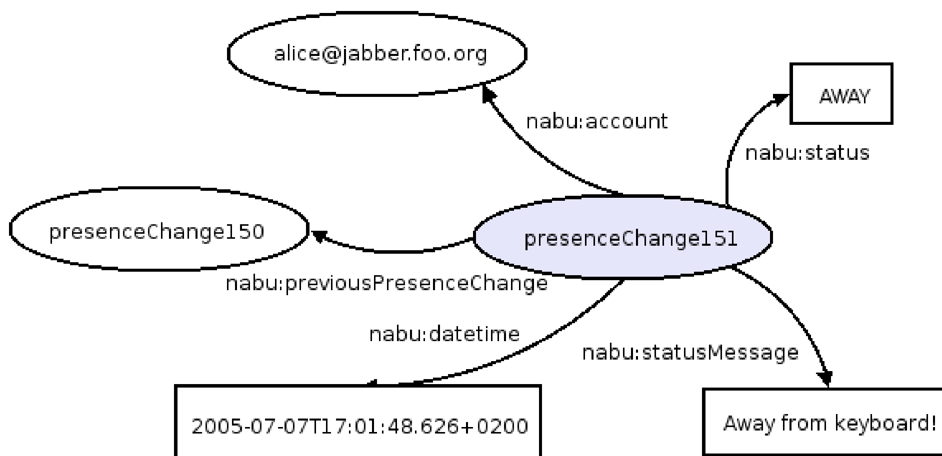
```
<nabu:P2PRoom
rdf:about="http://foo#instances/P2PRooms/jabber.foo.org/alice/jabber.foo.org/bob/">
    <nabu:members
rdf:resource="http://foo#instances/Accounts/Jabber/jabber.foo.org/alice"/>
    <nabu:members
rdf:resource="http://foo#instances/Accounts/Jabber/jabber.foo.org/bob"/>
</nabu:P2PRoom>
```

In multi-user chat, the semantics of a room are slightly different: While the P2PRoom is an addition by Nabu and does not exist in Jabber, the MUC protocol as defined in JEP-0045 [5] knows rooms: A room is e.g. support@conference.jabber.foo.org, a virtual place users can join. Unlike P2PRooms, these "MUCRooms" are not defined by their members, but by the room name and the room topic. A MUCRoom has a Jabber ID, just like accounts:

```
<nabu:MUCRoom rdf:about="http://foo#instances/MUCRooms/conference.kafka/foobar">
<nabu:hasJID>
   <nabu:JabberID
rdf:about="http://foo#instances/JabberIDs/conference.kafka/foobar">
      <nabu:hasResource></nabu:hasResource>
      <nabu:hasNode>support</nabu:hasNode>
      <rdfs:label>support@conference.jabber.foo.org</rdfs:label>
      <nabu:hasDomain>conference.jabber.foo.org</nabu:hasDomain>
   </nabu:JabberID>
</nabu:hasJID>
</nabu:MUCRoom>
```

2.4.4 Presence Change

If presence logging is enabled, every presence change, e.g. from "Offline" to "Online", or from "Online" to "Away", is stored in a PresenceChange instance:



nabu:status: the new presence status, one of online, offline, away, xa, dnd

nabu:statusMessage: The status message the user set

nabu:account: The account that changed its presence status

nabu:previousPresenceChange: The last logged presence change of the account *nabu:account*. All logged presences of a user are chronologically linked via the *nabu:previousPresenceChange* property.

2.5 Annotations

Nabu enables users to add their own statements to the RDF store. This makes it possible for users to add metadata to logged messages and share this metadata with their peers. For example, a user could set up a set of categories or tags to file his conversations to facilitate later searching. He could do this manually or even use a text classifier and categorize them automatically.

The user can add any statement he likes (except for statements from the Nabu schema, see below), but it is a good practice to reuse commonly used ontologies. Widespread vocabularies for metadata and categorization are e.g. Dublin Core[8] or SKOS[9].

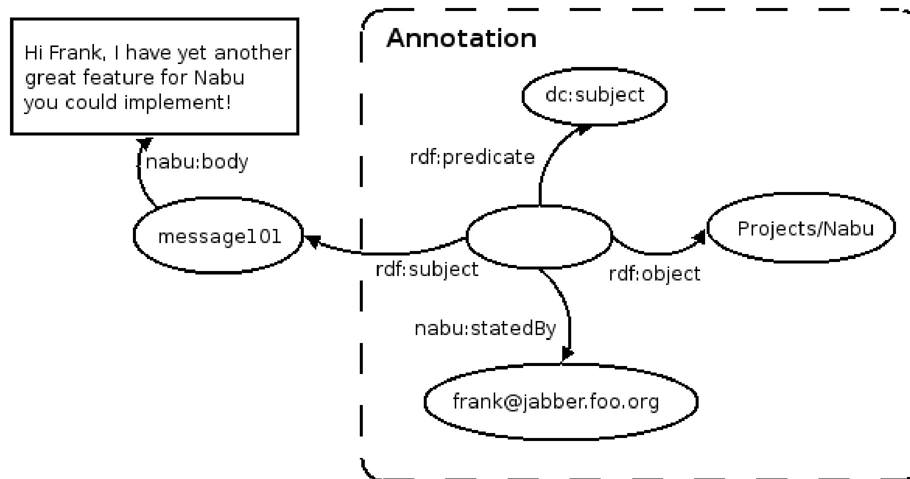
As an example, let us classify a (fictional) message sent by my advisor: "Hi Frank, I have yet another great feature for Nabu you could implement". To make searching easier, I want to assign the project the message is related to, in this case Nabu.

To add annotations, use the CREATESTATEMENT request:

```
CREATESTATEMENT RESOURCE http://foo/Messages/someMessage
   http://purl.org/dc/terms/subject http://foo/Categories/Projects/Nabu
```

The first argument must be one of RESOURCE or LITERAL and indicates whether the object of the statement should be handled as resource URI or as literal string. The following tokens are the (subject, predicate, object) triple representing the statement. Nabu transforms the example into the following RDF representation:

[8] http://dublincore.org/
[9] http://www.w3.org/2004/02/skos/



The statement is *reified*, which means that the statement itself becomes a resource that is linked to subject and object. That makes it possible to add properties to the statement. In Nabu, every user-added statement has a property *statedBy*, linking the creator of the statement. In our example, this is frank@jabber.foo.org. The statement is "owned" by the linked account. Only this account can delete the statement. Also, users can read the *statedBy* property and decide whether they trust the statement or not: Alice might decide that annotations made by Charlie are useful and consider them, but ignore Bob's statements because she found out that they do not make much sense.

Nearly every kind of RDF statement can be added. The only restriction is that properties from the Nabu ontology are not allowed for user statements. E.g., dc:subject (dc = Dublin Core[10]) is valid, but nabu:isInRoom is not, because the predicate nabu:isInRoom is part of the Nabu ontology. This prevents that users can (deliberately or not) corrupt the Nabu archive or compromise privacy settings. Properties from the Nabu ontology are managed by the server and can only be modified indirectly by the user using the requests defined in the Nabu protocol (p. 20).

To remove a statement, use DELETESTATEMENT:

```
DELETESTATEMENT RESOURCE http://foo/Messages/someMessage
   http://purl.org/dc/terms/subject http://foo/Categories/Projects/Nabu
```

removes the statement added above. It can only be removed by the user who added it.

[10] http://dublincore.org/documents/dces/

2.6 Log Sharing and Privacy Settings

To gain acceptance for Nabu and server-side logging in general, it's important to ensure the user's privacy. This means that Nabu must

1. leave the user in full control over what is logged and what not
2. give the user the ability to delete sensitive information at any time
3. Allow access to the archive only over a clearly defined interface that handles authentication and respects the privacy settings
4. be conservative when it comes to defaults (i.e. disable logging, use restrictive privacy settings)

On the other hand, one of Nabu's goals is to encourage information sharing within peers to make valuable information available to others when wanted. So a privacy model is needed that satisfies both needs, ensuring privacy and making it possible to share conversation logs.

In Nabu, every user is the owner of the message he sent, and he can control who can read his messages or delete them later if wanted. This means that a message is under control of the message sender and only of the message sender. If two users have a conversation, every user is responsible for his own messages and has no control over the messages he received from his dialog partner.

Access control is managed via *privacy policies*. A privacy policy contains a set of rules that control read permissions by allowing or denying access to certain accounts or groups of accounts. Every message logged has a link to a privacy policy that controls the access to the message. Every user has a list of policies he can assign to logged messages. There is always exactly one policy active at a time. Whenever the user writes a message, Nabu logs the message and links it to the currently active policy. Policies are linked, not copied: For instance, when the user adds a new account to his policy "friendsOnly", the added account gains access to all archived messages that already use the "friendsOnly" policy.

What does it actually mean that a resource is not accessible? When a message (or any resource in general) is not accessible, this means that the resource itself and its CBD[11] is completely hidden from the user: The resource itself and all links from or to the resource are hidden. When querying the model, the resource won't show up in the results. For messages in particular this means that neither the message content nor any links to the message are visible. This includes user statements (p. 14): If a user statement was added to link the message to a category, the statement is not visible. This is important, because we do not want other users to read the topics we were talking about, even if they cannot read the actual message content.

[11] http://sw.nokia.com/uriqa/CBD.html

2.6.1 Privacy Policies in detail

Every privacy policy has

- a name
- an owner
- a set of rules allowing or denying access to an account or a group of accounts

The name is an arbitrary string without spaces, e.g. ”default”, ”friends”, ”workGroup”. In the requests (p. 20) for policy management the name is used to identify the policy. Thus the policy name must be unique for a user (but of course two users can use the same name without conflicts).

The owner is the account that owns the policy. The owner can edit the policy and add or removes rules. The policy always implicitely grants access to the owner, so the owner can access his own messages even if the rules would deny it. It is only possible for a user to change the policy for a message when he owns the currently linked policy.

The rules: A policy can contain any number of rules of the form ”allowAccount <accountURI>”, ”denyAccount <accountURI>”, ”allowGroup <groupName>”, ”denyGroup <groupName>”.

The rules are applied in (deny, allow) order: If access is not explicitely allowed, it is denied. I.e. a policy without any rule denies the access completely (except to the policy owner).

If both rules exist that allow and deny access for an account, the deny-rule ”wins” and the access is denied.

2.6.2 Groups

As mentioned before, access permissions can not only be set per user but also per group. A group is a plain set of accounts, set up by the user to make privacy management easier. For example, a user could set up a group ”friends”, and add the accounts of his friends to the group. Instead of allowing access per account, he can do a simple ”allowGroup friends” and all accounts in the group gain access.

2.6.3 Examples

Here are some examples demonstrating how privacy policies are applied.

There are five accounts, Alice, Bob, Charlie, Daniel and Emily. Alice is the owner of the policies, and she created a group friends with Bob and Emily in it. (Note that in the implementation, full account URIs are used, but I use Alice instead of http://foo/Accounts/jabber.foo.org/alice for clarity here)

```
policyOwner Alice
allowAccount Daniel
allowAccount Bob
```

Allows access to Alice (as she is the owner), Daniel and Bob.

```
policyOwner Alice
allowGroup friends
allowAccount Charlie
```

Allows access to Alice as she is the owner, friends group, which is Bob and Emily, and Charlie. So just poor Daniel may not read the resource (well, the rest of the world can't either).

```
policyOwner Alice
allowGroup friends
denyAccount Bob
```

The first directive allows access to the "friends" group, i.e. Bob and Emily. But as the second directive denies access for Bob explicitely, only Emily has access (and Alice of course).

For more information on how to manage and use policies and groups, read the Nabu Protocol (p. 20) section.

2.7 User Observation

One topic addressed in the research project EPOS[12] is user observation: By observing the user's actions, EPOS tries to identify the context of the desktop user to support him in his work. Depending on the current context of a user, different contacts, files or other resources are relevant. For instance, the context information can be used to present currently relevant contacts from the addressbook to the user. EPOS does this using a *assistant bar*, a desktop panel listing e.g. relevant contacts, resources, and projects.

Collecting observation data is done by special plugins for e.g. word processors, WWW browsers or mail clients. Each plugin observes the user's actions in the respective application and sends them to a central context elicitation component. Nabu offers this functionality for instant messaging, notifying messages from or to the observed user to EPOS. As there is a multitude of Jabber clients and even in the research group various clients are used, it's more practical to implement this on the server-side instead of providing plugins for every client out there.

It is important to note that in Nabu user observation is **fully controlled by the observed user**. It must be activated by the user and can be stopped at any

[12] http://www3.dfki.uni-kl.de/epos/

time. So ”user observation” is not meant to enable your boss to monitor your actions, but to enable frameworks like EPOS to analyze you current context to support your work. Observing apps need your Jabber account password to register at the server.

Usually observation will be integrated into the context framework by using the client API (p. 38).

The observation works as follows:

To observe messages from and to Alice (alice@myserver.org), the observer program logs in at the server as alice@myserver.org, like the user does with her graphical client. The observer program must use its own resource, e.g. ’observation’. To start the observation, the program sends

```
OBSERVEMESSAGES on observation
```

to the server (to test observation, you can also send this manually to the bot). This register the resource ’observation’ as observer, from now on all messages Alice sends or receives are notified to the ’observation’ resource. A notification message consists of a subject, containing the URI of the notified message, and the message body, containing the message CBD[13].

Example:

Subject: ”newMessage http://foo#instances/Messages/km-intern.opendfki.de/heim/km-intern.opendfki.de/osterfel/20050713-115134.762”

Body:

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:nabu="http://www.opendfki.de/ont/nabu#">
   <nabu:Message rdf:about=
       "http://foo#instances/Messages/km-intern.opendfki.de/
        heim/km-intern.opendfki.de/
        osterfel/20050713-115134.762">
   <nabu:body>und? hats funktioniert?</nabu:body>
   <nabu:streamID>mir_6</nabu:streamID>
   <nabu:receivers rdf:resource="http://foo#instances/Accounts/Jabber/
                              km-intern.opendfki.de/osterfel"/>
   <nabu:sender rdf:resource="http://foo#instances/Accounts/Jabber/
                           km-intern.opendfki.de/heim"/>
   <nabu:messageType>chat</nabu:messageType>
   <nabu:subject></nabu:subject>
   <nabu:datetime>2005-07-13T11:51:34.781+0200</nabu:datetime>
   <nabu:previousMessageInRoom rdf:resource=
        "http://foo#instances/Messages/km-intern.opendfki.de/
         osterfel/km-intern.opendfki.de/heim/20050713-115108.943"/>
   <nabu:inRoom>
```

[13] http://sw.nokia.com/uriqa/CBD.html#definition

```
     <rdf:Description rdf:about=
     "http://foo#instances/P2PRooms/km-intern.opendfki.de/
      heim/km-intern.opendfki.de/osterfel/">
       <nabu:lastMessage rdf:resource=
       "http://foo#instances/Messages/km-intern.opendfki.de/heim/
        km-intern.opendfki.de/osterfel/20050713-115134.762"/>
     </rdf:Description>
   </nabu:inRoom>
 </nabu:Message>
</rdf:RDF>
```

To disable observation, the observer client must send

```
OBSERVEMESSAGES off observation
```

Read more on how to integrate user observation in applications in the development (p. 38) section.

2.8 The Nabu Protocol

Users can interact with Nabu via the Nabu Bot. The Nabu Bot is a server component listening to user commands.

Using the Nabu Bot, you can

- Query the repository (p. 9) via SPARQL
- Annotate messages (p. 14) by adding RDF statements
- Manage your privacy settings (p. 16)
- Register und unregister as user observer (p. 18)

To use the Nabu-Bot, you have to add it to your contact list. Read here (p. 8) for instructions. When the Bot appears in your roster you can send requests to the user as described below. For a detailed description of the responses, read the response reference (p. 45) in the reference section.

2.8.1 Basic Commands

LogPresence

Syntax: LOGPRESENCE ON|OFF|STATUS

ON/OFF enables/disables the logging of presence changes. If enabled, all presence changes of the user are logged by Nabu, otherwise Nabu ignores them. By default, presence logging is disabled. STATUS returns whether presence logging is enabled or disabled.

Returns responses 200, 250, 251

LogMessages

Syntax: LOGMESSAGES ON|OFF|STATUS

ON/OFF enables/disables logging of message changes. If enabled, all message sent by the user are logged by Nabu, otherwise Nabu ignores them. By default, message logging is disabled. STATUS returns whether message logging is enabled or disabled.

Possible responses: 201 (status of message logging), 252 (message logging enabled), 253 (message logging disabled)

DeleteMessage

Syntax: DELETEMESSAGE <uri>

Deletes a message. The message must be sent by the message sender, so every user is allowed to delete his own messages, giving him the possibility to remove sensitive information from the server.

Possible responses: 280 (message deleted), 401 (permission denied), 441 (resource not found)

2.8.2 Queries

Query

Syntax: QUERY SPARQL|SEARCHMSG <query>

Queries the user-visible part of the Nabu model. You can either

- start a simple query using SEARCHMSG: "QUERY SEARCHMSG Nabu" returns all messages whose body contains the string "Nabu". The search is case-insensitive, so "NABU" or "naBu" also match. This is a simple way to query the archive without constructing complex queries.
- use SPARQL: "QUERY SPARQL <queryString>" executes a query using the SPARQL[14] query language. You can use all types of SPARQL queries: Ask, Select, Describe and Construct. For more information and examples see the 'Searching the Archive' section (p. 9).

Possible Responses: 210 (query result), 420 (Wrong query syntax)

[14] http://www.w3.org/TR/rdf-sparql-query/

Model

Syntax: MODEL

Returns the Nabu model as RDF/XML. Resources (and their CBD[15]) not visible by the requesting user are omitted.

Possible Responses: 211 (model as RDF)

ShowCBD

Syntax: SHOWCBD <uri>

returns the CBD[16] of a given URI as RDF/XML. The CBD contains reifications (such as annotations) and left links (links *to* the resource). If the resource is not accessible by the requesting user, 401 is returned.

Possible Responses: 211 (model as RDF), 401 (permission denied), 441 (resource not found)

2.8.3 Annotations

CreateStatement

Syntax: CREATESTATEMENT LITERAL|RESOURCE <subject> <predicate> <object>

Adds an RDF statement to the graph. The statement is reified and has a property "statedBy" (see Ontology (p. 11) section for details), linking to the account who made the statement. The second token LITERAL|RESOURCE indicates if the object is a literal or a resource URI. For literals, everything after the predicate is considered as the literal string. Possible Responses: 270 (statement created), 401 (permission denied)

DeleteStatement

Syntax: DELETESTATEMENT LITERAL|RESOURCE <subject> <predicate> <object>

Removes all the user's <S, P, O> statements from the model. Statements of other users are not affected. The second token LITERAL|RESOURCE indicates if the object is a literal or a resource URI.

Possible Responses: 271 (Statement deleted), 442 (Statement not found)

[15] http://sw.nokia.com/uriqa/CBD.html
[16] http://sw.nokia.com/uriqa/CBD.html

2.8.4 Privacy Settings

ListGroups

Syntax: LISTGROUPS

Returns a list of groups defined by the user. The reponse has the format <n> [<groupURI> <groupName>]^n where n is the number of user-defined groups

Possible Responses: 230 (group list)

CreateGroup

Syntax: CREATEGROUP <groupName>

creates a group with a given group name. <groupName> may be an arbitrary Unicode string without spaces. group names must be unique per user.

Example: createGroup family

Possible Responses: 265 (group created), 432 (group already exists)

DeleteGroup

Syntax: DELETEGROUP <groupName>

deletes a user-defined group. The group is removed from all policies.

Possible Responses: 266 (group deleted), 433 (group not found)

EditGroup

Syntax: EDITGROUP <groupName> (ADD|REMOVE <accountURI>)| SETPOLICY policyName

Edits a user-defined group. The user can add or remove group members, or set a privacy policy for the member list.

Examples:

'EDITGROUP friends ADD http://foo/Accounts/myserver/alice' adds alice@myserver to the group 'friends'.

'EDITGROUP friends REMOVE http://foo/Accounts/myserver/alice' removes her again.

'EDITGROUP friends SETPOLICY friendsOnly' sets policy 'friendsonly', which might restrict access to the member list to the group members.

Possible Responses: 267 (group edited), 433 (group not found), 441 (account not found), 431 (policy not found)

ListPolicies

Syntax: LISTPOLICIES

lists the names of the user's policies. The format is

<n> [<policyName>]ˆn

Possible responses: 220 (policy list)

CreatePolicy

Syntax: CREATEPOLICY <policyName>

Creates a policy with a given name. The policyName is an arbitrary utf8 string but must not contain any spaces. The new policy is initially empty, i.e. access is denied for everyone except the policy owner. If the policy already exists, an error is returned.

Example: 'CREATEPOLICY friendsOnly' Possible Responses: 260 (policy created), 430 (policy already exists)

DeletePolicy

Syntax: DELETEPOLICY <policyName> <newPolicyName>

Deletes policy <policyName>. All occurences of <policyName> are replaced by the <newPolicyName>.

Example: 'DELETEPOLICY friendsOnly topSecret' deletes policy 'friendsOnly' and sets policy 'topSecret' for all resources that were using policy 'friendsOnly' before.

Possible responses: 262 (policy deleted), 431 (policy not found)

EditPolicy

Syntax: EDITPOLICY <policyName> ADD|REMOVE ALLOWACCOUNT|DENYACCOUNT|ALLOWGROUP|DENYGROUP <uri>

Allows the user to edit his policies. He can add or remove rules, allowing or denying access for groups or single accounts. (Deny, Allow) semantics are used: Access is denied by default and must be explicitely granted using ALLOWACCOUNT or ALLOWGROUP. DENY rules have higher priority: For instance, if Bob is member of group 'friends' and two rules 'ALLOWGROUP friends' and 'DENYACCOUNT bob' exist, access is denied for Bob. The explicit denial has higher priority than the ALLOWGROUP.

Examples:

'EDITPOLICY friendsOnly ADD ALLOWACCOUNT http://foo/Accounts/myserver/bob' adds a rule to allow access for bob@myserver.

'EDITPOLICY friendsOnly REMOVE ALLOWACCOUNT http://foo/Accounts/myserver/bob' removes the rule again.

'EDITPOLICY friendsOnly ADD ALLOWGROUP http://foo/Groups/myserver/bob/BestFriends' allows access for all members in the group 'best friends'.

Possible responses: 263 (policy edited), 431 (policy not found)

ShowPolicy

SHOWPOLICY <policyName>

Returns a description of policy <policyName>, listing all rules.

Possible responses: 221 (policy description), 431 (policy not found)

ShowActivePolicy

Syntax: SHOWACTIVEPOLICY

Like ShowPolicy, but desccribes the currently active policy.

Possible responses: 221 (policy description)

SetActivePolicy

Syntax: SETACTIVEPOLICY <policyName>

Activates policy <policyName>. The currently set policy is attached to sent messages.

Possible responses: 261 (policy set), 431 (policy not found)

ChangePolicy

Syntax: CHANGEPOLICY <URI> <newpolicyName>

Changes the policy for a given resource to <newPolicyName>. The resource must already have a policy owned by the user.

Possible responses: 264 (policy changed), 401 (permission denied), 431 (policy not found), 441 (resource not found)

2.8.5 User Observation

ObserveMessages

Syntax: OBSERVEMESSAGES (ON|OFF resource)|STATUS

enables/disables the user observation. "OBSERVEMESSAGES ON someResource" registers the resource someResource as user observer. All messages from or to foo@bar are notified to foo@bar/someResource.

STATUS returns a list of the observing resources.

Possible responses: 202 (obervation status), 254 (user observation enabled), 255 (user observation disabled)

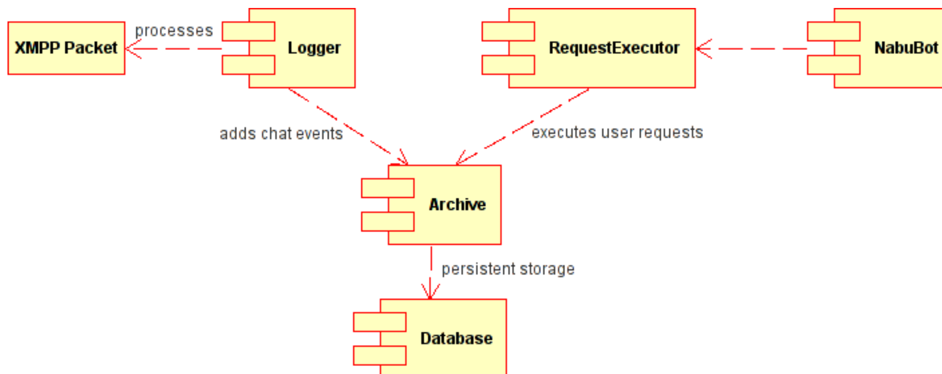
For details see User Observation (p. 18).

Chapter 3

Development

3.1 Design

Nabu is implemented as Jive Messenger plugin[1]. The following graph shows the top-level components of Nabu:



The central component of Nabu is *Archive*. *Archive* contains the RDF model, consisting of the public model, that stores the conversation logs and can be accessed from outside, and an internal model, managing internal configuration data and privacy policies. The Nabu implementation uses Jena[2] for RDF handling, graphs are stored persistently in a database, where the database backend is fully encapsulated by Jena.

Archive is accessed in two ways:

- Logging: the plugin intercepts XMPP messages, converts them to RDF and stores them in *Archive*. This is done by the *Logger* component. The

[1] http://www.jivesoftware.com/builds/docs/liveassistant/latest/documentation/plugin-dev-guide.html

[2] http://jena.sourceforge.net/

logger component simply takes the message, checks whether logging is enabled and if so, adds the RDF message to the RDF graph.

- User requests like search queries are executed on the RDF graph. This is done by the *RequestExecutor* interface. It takes parsed requests in the form of request objects, executes them, and returns a response object.

The *Nabu Bot* component is the interface between the users and the plugin. It parses user requests as defined in the Nabu protocol (p. 20), creates request objects and passes them to the *RequestExecutor*. It takes the returned responses, encodes them in a string and sends them back to the requestors.

3.2 Internal RDF Schema

The Nabu store consists of two RDF models:

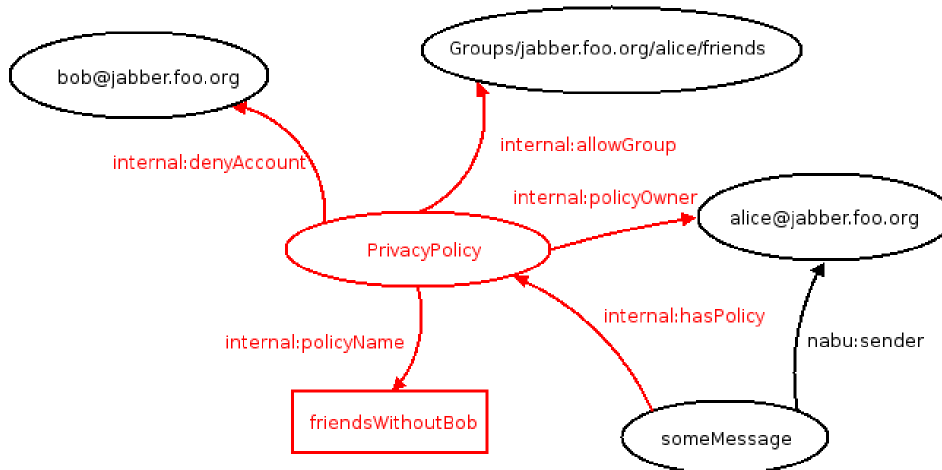
- The actual RDF data, that can be queried from the outside, i.e. logged messages, presence changes and annotations. This part is described in the Ontology section (p. 11) of the user handbook).
- Internal data, like account settings (e.g. logging enabled/disabled), and privacy policies. This data can't be queried from the outside, and the users will never see the RDF representations. It's only accessible through the requests defined in the Nabu protocol (p. 20), like e.g. LOGMESSAGES or SHOWPOLICY. This section describes these classes.

The internal classes and properties are in the namespace http://www.opendfki.de/ont/nabu/internal# and are prefixed with "internal" in this document.

In the following graphs, they are shown in red color, opposed to the classes and properties from the public part of the Nabu ontology which are in black. For a full reference of the internal RDF schema, see the RDFS[3].

[3]http://nabu.opendfki.de/cgi-bin/trac.cgi/file/trunk/ontology/nabu.rdfs?rev=126&format=raw

3.2.1 PrivacyPolicy



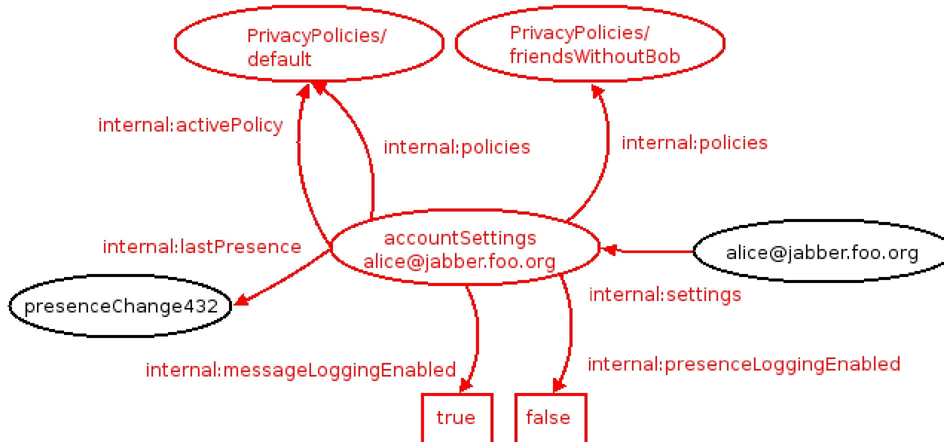
This graph shows the last policy example from the Log Sharing and Privacy Model (p. 16) handbook section. The policy "friendsWithoutBob" is owned by Alice. It allows access to her "friends" group, and denies it for Bob:

```
policyName friendsWithoutBob
policyOwner Alice
allowGroup friends
denyAccount Bob
```

The policy is attached to a message "someMessage" which was sent by Alice. While messages, groups and accounts are part of the public model, the privacy policy itself and all properties like allowGroup, denyAccount and hasPolicy are stored in the internal model.

3.2.2 AccountSettings

For every account in the public model, the internal model contains a corresponding AccountSettings instance saving settings related to this account:



This example graph shows the AccountSettings instance of alice@jabber.foo.org. It has the following properties:

- *internal:messageLoggingEnabled* and *internal:presenceLoggingEnabled*: Store whether message respectively presence logging are enabled or not. Both default to *false*.

- *internal:policies* link to the privacy policies owned by the account.

- *internal:activePolicy* links to the currently active policy. This policy is attached when a presence change or message is logged, as described in Log Sharing and Privacy Settings (p. 16).

- *internal:lastPresence* links to the last logged presence change of the respective account. This makes it fast and easy for the logger to find the last presence and link new presences to it via the *nabu:previousPresenceChange* property.

3.3 URI Schema

This section describes the URI schema used for the logged RDF data. Every instance is prefixed with ${instanceprefix}, making the URIs unique. The instance prefix must be set in the Nabu configuration.

Note: The URI schema is for internal use in the server only. You should not rely on the URI schema when processing RDF received by Nabu but use the properties linked to the resources. E.g. do not parse the date from the URI but read the nabu:datetime property!

3.3.1 Time Stamps

Time stamps are YYYYMMDD-hhmmss.fff (f for milliseconds)

Example: 20050416-221603.042 = 04/16/2005, 22:16:03.042

3.3.2 Account

Schema:

```
Accounts/$DOMAIN/$NODE
```

Example: frank79@jabber.fsinf.de ==> Accounts/jabber.fsinf.de/frank79

3.3.3 Room

P2PRoom

Schema:

```
${instanceprefix}/P2PRooms/$ACCOUNT1/$ACCOUNT2
```

Accounts are ordered alphabetically (ascending):

```
P2PRooms/jabber.fsinf.de/frank79/serv-3100.dfki.de/schwarz
```

is the P2P room of frank79@jabber.fsinf.de and schwarz@serv-3100.dfki.de. frank79@jabber.fsinf.de is listed first, as jabber.fsinf.de < serv-3100.dfki.de

```
P2PRooms/jabber.fsinf.de/admin/jabber.fsinf.de/frank79
```

is the room of frank79@jabber.fsinf.de and admin@jabber.fsinf.de. admin@jabber.fsinf.de is listed first, as the domain is equal and admin < frank79.

MUCRoom

Schema:

```
${instanceprefix}/MUCRooms/$DOMAIN/$ROOMNAME
```

For example, the room "support@jabber.foo.org" has the URI /MUCRooms/jabber.foo.org/support.

3.3.4 Message

Schema:

```
${instanceprefix}/Messages/$ROOM/$TIMESTAMP
```

Example:

A message sent by frank79@jabber.fsinf.de to schwarz@serv-3100.dfki.de (or vice versa) on July 16th 2005, 22:16:03.042 has the URI:

Messages/jabber.fsinf.de/frank79/serv-3100.dfki.de/schwarz/20050716-221603.042

It is assumed that Nabu processes not more than one message per milli-second.

3.3.5 PresenceChange

Schema:

```
${instanceprefix}/PresenceChanges/$ACCOUNT/$TIMESTAMP
```

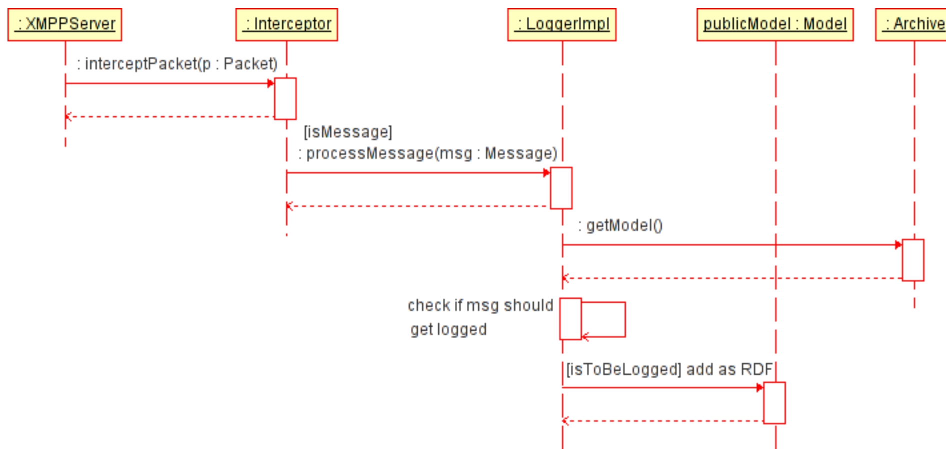
Example: A presence change of frank79@jabber.fsinf.de on July 16th 2005, 22:16:03.042: PresenceChanges/jabber.fsinf.de/frank79/20050716-221603.042

3.4 Implementation

In this section, I describe three central aspects of the implementation:

- Message logging, as it is the central mechanism in Nabu
- Execution of user requests, especially the optimizations that were needed to execute SPARQL queries in a reasonable time frame
- Logging of multi-user chats, as it differs from the logging of one-to-one chats in and needs some more effort to be mapped to the Nabu ontology.

3.4.1 Logging a message



The sequence diagram above shows the logging of a one-to-one message.

To receive all packets sent on the server, Nabu registers a PacketInterceptor[4] at the XMPP Server. The server passes every processed XMPP packet in a relatively low-level fashion (as a global packet sequence, with duplicates and without any grouping or sorting) to the interceptor, so Nabu can filter relevant packets and log them.

The *Interceptor* takes the packet, does basic filtering (discarding irrelevant packet types), casts the packets to subtypes (Message, Presence) and passes them to the *LoggerImpl*. *LoggerImpl* now checks whether the packet should be logged, by reading the account settings (p. 28) of the message sender from the Archive. If message logging is enabled, it continues checking whether the message is equal to the last message logged, to avoid duplicates. This is neccessary as every one-to-one message is intercepted twice, when entering the server (coming from the sender) and when forwarded to the message receiver. As packet IDs are optional, the Logger falls back on comparing message subject and body when no ID is set: If subject and body are equal, the message is considered as duplicate and discarded.

If the message passes the dupe test, Logger converts it to RDF (using Jena2Java) and stores it in the model. The currently active privacy policy is attached by adding the statement (message, nabu:hasPolicy, activePolicy) to the internal model.

3.4.2 User Requests

The processing and execution of user requests is separated in several steps. One important design goal was to separate request representation, parsing and

[4]http://www.jivesoftware.org/builds/messenger/docs/latest/documentation/javadoc/org/jivesoftware/messenger/inter

execution from each other. That means that the protocol (p. 20) used over the Nabu bot is encapsulated and separated from the Nabu core. This makes it easy to implement other protocols than XMPP for archive access, like e.g. XMLRPC, by implementing a few interfaces.

When the user sends a request, the Bot receives the request string in an XMPP message. The bot passes the request string to its implementation of the RequestParser interface. The parser returns the corresponding Request object, which is an abstraction from the protocol syntax. The bot now passes the request to the RequestExecutor implementation that was passed to the bot at creation time. The request executor executes the request, having full access to the RDF store. It creates a corresponding response object and returns that to the bot. The bot again uses its ResponseEncoder implementation to convert the response into a string and sends it back to the requestor.

To ensure privacy, it is important that every interface receiving user requests verifies the identity of the requestor. We get that for free when using XMPP (as the user is identified and authenticated when sending the request to the bot), but other implementations have to take care of this, otherwise the privacy model would be compromised.

Executing queries

The request execution is pretty much straight-forward and fast for most of the requests and so I won't go into details here for each and every request type. Some more thought was needed for the implementation of SPARQL, executed using the QUERY request. The SPARQL interface must cope with different requirements:

- It must respect the privacy model (p. 16). This includes the removal of non-readable resources from query results ("post-filtering"), but there are also more subtle cases that can't be addressed by simple post-processing of query results: The result of a query like "Show me all accounts that exchanged messages in the last 7 days" would return a list of account pairs. Account resources are globally visible, so the post-processing would just leave them in the result, even if the corresponding messages are filtered out. But this is not sufficient: Not only hiding the actual message content is required, but also hiding the information that both users have exchanged messages is important. So non-accessible resources must be filtered *before* query execution.

- Query execution must scale up to large archives. Considering that the archive size increases linearly over time, and a Nabu installation could log messages over years, archives can get considerably large. Nabu should return query results in a reasonable time frame even when run on such large archives. Also, Nabu must scale in terms of memory consumption.

The privacy model requires the CBDs of privacy-protected resources to be hidden from unauthorized users. When working on a global RDF graph as Nabu does, this means that the graph most be pre-processed for each query to create the individual "view" of the requesting user, removing the statements that he is not authorized to access. It was decided to do this preprocessing instead of e.g. managing an individual model for every user, duplicating huge parts of the model, which would be error-prone (consistency problems) and make basic actions like logging a message or changing a policy expensive tasks.

The pre-processing algorithm in pseudo-code:

```
Model filterModel(Model m)
   toRemove = Model.new # creates an empty model in memory
   for each statement (resource, internal:hasPolicy, policy)

            in archive.internalModel
   begin
       if policy.deniesAccess(account)
          toRemove.add(CBD(resource)) # add CBD of non-accessible resource
       end
   end
   return m.difference(toRemove) # return the mainModel without

                                 # the statements in toRemove
end
```

There are two major bottlenecks in this algorithm that make a naive implementation slow (preprocessing on small archive containing 10000 statements took about 40 seconds, which is of course unacceptable). The first bottleneck is the check whether a policy allows access for every resource is expensive. Checking it for every protected resource creates an enormous overhead. Fortunately this can be solved quite easily: As the number of policies is very small compared to the number of protected resources (usually a user will have only a couple of policies), caching the deniesAccess result reduced the time needed for filtering by about 50%.

The second bottleneck is the creation of the CBDs. This was addressed by using a global CBD cache: The CBDCache class caches requested CBDs. As CBDs may change when statements are added to or removed from the model, the cache registers as listener at the model and invalidates the cache entries of statement subject and object when receiving an addedStatement or removedStatement event. Using these two optimizations, the time for filtering was reduced by about 95 percent.

Another approach tested was caching of the filtered models: When a user starts a query, the filtered model is added to a (user, model) map. If the same user starts another query, Nabu looks if there is an entry in the map and reuses the model if so. The advantage of model caching is that the costs for filtering reduce to zero on a cache hit, where filtering even with CBD caching is still an O(n) operation. Disadvantages:

- the cached model is not in sync with the main model and so user queries would operate on potentially outdated data. Expiring caches using a time limit decreases the hit ratio, and syncing all caches model on addedStatement/removedStatement is far more expensive than maintaining a CBD cache.

- On a cache miss, the full time for model filtering is needed. This makes the first user query very slow, and the speed of successive queries unpredictable (so a query may take either e.g. 1 second or 1 minute, which could confuse users).

- memory consumption: For a main model of size m, a cache of size n needs O(n*m) memory.

Model caching is implemented in ModelCache, but not activated in favour of CBD caching. A combination of CBD and model caching would be possible though. This would combine zero costs on a model cache hit and lower the costs for filtering on cache miss.

3.4.3 Multi-user Chat Support

Jabber was, as many other instant messaging protocols, primarily designed for one-to-one chat between two users only. So the core of the Jabber protocol does not support multi-user chat (MUC in short). MUC support is provided by a protocol extension instead, defined in the Jabber Enhancement Proposal(JEP) 0045 [5]. The concepts used in group chat as defined in JEP 45 are different to the concepts of one-to-one chat: Where one-to-one chat is uses the typical instant messaging concepts, MUC is similar to chat systems as IRC[5]. Nabu does not support the full feature set of JEP 0045 but only the most important part of it: the logging of text messages sent in chat rooms.

Ideally, there would be no strict separation of one-to-one and MUC chat: It should be possible to switch from a two-user to a n-user chat by simply adding more persons to the existing chat. Two-user chat would be just a special case of a n-user-chat. The Nabu ontology (p. 11) uses this concept and abstracts from the details of the MUC protocol as far as possible: A MUC message is simply a message with multiple receivers instead of one, and has a MUCRoom linked as "location". The higher level of abstraction from the protocol makes the ontology simpler, but also the implementation in Nabu a bit more complex.

Before explaining how MUC logging is implemented, lets first have a look how MUC works in principle. The central events important for message logging are joining a room, sending messages, and leaving the room:

[5]http://www.mirc.com/irc.html

Joining a Room

When Alice wants to join support@conference.jabber.foo.org, she first sends a presence packet to the room:

```
<presence
    from='alice@jabber.foo.org/Psi'
    to='support@conference.jabber.foo.org/SuperAlice'>
  <x xmlns='http://jabber.org/protocol/muc'/>
</presence>
```

The ”to” attribute contains the JID of the MUC room plus the desired nick for that room. When entering a MUC room, every user must choose a nick. This nick is used to identify him inside this room. The ”real” Jabber ID ”alice@jabber.foo.org” is not necessarily visible to other room occupants. So Alice chooses the (admittedly not very creative) nick ”SuperAlice”.

Sending Messages

After entering the room, Alice might want to send a (broadcast) message to the other occupants. This is done by sending a message from her nick to the room JID:

```
<message
    from='support@conference.jabber.foo.org/SuperAlice'
    to='support@conference.jabber.foo.org'
    type='groupchat'>
  <body>Hello everybody!</body>
</message>
```

When receiving the message, the MUC component broadcasts the message to all room occupants:

```
<message
    from='support@conference.jabber.foo.org/SuperAlice'
    to='bob@jabber.foo.org'
    type='groupchat'>
  <body>Hello everybody!</body>
</message>
<message
    ...
    to='charlie@jabber.foo.org'
</message>
```

Exiting a Room

The user exits the room by sending a presence packet with type=”unavailable” to her nick JID:

```
<presence
    from='alice@jabber.foo.org/Psi'
    to='support@conference.jabber.foo.org/SuperAlice'
    type='unavailable'/>
```

Logging MUC Messages

To implement MUC support, there are three issues Nabu has to address:

Intercepting MUC messages. Nabu does this by registering a special interceptor for the MUC subdomain (usually "conference") at the XMPP server. All packets sent to or from the MUC domain are passed to the interceptor. To reduce complexity, MUCInterceptor creates a MUCRoomHandler for every room and passes messages sent in a room to the responsible handler.

Resolving room nicks to "real" Jabber IDs. Apart from using different room subclasses for the room concept (MUCRoom/P2PRoom), Nabu abstracts completely from MUC details and stores one-to-one and MUC messages in the same manner. Thus Nabu does not store MUC nicks but the actual Jabber ID of the user. For instance, the example from above will be stored with alice@jabber.foo.org as nabu:sender and bob@jabber.foo.org, charlie@jabber.foo.org as nabu:receivers.

Keeping track of room members. To set sender and receivers correctly, Nabu must listen to join room and exit room events and maintain a list of the accounts being room occupants. This is very important, as wrong nabu:receivers links influence privacy: accounts incorrectly linked as receivers can read messages they never received.

Both the nick -> Jabber ID mapping and the occupant list are managed per room by the corresponding room handler.

3.5 The Client API

This section shows how you can access Nabu from your application. To make use of Nabu in your application, you need a Jabber client library to connect to Nabu, and an RDF parser to parse replies.

For Java, the Nabu project has a simple API you can use to listen to chat events for user observation and to communicate with the Nabu server via requests. The API encapsulates both the connection handling and the encoding/parsing of request and responses. The API does not include an RDF parser to avoid dependencies to a specific RDF implementation. To parse the RDF , you can use the Jena framework[6] (it will be used in the examples below).

3.5.1 Installation

To use the client API in your Java application, get the current version of nabu-clientAPI.zip from the Nabu homepage, unpack it and add the contained JAR files (nabu-client.jar, nabu-remote.jar, smack.jar, smackx.jar) to your classpath.

If you build from source, go to the nabuclient/ directory and run

[6] http://jena.sourceforge.net/

```
ant nabuclient_jar
```

After build, you can find the JAR files in nabuclient/target/lib. Add them to your classpath.

3.5.2 Connect to the server

The following code snippet shows how to connect to the server:

```
import de.dfki.km.xmpp.nabu.observer.Client;
import de.dfki.km.xmpp.nabu.observer.impl.ClientImpl;

// [...]

   String USER = "alice"; // the user name (node part of the Jabber ID)
   String RESOURCE = "nabu"; // the resource to be used, can be any

                             // resource that is not in use e.g. by

                             // your graphical client
   String PASS = "topsecret"; // the jabber password of alice
   String HOST = "foo.jabber.org"; // the server,

   int PORT = 5222; // the port, usually 5222 for unsecured

                    //connections and 5223 for SSL
   String BOT = "nabubot@nabu.foo.jabber.org"; // the JID of the nabu bot,

                                               // as listed in the contact list

   Client client = new ClientImpl(); // for a SSL connection, use

                                     // "new ClientSSLImpl()" (note that the
port
                                     // will change too, e.g. to 5223)
   try

   {
      client.login(USER, RESOURCE, PASS, HOST, PORT);
   }
   catch (Exception e) {}

   client.setBot(BOT);

   // here you can send requests, start observation etc. (see below)
   // [...]
```

3.5.3 User Observation

To use user observation, implement the Listener interface:

```
import de.dfki.km.xmpp.nabu.observer.Listener;
```

```
public class MyListener implements Listener
{

    public void receiveObservationNotification(String uri, String cbd)
    {
        System.out.println("Received message with URI " + uri);
        System.out.println("Description in RDF:");
        System.out.println(cbd);
        // Act on notification
    }

    public void receiveResponse(Response response)
    {
        System.out.println("A new response arrived! Response code: "
                           + response.getCode());
    }
}
```

Create an listener instance and register it at the client and start the observation:

```
   client.setBot(BOT);
   client.addListener(new MyListener()); // adds the listener
   client.startObservation(); // starts the observation
```

Now observation is activated and every received notification is passed to MyListener::receiveObservationNotification().

To stop the observation, call client.stopObservation();

3.5.4 Sending Requests

The client API also offers a convenient way to send request to Nabu and to receive responses without building request strings and parsing responses. To send a request, simply create a request object and pass it to the client. Let's send the annotation example (p. 14) from the handbook via the API.

```
CREATESTATEMENT RESOURCE http://foo/Messages/someMessage
http://purl.org/dc/terms/subject http://foo/Categories/Projects/Nabu
```

becomes:

```
int type = CreateStatementRequest.ObjectType.Resource;
String subject = "http://foo/Messages/someMessage";
String predicate = "http://purl.org/dc/terms/subject";
String object = "http://foo/Categories/Projects/Nabu";

CreateStatementRequest req =

   new CreateStatementRequest(null, type, subject, predicate, object);

client.sendRequest(req);
```

To handle responses, reimplement receiveResponse() from the Listener interface. To handle the different sub classes, you might also implement the ResponseVisitor interface:

```
public class MyListener implements Listener, ResponseVisitor
{
    public void receiveResponse(Response response)
    {
        visit(response);
    }

    public void visitMessageLoggingStatusResponse(MessageLoggingStatusResponse
response)
    {
        System.out.println("message logging is " + (response.getStatus()

                           ? " enabled " : "disabled") );
    }

    public void visitQueryResultResponse(QueryResultResponse response);
    {
        Model m = ModelMaker.createDefaultModel(); // create a Jena Model of

                                                   // the RDF
        m.read(new StringReader(response.getResult(), null);
        // do something with the model
    }
    // ...
}
```

By implementing the visitX class for every response type X, you can handle all the response types you are interested in. For instance, the visitQueryResultResponse() implementation in the example uses Jena to parse the RDF sent in the response.

Chapter 4

Conclusion

The Nabu project is an attempt to bring the Semantic Web and instant messaging together, making the increasing amount of information exchanged via instant messaging accessible using semantic web technology.

An ontology was developed to describe instant messaging conversations. Using the RDF standard to represent the data and the promising SPARQL query language for user queries, Nabu integrates well in existing semantic web infrastructures. To make better use of the stored information, users can attach metadata to their logs.

A privacy model was developed to control the accessibility of RDF data, an area where no proven implementations or standards yet exist. Working on resource-level, it is possible to control accessibility per resource. Although it has limitations when you need more fine-grained control and hide certain properties only, it works well for Nabu.

The concepts were implemented as an extension for the Jive XMPP server. Thus a proof-of-concept implementation is available and can be used by interested people to integrate instant messaging and semantic web. In the DFKI KM working group, it is already used in the EPOS[1] project. The user observation functionality was successfully integrated into the context elicitation and tested.

Nabu is still a prototype. To make it suitable for wide-spread use, more effort and feedback is needed. The main issues are:

- The user-nabu communication is currently text-based. This is flexible because it can be used on every platform and in every client, but is neither convenient nor user-friendly. Graphical frontends would be desirable, preferably integrated in client software (via plugins). Other options are a web frontend or the integration in frameworks for desktop search.

- Evaluation is needed to find out whether the chosen privacy model meets the user requirements. This needs experience from daily use of "real

[1] http://www3.dfki.uni-kl.de/epos/

users", as different usage patterns need different privacy models. At the moment, there is always one policy active at a time. The advantage is that it is easy to manage and clear which policy is used for the current chat. It would also be possible to specify a policy for specific chats, e.g. "everything I write in MUC room #workgroup should be readable by the whole workgroup". While this is more powerful, it has the disadvantage is that the user could forget about the channel-specific setting and share information with more people than intended. A third option would be to always use restrictive privacy settings when logging (i.e. only participants can read messages). Users would manually share the log afterwards by marking the conversation in their client plugin and assigning a less restrictive policy. This usage pattern is already supported, the user must just leave the default policy active, and assign other custom policies to logged messages.

- On the implementation side, the time needed for query execution could become critical on servers with many users that store chat information over a long time. The scalability of the used approach (working on the whole graph, removing the invisible information from the graph when querying) is limited, and caching CBDs and models will help only up to a certain graph size, user number and query frequency.
- Currently Nabu is only accessible via the XMPP protocol. To make access easier, it should be possible to query the archive using HTTP(S) requests. On the other hand, every additional way to access the archive is an additional security risk. Besides the usual vulnerabilities web servers have, the interface must manage user authentication, something we get for free when using the Jabber protocol.

Bibliography

[1] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[2] Marc Eisenstadt and Martin Dzbor. BuddySpace: Enhanced Presence Management for Collaborative Learning, Working, Gaming and Beyond. Submission to JabberConf Europe 2002.

[3] Justin Karneges and Ian Paterson. JEP-0136: Message Archiving. Jabber Enhancement Proposal. URL `http://www.jabber.org/jeps/jep-0136.html`.

[4] Dennis Quan, Karun Bakshi, and David R. Karger. A unified abstraction for messaging on the semantic web. In *WWW (Posters)*, 2003.

[5] Peter Saint-Andre. JEP-0045: Multi-User Chat. Jabber Enhancement Proposal. URL `http://www.jabber.org/jeps/jep-0045.html`.

[6] Sven Schwarz. A Context Model for Personal Knowledge Management. In *Proceedings of the IJCAII'05 Workshop on Modeling and Retrieval of Context*, Edinburgh, 2005.

Appendix A

Response Codes

This section describes the possible server responses of the Nabu protocol. Every request must be answered by exactly one of these responses. There are three types of responses:

- *Replies.* Replies return information, like logging states ,query results etc. Reponse codes 200-299
- *Acknowledgements.* Acknowledge that the corresponding request was executed, without containing more detailed information. Response codes 300-399
- *Error responses.* Resporting errors or illegal requests, response codes 400-599

A response might have a explanatory string appended to the formal response,

like "200 TRUE message logging is enabled". Everything after "200 TRUE" has no formal meaning and is for human readers only. It should be ignored by parsers. Responses of variable length like query results do not contain these explanatory comments.

If the request was prefixed by a request ID, the response will be prefixed with the same ID. Example: For a request "0x04DDAA LOGMESSAGES STATUS", the response might look like this: "0x04DDAA 201 TRUE message logging enabled".

A.0.5 Replies

Replies are responses that return information, like logging states, query results etc.

200 (status presence logging)

Syntax: 200 TRUE|FALSE

Presence logging is enabled iff TRUE is returned-

201 (status message logging)

Syntax: 201 TRUE|FALSE

Message logging is enabled iff TRUE is returned-

202 (status user observation)

Syntax: 202 <n> [<resource>]^n

Returns a list of resources registered as user observers. <n> is the number of resources, followed by the resource names.

Example:

```
202 2 observer1 observer2
```

Here two resources, observer1 and observer2, are registered as observers.

210 (query result)

Syntax: 210 <queryResultAsRDF>

Returns the result of a query in RDF/XML format.

211 (model result)

Syntax: 211 <modelAsRDF>

Returns the complete user-visible part of the Nabu model in RDF/XML format

220 (policy list)

Syntax: 220 <n> [<policyName>]^n

Returns the the policy list of the requesting user. n is the number of privacy policies, <policyName> is a policy name (not a URI!).

Example:

```
220 3 default friendsOnly workGroup
```

A list containing three policies, with names "default", "friendsOnly" and "workGroup".

221 (policy description)

Syntax: 221 <policyName> <policyOwner> <n> <directive>ˆn

<policyOwner> is the Jabber ID of the owner account, <n> is the number of directives. <directive> has the form ”<rule> <accountOrGroupURI>”.

Example:

```
221 default alice@jabber.foo.org 2
 allowGroup  http://foo.org/nabustore/Groups/jabber.foo.org/alice/workGroup
denyAccount http://foo.org/nabustore/Accounts/jabber.foo.org/bob
```

The policy name is ”default”, owned by ”alice@jabber.foo.org”. The policy has two directives, allowing access to the group workGroup and denying it for bob@jabber.foo.org.

222 (active policy)

Syntax 222 <policyName>

Returns the name of the currently active policy

230 (group list)

Syntax: 230 <n> [<groupURI> <groupName>]ˆn

Returns a list of user-defined groups n is the number of groups, followed by a (group URI, group name) pair for every group

Example:

```
230 2 http://foo.org/nabustore/Groups/jabber.foo.org/alice/friends friends
      http://foo.org/nabustore/Groups/jabber.foo.org/alice/family family
```

249 (debug)

This returns debug information, request via the DEBUG request. This response is not formally specified and for interactive use only.

A.0.6 Acknoledgement responses

These responses acknowledge that the corresponding request was executed, without containing any detailed information.

250 (presence logging enabled)

Presence logging was enabled.

251 (presence logging disabled)

Presence logging was disabled.

252 (message logging enabled)

Message logging was enabled.

253 (message logging disabled)

Message logging was disabled.

254 (user observation enabled)

User observation was enabled.

255 (user observation disabled)

User observation was disabled.

260 (policy created)

Policy was created.

261 (policy activated)

The active policy has changed.

262 (policy deleted)

Policy was deleted.

263 (policy edited)

Policy editing was successful.

264 (policy changed)

Policy was changed for a resource.

265 (group created)

Group was created.

266 (group deleted)

Group was deleted.

267 (group edited)

Group was edited.

270 (statement created)

User statement was added to the graph.

271 (statement deleted)

User statement was removed from the graph.

280 (message deleted)

Message was deleted.

A.0.7 Error responses

400 (bad request)

The request is unknown (unknown command name), or the syntax is wrong (invalid parameters).

401 (permission denied)

Permission was denied due to security and/or privacy settings. This is e.g. returned when the user tries to read a message he is not allowed to access.

420 (invalid query)

The query syntax (e.g. SPARQL) was invalid. A more description is appended.

430 (policy already exists)

Returned when the user tries to create a policy that already exists.

431 (policy not found)

Returned when the request refers to a non-existant policy.

432 (group already exists)

Returned when the user tries to create a group that already exists.

433 (group not found)

Returned when the request refers to a non-existant group.

441 (account not found)

Returned when the request refers to a non-existant account.

442 (resource not found)

Returned when the request refers to a non-existant resource.

443 (statement not found)

Returned when the request refers to a non-existant statement.

444 (group list not found)

Returned when the request refers to a non-existant group list.

500 (server error)

Returned when an internal server error occurs, due to a bug in the implementation.