Nabu – A Semantic Archive for XMPP Instant Messaging

Frank Osterfeld, Malte Kiesel, Sven Schwarz
DFKI GmbH - Knowledge Management Dept.
Erwin-Schrödinger-Straße, Bldg. 57
D-67663 Kaiserslautern, Germany
{frank.osterfeld, malte.kiesel, sven.schwarz}@dfki.de

Abstract

Instant messaging (IM) has become more and more common these days, and is complementing e-mail and other means of electronic communication. However, due to its heavily context-dependent nature, searching archives of instant messages using only full text search is a tedious task. Also, in contrast to mails, files, and other electronic media, instant messages typically do not feature a unique identifier or location, making it difficult to reference a particular instant messaging conversation. Nabu is a semantic archive for XMPP instant messaging designed to address these problems by implementing a semantic message store, using RDF(S) as its storage format. It is implemented as a server module and will log messages, manage access control to the archives on a per-user basis, and allow other components to observe and annotate messages.

1 Introduction

The importance of instant messaging (IM) for private and organizational communication has increased over the last years. IM, the instant sending and receiving of (mostly short) text messages between two or more users, complemented by a list of peer contacts along with their online status, has become one of the most used communication channels on the internet, and more and more valuable information is exchanged via instant messages, especially among colleagues at work.

Despite of the increasing amount of information exchanged, IM client support for archiving and searching the messages exchanged is poor. This is understandable, as on the one hand, most IM client applications are intended for private users for whom other features are more important. On the other hand, IM messages are typically very short and heavily tied to their particular context, thus making efforts to organize the archive of exchanged messages a lot more difficult than it is the case with other means of communication, such as e-mails, where the text is essentially self–contained. Moreover, e-mails come with a variety of additional information such as a subject or thread references which are usually missing in instant messages.

While e-mail can be archived in a long-term manner on 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 proprietary 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, and they quickly become out of sync. In addition, information gets lost easily. Using proprietary, client- and protocol-specific formats to store the information complicates managing and searching the stored information using other interfaces than the client UI.

In this paper we present Nabu[1], an open–source 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. The Jive Messenger XMPP server[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 [Berners-Lee *et al.*, 2001] 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 server. For retrieving the stored information, it supports the SPARQL query language [Eric Prud'hommeaux, 2005], which is currently going through the standardization process at the W3C.

Using XMPP as transport protocol for SPARQL queries and commands has several benefits. For example, XMPP takes care of authentication and encryption; also, XMPP uses a persistent connection, delivering higher performance than protocols that use non-persistent connections such as HTTP, which is used as transport protocol by XML–RPC and SOAP.

In addition to the logging of chat messages, further features of Nabu are:

[1] http://nabu.opendfki.de/
[2] http://www.jabber.org/
[3] http://www.jivesoftware.org/messenger/
[4] http://www.w3.org/RDF/

- 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, e.g., making a conference log available to the 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 [Schwarz, 2005], sending message notifications to the EPOS user observation (when enabled by the user). Other applications can also receive these events by registering with the Nabu component.

The rest of this paper discusses related work, Nabu's architecture, the RDF schemes used, RDF access control mechanisms, Nabu's observation feature, and possible applications, followed by conclusions.

Related Work

[Karneges and Paterson, 2004] proposed a storage format and protocol for server-side message archives as a Jabber protocol enhancement. The proposal suggests a simple protocol and storage format for message archiving. It defines its own format and does not use existing standards for message storage and retrieval apart from XML.

The *Haystack project* [Dennis Quan and Karger, 2003] builds a client for information management by integrating various information sources into one frontend, using an infrastructure based on RDF. A messaging model was developed [Quan *et al.*, 2003] 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 client-based.

The *BuddySpace* research project [Eisenstadt and Dzbor, 2002] extends the *presence* concept in Jabber (simple offline/online/busy states), and adds information such as geographical location, current work focus etc. Furthermore, it investigates how such additional semantics can be used to facilitate collaboration over networks. The BuddySpace Jabber client[5] demonstrates the concepts.

2 Architecture

Nabu is implemented as a plugin for the Jive Messenger XMPP server[6]. By implementing Nabu as a server–side component, every user of the Nabu–enabled server can use its services without requiring the installation of a client–side plugin. Since there are dozens of XMPP clients[7], this was clearly the optimal solution. Also, sharing annotations would be much more difficult with a client–side implementation since clients cannot be expected to be online at all times. Finally, moving complexity to the server side nicely fits into the XMPP philosophy.

[5] http://buddyspace.sourceforge.net/

[6] http://www.jivesoftware.com/

[7] In our department, at least four different clients are in use.

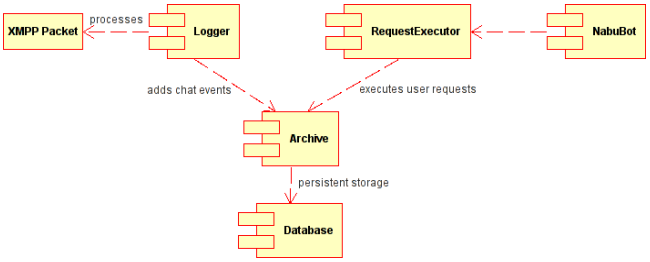

Figure 1: Nabu Components.

The graph shown in Figure 1 describes the top-level components of Nabu.

The central component of Nabu is the *Archive*. The *Archive* contains the RDF model, consisting of the public model that stores the conversation logs which can be accessed from outside, and an internal model, managing internal configuration data and privacy policies. The Nabu implementation uses Jena [Andy Seaborne et al., 2005] for RDF handling. Models are stored persistently in a database. The database backend is fully encapsulated by Jena.

The *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 logger component simply takes the message, checks whether logging is enabled and if so, adds the RDF message to the RDF graph. Message URIs are created using the address of the XMPP server, ensuring uniqueness.
- User requests: the *RequestExecutor* interface allows the user to, for example, search the RDF graph. 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, 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.

The Nabu Bot uses the Nabu protocol for transferring user queries and answers. The commands of the Nabu protocol are encapsulated as the body of chat messages. Other bindings to the XMPP protocol are possible[8] – also implementation of such a binding is quite straightforward.

In the following section, we take a look at the RDF schemas used by Nabu.

3 The Nabu Ontology

This section explains the most important parts of the ontology, i.e., the RDF schema, Nabu uses for logging. It covers the most important classes and properties for representing messages, accounts, and presence changes in RDF. Nabu handles two types of data:

- The actual RDF data that can be queried externally, i.e. logged messages, presence changes and annotations.

[8] For example, Dan Brickley's foaftown proposes an XMPP binding for SPARQL.

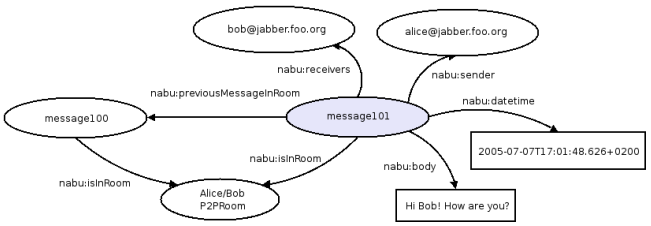

Figure 2: RDF representing an Instant Message.

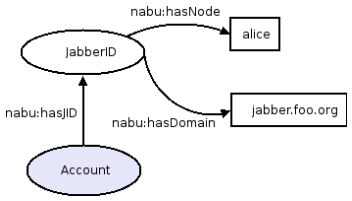

Figure 3: RDF representing an Account.

- Internal data, like account settings (e.g., logging enabled/disabled), and privacy policies. This data cannot be queried from the outside, and the users will never see the RDF representations. It is only indirectly accessible through the requests defined in the Nabu protocol.

In the following sections, the RDF schema of the data that can be queried externally will be discussed. The schemas of the data that cannot be directly queried will be presented in section 5.

3.1 Message

The most important class in the Nabu ontology is *Message*, shown in Figure 2.

- *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[9]. Note that if the sender and receiver are on different servers and each 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 (MUC), 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 the resource part of the participant's Jabber ID is 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.
- *nabu:previousMessageInRoom*: Links to the previous message in the room. This is useful for tracking conversations and for exploring logged conversations with a specific chat partner over time.

Unfortunately, in multi-user chat rooms it is difficult to track what message(s) a user is replying to. In practice, most users prefix their messages with a string denoting the receiver in chatrooms (e.g., "Frank: Please refrain from doing this."). However, Nabu does not address this issue, as other components can add annotation as needed using an heuristic.

[9]http://www.w3.org/TR/xmlschema-2/datatypes.html#dateTime

File transfers and other activities such as video or voice chat are currently not supported. Extending Nabu to implement this functionality and subclassing the Message class accordingly should be trivial.

3.2 Accounts

The *Account* class represents a user account, as shown in Figure 3. Every user account is uniquely represented by a *Jabber ID*, like "alice@jabber.foo.org". For privacy reasons, Nabu does not store person–account associations. If such a mapping is required, one may store such information using FOAF[10], which already includes a *foaf:jabberID* property which allows linking a FOAF:Person to a Jabber ID.

Every account has a Jabber ID representing the account. However, not every Jabber ID represents an account (see MUC rooms), so Jabber IDs and accounts are not identical.

3.3 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 the "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:

```
<P2PRoom rdf:about="&foo;P2PRoom-alice-bob">
  <members rdf:resource="&foo;Account-alice"/>
  <members rdf:resource="&foo;Account-bob"/>
</P2PRoom>
```

In multi-user chat, the semantics of a room are slightly different. While the P2PRoom is a Nabu concept and does not exist in Jabber, the MUC protocol as defined in JEP-0045 [Saint-Andre, 2002] introduces the concept of rooms. A room has its own Jabber ID, just like accounts, e.g., support@conf.foo.org, so unlike P2PRooms, these "MUC-Rooms" are not defined by their members, but by the room name and topic.

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

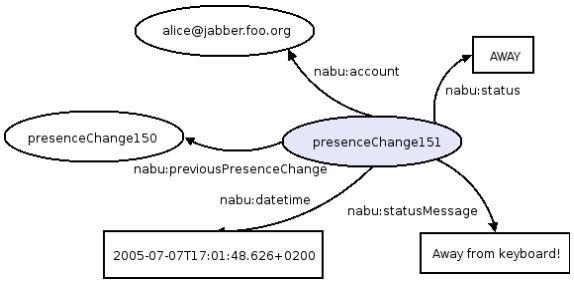

Figure 4: RDF representing Online Presence.

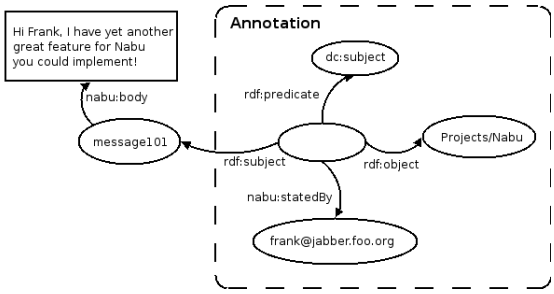

Figure 5: RDF representing an Annotation.

```
<MUCRoom rdf:about="&foo;MUCRoom-foobar">
  <hasJID>
    <JabberID rdf:about="&foo;JabberID-foobar">
       <hasResource/>
       <hasNode>support</hasNode>
       <rdfs:label>support@conf.foo.org</rdfs:label>
       <hasDomain>conf.foo.org</hasDomain>
     </JabberID>
  </hasJID>
</MUCRoom>
```

3.4 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, as shown in Figure 4.

- *nabu:status*: The new presence status.
- *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.

4 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 to file his conversations to facilitate later searching. He could do this manually or could use a text classifier and categorize automatically. Since instant messages are heavily dependent on their context (for example, imagine a user receiving an e-mail with a question and answering via instant messaging), one may also decide to use annotations to link the messages to their context. We will discuss this in sections 7 and 8.

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 Dublin Core[11] and SKOS[12]. We have to stress that Nabu does not restrict the user to annotating messages with concepts – it is perfectly possible to link arbitrary RDF constructs to chat messages. This way Nabu is flexible enough to allow tagging messages with context information such as “The user was looking at the DFKI website when sending this message”. Using Nabu’s observation features (see section 7), software running on the user’s machine can automatically annotate new messages when they arrive with information the annotation software has access to. Also, it is possible to create semantic links between messages: for example, it is possible to implement a more complex heuristic for determining the reply–chain of a message (see section 3.1 for an explanation on why this is not trivial). Adding this information to messages does not require extending Nabu – one can also write a client–side component that uses message annotations for adding this information instead.

As an example of a simple manual annotation, let us classify a (fictional) message: ”Hi Frank, I have yet another great feature for Nabu you could implement“. To make searching easier, we want to specify the project the message is related to, in this case Nabu.

The CREATESTATEMENT request adds annotations to a message:

```
CREATESTATEMENT RESOURCE
  http://&foo;Message-101
  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 shown in Figure 5.

The annotations are *reified*, which means that each 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 take them into consideration, but ignore Bob’s statements.

Nearly every kind of RDF statement can be added. The only restriction is that properties from the Nabu ontology are

[11] http://dublincore.org/

[12] http://www.w3.org/2004/02/skos/

not allowed for user statements. E.g., *dc:subject* (dc = Dublin Core[13]) is valid, but *nabu:isInRoom* is not, because the predicate *nabu:isInRoom* is part of the Nabu ontology. This prevents users from currupting (deliberately or not) the Nabu archive or compromising privacy settings. Properties from the Nabu ontology are managed by the server and can only be modified indirectly by commands of the Nabu protocol.

5 Log Sharing and Privacy Settings

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

1. Leave the user in full control over what is logged.
2. Allow users to delete sensitive information at any time.
3. Allow access to the archive only through a clearly defined interface that handles authentication and respects the privacy settings.
4. Implement conservative default settings (i.e., disable logging, use restrictive privacy settings)

On the other hand, one of Nabu's goals is to encourage sharing between peers to make valuable information available to others when wanted. Therefore a privacy model is needed that supports both ensuring privacy and allows sharing of conversation logs.

In Nabu, every user is the owner of the messages he has sent, and he can control who can read his messages or delete them later if he wants. This means that a message is under control of the message sender only. If two users have a conversation, each 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, and every user has a list of policies he can assign to logged messages. There is always exactly one policy active at any one 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, if 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? If a message (or any resource in general) is not accessible, this means the resource itself and its concise bounded description[14] is completely hidden from the user: The resource itself and all links to or from the resource are hidden. When querying the model, the resource does not show up in the results. For messages this means that neither the message content nor any links to the message are visible. This includes annotations: If an annotation was added to link the message to a category, this 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.

5.1 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 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. That is, a policy without any rules denies all accesses (except to the policy owner).

If both rules exist that allow and deny access to an account, the deny-rule takes precedence and the access is denied.

5.2 Groups

As mentioned before, access permissions can be set not only 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 this group. Instead of allowing access per account, he can do a simple "allowGroup friends" and all accounts in the group gain access.

5.3 Examples

Here we present some examples demonstrating how privacy policies can be 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 we use Alice instead of http://foo/Accounts/jabber.foo.org/alice for clarity here.

The following policy allows access to Alice (as she is the owner), Daniel and Bob.

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

The following policy 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 (nor can the rest of the world).

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

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

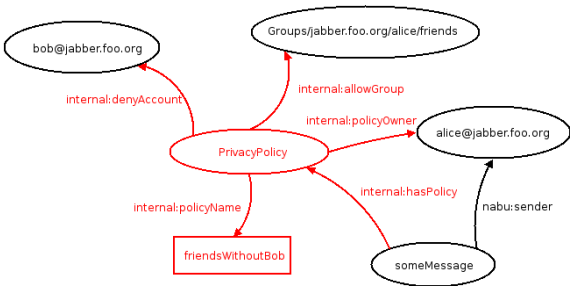

Figure 6: A Privacy Policy.

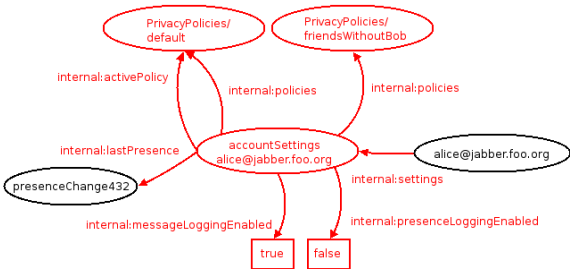

Figure 7: RDF representing Account Settings.

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

In the following example, the first directive allows access to the *friends* group, i.e. Bob and Emily, but as the second directive denies access to Bob explicitely, only Emily has access (and Alice of course).

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

Internally, privacy policies are realized using the the RDF shown in Figure 6. The parts of the RDF shown in black can be queried using Nabu's query features. The other parts can only be retrieved and manipulated using Nabu commands.

This graph shows the last policy example. The policy *friendsWithoutBob* is owned by Alice. It allows access to her *friends* group, but denies it for 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.

5.4 Account Settings

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

This example graph in Figure 7 shows the AccountSettings instance of alice@jabber.foo.org. It has the following properties:

- *internal:messageLoggingEnabled* and *internal:presenceLoggingEnabled*: Store whether message and 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.
- *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.

6 Querying the Archive

Once Nabu logs a user's conversations, the user probably wants to search them at some point. For querying the archive, Nabu uses the SPARQL query language [Eric Prud'hommeaux, 2005]. SPARQL is a language for querying RDF stores, similar to SQL. Being powerful and versatile, it allows arbitrarily complex queries. Unfortunately it's also quite complex for everyday use, so a GUI for the most common queries would be desirable.

Example: One wants to search for all messages containing "Nabu". The following command performs this search:

```
QUERY SPARQL
DESCRIBE ?msg
WHERE { ?msg nabu:body ?body .
  FILTER REGEX(?body, "Nabu", "i") }
```

The query returns all messages ?msg that have a body ?body matching the regular expression "Nabu" ("i" makes the search case-insensitive). For simple string searches, there is also a shortcut available in Nabu in the form of the "QUERY SEARCHMSG" command.

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=...
   <Message rdf:about=
     "&foo;Message-094210.520">
   <body>Me thinks, Nabu rocks big time!</body>
     <previousMessageInRoom rdf:resource=
       "&foo;Message-143802.712"/>
     <inRoom rdf:resource=
       "&foo;P2PRoom-frank2/"/>
      <subject/>
      <messageType>chat</messageType>
      <streamID/>
      <sender rdf:resource=
         "&foo;Account-frank2"/>
      <receivers rdf:resource=
         "&foo;Account-frank"/>
      <datetime>2005-07-14T...</datetime>
  </Message>
</rdf:RDF>
```

Note: Nabu returns only RDF data that has been declared as accessible. By default, this includes all messages that have been sent or received by the user who issues the query. Normally he won't see messages exchanged between other users.

If a user decides to, he can grant others access to a conversation log (for example, co-workers might decide to share the log of an online meeting with other team members).

7 User Observation

One topic addressed in the research project EPOS [Dengel *et al.*, 2002] is user observation: By observing the user's actions, EPOS tries to identify the context of the desktop in order to support the user in his work [Schwarz, 2005]. 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 implements this using an *assistant bar*, which is a desktop panel listing relevant contacts, resources, and projects.

Collecting observation data is done by plugins for the user's applications, 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.

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 time. Observing applications need the observed password of the account in order to register at the server.

Usually observation will be integrated into the context framework by using the client API that comes with Nabu.

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, this can also be sent manually to the bot). This registers 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[15].

8 Applications

Numerous applications can be realized with the techniques presented. Let us enumerate some of these.

- *A message archive* – This is Nabu's most obvious application. As Nabu is a server-side component, similar to IMAP, where users may access their messages in the central archive from anywhere. Also, no inconsistencies can occur.
- *A semantic store* – As items stored in Nabu can be annotated, messages bear not only syntax but also semantics.
- *A powerful message search platform* – Using SPARQL, the archive supports both fulltext search and semantic search exploiting the relations specified in the message's annotations.
- *An exchange platform for information* – As a user's Nabu repository features fine-grained access control, other users may be granted access to a user's messages and message annotations, extending the other user's knowledge repository.
- *A gateway for integrating instant messages to your personal information model* – Nabu enables any RDF-capable software to access the user's instant messages. This way, instant messages can be integrated into the user's personal information model in frameworks such as Gnowsis [Sauermann and Schwarz, 2004].
- *A personal semantic knowledge base* – Nabu is not only about instant messages. It can store anything that may be represented in RDF. Together with Nabu access control and the access mechanisms provided by the XMPP protocol, a simple but powerful personal shareable semantic knowledge base arises. As Nabu is intended to run on a server that is continuously available, this solves problems with spurious availability of data in case the repository is implemented on the user's machine. Also, most technical problems related to reachability due to firewalls or network address translation scenarios do not occur in this approach.

9 Conclusion and Further Work

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 into 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 one needs more fine-grained control, like hiding only certain properties, it works well for Nabu.

The concepts were implemented as an extension for the Jive XMPP server. This proof-of-concept implementation is available[16] 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 [Dengel *et al.*, 2002] project. The user observation functionality has been successfully integrated into the context elicitation.

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

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

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

- Nabu's user interface is currently text-based. This is flexible because it can be used on any platform and with any 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 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 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.
- Currently Nabu is only accessible via the XMPP protocol. In order to make the repository available to software without requiring an XMPP library, it should be made possible to query the archive using HTTP(S)/XML–RPC/SOAP protocols.

Acknowledgments

This work has been supported by a grant from The Federal Ministry of Education, Science, Research, and Technology (FKZ ITW–01 IW C01).

References

[Andy Seaborne et al., 2005] Andy Seaborne et al. Jena Semantic Web Framework, 2005.

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

[Dengel *et al.*, 2002] Andreas Dengel, Andreas Abecker, Jan-Thies Bähr, Ansgar Bernardi, Peter Dannenmann, Ludger van Elst, Stefan Klink, Heiko Maus, Sven Schwarz, and Michael Sintek. Evolving Personal to Organizational Knowledge Spaces. Project Proposal, DFKI GmbH Kaiserslautern, 2002.

[Dennis Quan and Karger, 2003] David Huynh Dennis Quan and David R. Karger. Haystack: A platform for authoring end user semantic web applications. In *International Semantic Web Conference*, pages 738–753, 2003.

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

[Eric Prud'hommeaux, 2005] Andy Seaborne (edts) Eric Prud'hommeaux. Sparql query language for rdf. W3c working draft, W3C, 2005.

[Karneges and Paterson, 2004] Justin Karneges and Ian Paterson. JEP-0136: Message Archiving. Jabber Enhancement Proposal, 2004. URL `http://www.jabber.org/jeps/jep-0136.html`.

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

[Saint-Andre, 2002] Peter Saint-Andre. JEP-0045: Multi-User Chat. Jabber Enhancement Proposal, 2002. URL `http://www.jabber.org/jeps/jep-0045.html`.

[Sauermann and Schwarz, 2004] Leo Sauermann and Sven Schwarz. Introducing the gnowsis semantic desktop. In *Proceedings of the International Semantic Web Conference 2004*, 2004.

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