Supporting workgroups collaborating via email using the semantic web and RDF[1]

Olu Ibidunni
Information Infrastructure Laboratory
HP Laboratories Bristol
HPL-2002-316
November 27th , 2002*

E-mail: Andy_Seaborne@hpl.hp.com

RDF, semantic web, email, collaborative processes

One of the promises of the semantic web is that machines will be able to help people with routine information-oriented tasks such as finding, collating and low-level-processing information. In this report, we look at the application of RDF technology to one such area - W3C working group processes. Because email is central to such groups, we provide automation of tasks such as action management through email. We also provide RDF-based searching to help manage the information overload.

* Internal Accession Date Only Approved for External Publication
[1] Enquiries concerning this report should be directed to Andy Seaborne email: Andy_Seaborne@hpl.hp.com

Technical report

By Olu Ibidunni

Introduction

This report was written for HP documenting issues related to the development of a semantic web application. In this report the main system discussed is an email bot, part of a meeting assistant. The email bot is covered in three parts and issues that affect both of its parts are discussed in detail in the first section. More specific issues are then discussed in the other two sections.

The Project

Overview

To build a semantic web application with sound architecture, highlighting the issues raised. These issues were found to occur in two main areas:

- From the technology itself: Issues related to the semantic web concept and the functions of a semantic web application and
- From the development tools used: Issues related to the technology used to implement the semantic web application. In this project RDF technology was used with the Jena Java API.

Semantic Web technology

The information currently available on the Internet provides a wonderful pool of information to people all over the world. The impact it has had on our everyday lives cannot be under-estimated. The semantic web takes this one step further providing enriched data that is not only meaningful to human beings but also to computers. This provides numerous possibilities for advanced features in existing systems and the automation of processes currently done manually. A good example of the latter is the filling-in of personal information forms on the web. The semantic web will allow people to point to their representation in future. In this project building a semantic web application meant we could explore the additional benefits of a semantic web application.

Resource Description Framework - RDF

In RDF, data is represented by graphs and the nodes of the graphs are either resources or literals. A resource is anything we are trying to represent and literals contain actual data. A resource can have a URI (Uniform Resource Identifier) that uniquely identifies the resource. An example would be an RDF graph to represent myself on the Internet. I will be represented by a resource that has a URI and this resource would have properties such as name, address and telephone number. The values of my properties are literals, so the literal for my name property is 'Olu'. This is a simple case but in more complex cases resources are linked to other resources. An example here is where I wanted to represent the fact that my name has two parts, surname and first name. To do this I would have a blank node that has both of these literals coming off it. In this project all the entities used are represented using RDF with the resulting semantic web application used to manage the entities. RDF is less rigid than a normal database with the benefit of merging graphs. It is also more meaningful due to the data structure defined by the schema. For more information on RDF see the W3C web site `http://www.w3.org/TR/NOTE-rdf-simple-intro`

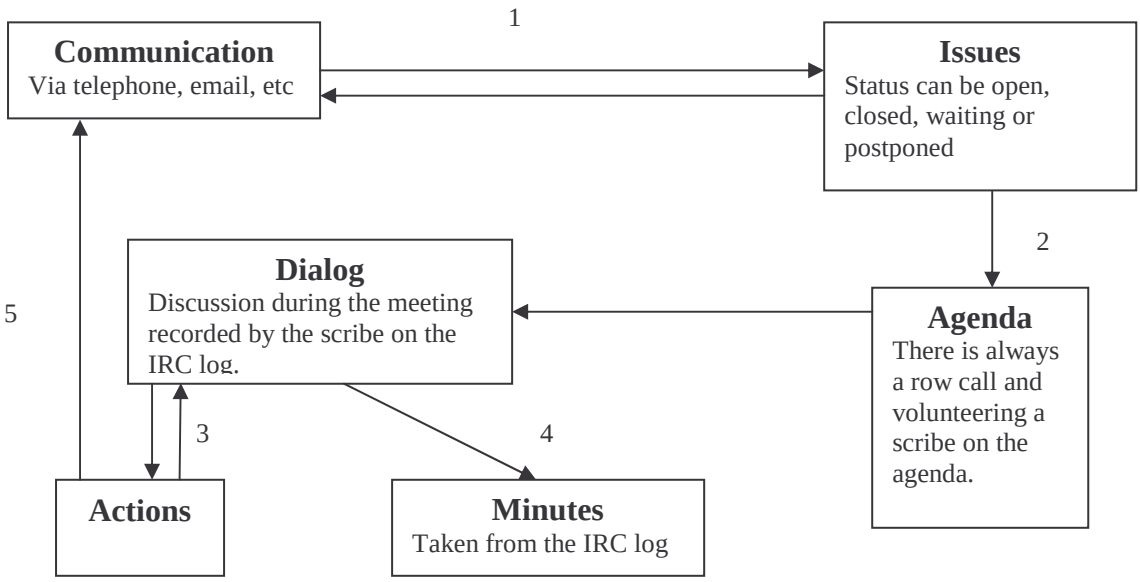
Jena

Jena is a Java API that can be used to create and manipulate RDF graphs and has java classes to represent graphs, resources, properties and literals. The API provides convenience methods for application developers so they do not need to make low-level calls. The interfaces representing resources, properties and literals are called Resource, Property and Literal respectively and a graph is called a model, represented by the Model interface. In this project all the RDF instance data is created via Jena and statement and resource iterators are used to access the instance data in the models. For more information see the Jena tutorials at `http://www.hpl.hp.com/semweb/doc/tutorial/index.html`

Working groups

The application was built for W3C working groups. The working groups have a formal process for resolving issues with the aim of creating a standard. Part of this process is the regular meetings via telecom that take place once a week. During the discussion at these meetings and elsewhere issues related to the working group standard are resolved by its members. Meetings always follow a set

agenda and the chairman uses minutes from the meeting to bring documentation up to date. The document outlining the formal process of W3C working groups can be found at http://www.w3.org/Consortium/Process-20010208/



1. The chairman makes the decision on what topics become an issue from the discussions that take place. Both members and non-members of the working group can raise the issues. Issues are also discussed, with the aim of resolving them.
2. The chairman produces the agenda based on the unresolved issues
3. During the meeting people are assigned actions based on the issues being discussed and get mentioned in the IRC log
4. The minutes are produced from the IRC log of the meeting
5. Actions are discussed off-line via email and other means of communication.

Fig1. Overview of the working group data flow in RDFCore working group

The meeting assistant

The tools in place for assisting with the working group information process are quite limited and left a lot of room for automation using the semantic web. In particular a lot of communication in the working groups takes place via email as members of the working group are often spread around the world. The idea was that this flow of information could be used in a number of ways

- To control the activities of the working group
- To inform members of the working group
- To resolve issues

The project was to build a meeting assistant using RDF. The aim was that the meeting assistant would reduce the amount of work the chairman and members of the working group had to do. The meeting assistant is made up of a number of bots, each bot using a particular existing technology. These bots would be used to manage entities with the system like actions, people and agendas. Some of the bots envisaged include the IRC bot, the web front, the email bot and so on. The author of this report designed and built the email bot.

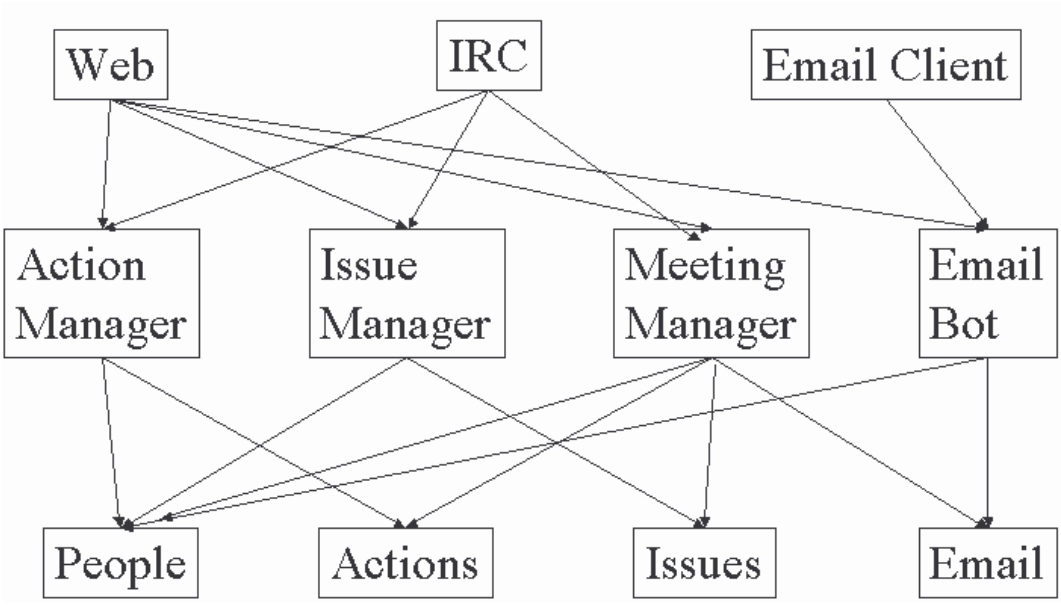


Fig2. Overview of the meeting assistant

The Email bot

Overview

The email bot was built to enhance the functionality of the email archive that stores emails posted the working group mailing list. The aim was to provide functionality for managing other entities in the overall meeting assistant such as actions, issues and agendas and the bot comes in two main parts

- A polling program and
- A search engine.

Architecture

The system has been broken down into three layers

- Data layer where the RDF is stored in a relational database. Also in this layer the cache sits on top of the database to provide faster data access.
- Logic layer is where data is processed through the Jena API. The Jena calls are made from classes that have methods to provide easy access to data and avoid repetition.
- Presentation layer provides the interface to the user with the HTML forms for the search engine and the command syntax for the polling program.

Although other bots in the overall meeting assistant also implement a three layer architecture there is a fundamental difference in the ways the architecture is used in the emailbot. The user interface is not the sole entry point into the system. The user interface can be used to create entities like actions the same way the web front-end would be used but emails are monitored and stored automatically. Another key difference is that the bot will be used to manage entities created by other entities like agendas and will require indirect access to them.

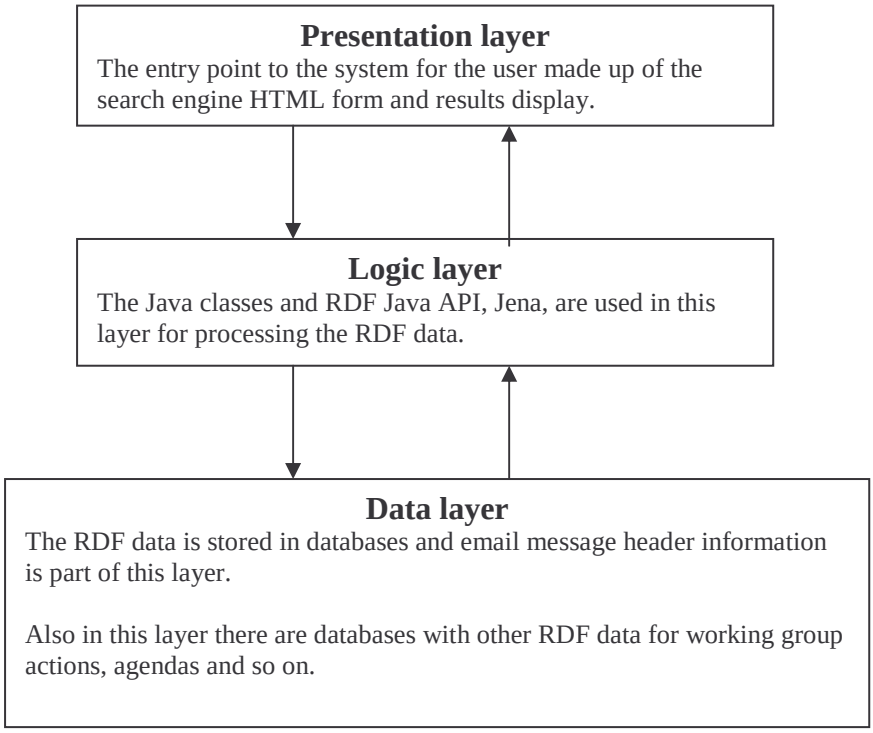


Fig3. Overview of system architecture

The Email Schema
The current email schema is based on the RFC822 specification for email headers. It is a flat structure and can be found in the file EMAIL.RDF. Extensions to the schema will only be compatible if the current fields remain present in the schema. This is because the system is based on the schema and has the properties hard coded into it.

The main program
In the main program there are two separate threads, one for the web server and the other for the polling program. The program reads in a configuration file written in RDF that has the following information

- Working groups to monitor
- The connections between working groups, mailing lists and the main databases

Caching
In Jena accessing a remote model is a time expensive process because each time the system retrieves a resource there are multiple calls to the database. This is due to the fine grain access Jena provides where equivalent of database select statements are used to access properties. Also, data access over the network slow the process down even further. Memory access on the other hand is very quick and to avoid unnecessary delays, the system works purely with memory models. There is a cache interface to the database used by both the search engine and the polling program. This cache stores all the data in the relational database. When the system starts up everything in the database is immediately copied to the cache. This cache has the potential of getting very large and needs a good management algorithm. At present there isn't one but even if there is swapping to disk it is still faster than accessing the database directly. The virtual memory manager that allows the system to use more memory than is physically available moves the data between disk and memory in fixed amounts called pages.

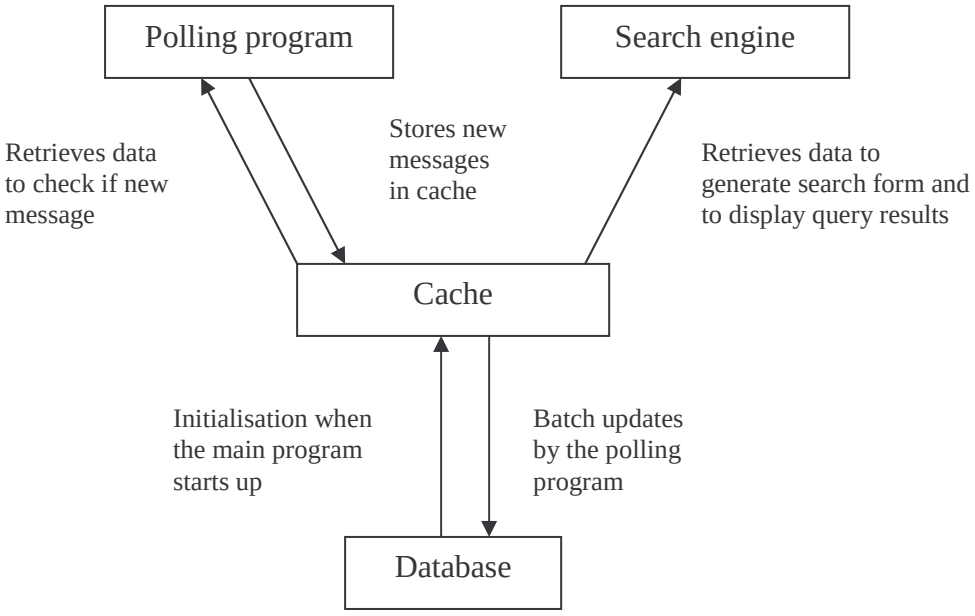


Fig4. Overview of system memory management

The search engine

Overview

The search engine provides a way for the user to access the information in the database that has the email header information. In many systems explored the original archives provide a means of browsing the data and in certain cases also have a search engine. Some archives did not have a search engine and others had a search engine with limited capabilities. Some of the deficiencies include

- No searching across mailing lists
- Unable to perform combined searches using dates, people information and email information
- Tedious user interface where it would be easier for the user to select information from a menu

Search engine form

It would be favourable to generate the form dynamically but this is outside the scope of this project. For the unique fields there is the option of selecting from a menu or entering the data into a text box. In very large databases menus are no longer an option, as they get too long. One way of getting around this would be to display in the menus values from the mailing lists the user has selected and not all the values in the database.

Sub-string can be used for the sender name and email address. Using the exact string for these values poses problems because for the same person their name may be formatted differently depending on the list. This means an exact string search across different lists will return messages from just one list, not having the desired effect of returning all the person's emails from the different lists. For this reason menus are not provided for these fields.

Displaying the results

The results display the headers of emails that match the search criteria specified. The URI of each message displayed is a link to the email in the source archive. This link allows the user to view the body of the email.

Speed

To speed up the query process a number of caches were explored including:

- Caching all the data in the relational database
- Caching previous query results and
- Caching only the message Ids in the database

When the cache does not have all the data in the database, when it only stores previous query results, it poses limitations for searching. For the search engine it is beneficial only with queries based on the unique fields, message ID and URI. For combined searches it is likely to slow the query process down for the following reasons:

- For queries over a range of values the engine would need to search the cache and then search the database to ensure it has found all the results
- When checking the database for the remaining results after searching the cache, the engine will need to keep track of results it has already seen.

Multiple mailing lists

The ability to search across many different mailing lists gives the system extra flexibility but can lead to longer delays. To avoid this the mailing list parameters selected by the user are used to retrieve iterators over the messages in just those lists and this reduces the search area. When unique fields are provided the search engine will stop once it finds a matching field. With combined searches the engine has to look at all the emails in the lists specified to see if they meet the criteria.

The polling program

Overview

The polling program performs two main functions

- Detects new emails and adds them to the system
- Listens for commands from the user and performs the task

Configuration file

The polling program can monitor several mailing lists simultaneously. When the poller receives a command from a mailing list it needs to access the correct working group data. This will involve accessing different databases each storing the working group entities such as actions, agendas and so on. The configuration file specifies where the main working group database can be found, providing an entry point into the RDF world.

The configuration file is written in RDF and specifies the working groups that the polling program polls. All the mailing lists associated with the working groups are polled. Groups can have more than one mailing list. The configuration file also links to the main database for each working group. The configuration file and other entities are accessed as RDF objects using the RDF object server (1). There can be links between RDF objects and this allows data to be spread over many databases but navigated as if there was just one database.

Using a configuration file makes the system a lot more flexible allowing the user to add and remove new lists by modifying the configurations file and rebooting the system.

New Emails

The polling program retrieves the index file of each mail archive and extracts all the message links. To extract these links the system looks for specific HTML tags that identify the links in the index page. The message links point to a HTML file with the message header and body. For each message the header information is scraped from the HTML file and stored in the RDF database. The RDF is created using JENA calls made from the message database class (MessageDB.java) in the method (addMsg) that adds new messages. The method takes message URI as a parameter used to retrieve the message HTML file.

To determine whether a message is new the system takes all the links in the index pages and checks whether the ID of each message is already in the database. This process is comparatively slow but ensures that messages are not added to the database twice. An alternative would be to maintain a pointer to the last email seen in the index file and to add any emails after the pointer to the database. The problem here is that in an archive were messages can be deleted, the system has to makes sure that the pointer is pointing to the correct email. Alternatives approaches for retrieving new emails would be to pull new emails by subscribing to the list or to push new emails by intercepting the messages before they are archived.

Pulling new emails

Here the bots subscribes to the list and receives any new messages directly. The advantage of this approach would be that any emails received by the system are guaranteed to be new emails. The time spent checking whether emails are already in the database would be avoided.

Pushing new emails

Here the emails would be detected sooner as the delay in the present system were the archive is updated would no longer exist. Intercepting messages before they are archived would also be a lot neater as the email headers will be in their purest from, not altered by the archive. This would remove the need to tailor the different archive formats to a standard format the system recognises. At this point the emails will not have URIs and so the emailbot will need to go back and retrieve this later in batches to provide access to the body of the message. Or an alternative would be for the system to work independently and store the email header and body. This removes the bots dependency on the archive entirely, making the system more stable and reliable.

Command syntax

Before storing new messages in the database the bot checks to see if there is a command to execute. The commands are written as the subject of the email but in cases where there are a number of

parameters such as adding references, it is better to have the commands in the body of the email. There are a number of issues that must be considered here. If the command is written in the body of the email it gives the user the flexibility of choosing their own email subject but the bot command, it will not have the same informative value to subscribers. This is because when a command is written in the subject of an email and performed successfully, everyone is aware of the change immediately.

A combination of the two approaches maybe best where the user can specify the command in the subject if they want other people to know about it or specify it in the body of the email if not. When specifying it in the body of the email there should be syntax to indicate that the bot should check the body of the email for a command. This will avoid the bot having to check the body of every new email when only very few of them will contain commands. This syntax should be in the subject of the email but still allow the user to specify his or her own subject alongside it. Overall the syntax must be concise and easy to remember. The order of the command parameters is very important and inflexible. This can be refined to a more user-friendly syntax by putting less emphasis on the order of parameters.

NB: Outlook introduces a space in the subject after every 64 characters in the email subject. In commands with several parameters this can cause a problem. The current system requires the user to put quotes round the command arguments but if the space is introduces into an email address it will cause problems. The solution would be to manually strip out these spaces. Other email applications may have such bugs and it is worth checking first.

Storing new messages
The main cache is updated directly by the poller at regular intervals. The database is updated from the cache in batches also by the poller. To improve efficiency a Jena call that retrieves all the IDs in the database is made using a selector. This reduces the time it takes to initialise the cache. When new emails are detected the poller updates the cache memory model, adding the new emails header information. A batch system is used for updating the database because database access slows the system down tremendously. The cache is updated directly and every so often the poller updates the database using the up to date cache, again using selectors for speed. In the main program there is a final update of the database when the user quits the system so emails in the cache are not detected twice.

Timing the poller
The time delay before checking new emails and the number of these cycles that make up a batch to update the database can be changed. It is worth noting that increasing the delay on the poller means it will detect more emails when it checks for new messages. Regardless of the traffic it is best not to poll the lists too often as it checks all the emails in the cache each time and this takes up processing power. The same rule applies to specifying the number of cycles in a batch because again each time it checks all the messages in the database and this is even slower than memory access. In future it would be good to allow the user to specify the timing of the system. Overall the system is fast and the delays are not very significant. After implementing a memory model even the database updates are quick.

Processing user commands
When the bot receives a user command it performs the task by calling the appropriate handler. The chain design pattern was implemented where incoming commands are passed along a chain of different handlers. String pattern matching is used to determine the appropriate handler. The chain design pattern works in this situation because there is only one handler for each command. If the system were extended to have general handlers that apply to all the commands this architecture would no longer work. A situation where more than one handler would be needed to process a request is one where there is a logger that records all the commands passed to the bot and their outcome, as well as a handler to process the specific command given.

Adding new handlers
Due to the nature of a meeting assistant there is a need to add new commands to the bot. Different working group meetings will have variations of commands that will be useful in the tool. The ability to add these commands easily is vital. The process for adding a new handler in the current system is to write a class that implements ProcessRequest and extends Handler. The new handler must then be added to the chain of handlers in the polling program. The Handler class contains methods used by

more than one of the handlers such as a method that returns an action resource when passed the action ID or URI. In the current system actions no longer have URIs but for entities that have Ids and URIs we found that it is convenient to have access to such entities via their ID or URI without distinguishing between the two.

Adding new adaptors
Different archives format the index file in different ways and because the polling program is looking for specific patterns it cannot work with these variations. At present the system only works with the lists at http://dbanks.hpl.hp.com/hypermail/ but there are several Hypermail archives and this gives rise to the need for adaptors. A proposed architecture is one where each archive has an adaptor to convert its index page to a standard format that the bot recognises. This means all the index files received by the bot will be consistent allowing it to perform the rest of it's functions as normal with any archive. For this there is the need for a standard bot index file format. This index file format only needs to cater for message links in the original archive index file because that is all the polling program uses.

Redundant information
There is a danger with this system that email sent to the bot will clog the mailing lists with mechanical data that is not useful to the subscribers of the list. To avoid this the aim was to keep the flow of information on the lists the same and to have a system that utilised this information. An example here is the regret command, the message would have been sent to the list anyway but is now automatically processed by the bot. So here the bot is making full use of the information already available and is not adding any additional information to the list.

In situations where the bots introduces new information to the list, the information must be beneficial to the subscribers. It can do this by informing subscribers of changes to the working group database. An example here would be when the state of an action is changed from active to complete via a web interface. In this case the bot will change the status of the action and people on the list will be informed of the change simultaneously. This gives members of the group a chance to review the completed action before the next meeting. It would be wrong to send the command itself to the list because if it is not completed successfully it will be misleading. Only the successful commands should be put on the list. To do this, such commands should to sent to the bot directly and not to the list. Then once the bot has successfully completed the command it can send the feedback to the list. If it is not completed successfully the error message should be sent to the person that gave the command originally.

There are certain changes that require consensus and this cannot be done via email but a proposal is to break such commands down into two steps. For example if someone is asked to write a document they can use the email bot to inform others that they have completed the document. Once informed people can read the document and vote on the issue at the next meeting.

There are also commands that involve just one person. For example a person can request a list of his/her uncompleted actions. Such commands should be sent to the bot and the reply should be sent back to the person directly. Putting such commands and it's response on the list would not be of any benefit to the other subscribers.

Illegal arguments
The bots does not have any form of error checking or validation. If the user sends a command that it does not recognise it will ignore the command and store the message in the database.

Conclusion

The email bot still needs more work before working groups can use it but the project has raised a number of issues.

Semantic web applications

A semantic web gives the meeting assistant application the ability to do things a normal meeting assistant application could not do. Some of the observations from this project are:

- Instance data once semantically correct can be used and interpreted by any other semantic web application. The only issue here is what the correct representation of the data actually is. This caused a lot of debate amongst the group and on many occasions there was more than one correct solution. Also in cases where an absolute representation was difficult, a representation suiting the desired application was used. This limits the use of the data being represented and this can in turn lead to needless repetition. Adequate time most be given for deriving absolute data representation and if minimal representation are used, time should be given to make use they are extensible.
- Extending the schemas to represent more information is possible without destroying the existing system functionality. Applications that are based on plug-in schemas are the most desirable but even in systems that have the schema as an integral part, the schemas can be extended. Advanced features can then be based on the additional elements of the schema.

More work should have been done on integrating our work with existing system. The power of the semantic web is that different system will be able to talk to each other by interpreting the same data. Standalone systems are not making full use of the potential the semantic web has. In future fully exploring both semantic web and normal applications already out there before embarking on the project would be useful for this. There is also the need for sound system architecture in semantic web applications that make it possible to add on any new features and there may be variations within the system to cater for. In the email bot there were variations in the email archives it was intended to work with.

Technology used

To a novice RDF in any format is unreadable and there needs to be application for creating RDF used by applications. This became apparent with the configuration file for the polling program. Also the project did not fully explore the ability to merge different graphs in RDF. Jena methods for accessing information in the models were used extensively in the project. They provided quick data access via methods returning iterators over resources and statement using selectors. Generally in Jena there was the need for a lot of type cast, making coding inconvenient at times. Data access via java classes that made Jena calls seemed to slow the system down considerably. In the message database class the iterators where can created in the constructor and used in the search servlet form generation but this took a lot of time and the system was changed to used selectors instead speeding it up considerably. In developing this application there was the need for classes representing messages and the messages database. Jena does not cater for this and a Java class was written to do this instead. The object server was also used, providing access to other entities like action and so on. Because Jena does not have access to the model in anyway apart from it statements, resources and properties, writing code at time was unnecessarily long and repetitive. The database model access in Jena is extremely slow and no good for application development. There is a desperate need to make data access a lot quicker.

Other bots

There were several similarities between the different bots that made up the meeting assistant and this means there is potential for compatible system architecture across them. Some of the similarities found were

- The web front end and the email bot are both being used to create entities like actions. There is the possibility for a generic interface for specifying tasks for each bot to perform.
- The IRC bot and the email bot both had to be aware of the context they were working in. The IRC bot needs to know the current item in the agenda being discussed in the meeting and the email bot need to know the working group the mailing list it received a command from is associated with.
- In all the bots there is the need for handlers that perform different tasks. These handlers can be shared by the different bots. The way the handlers are used in the bots will vary because

at present the email bot implements a chain of handlers while the IRC would probably need to iterate through the handler as one than one handler will apply. For example in the IRC bot each command needs to be recorded for the log as well as executed.

Development process

The approach used to develop the system was one where prototypes are continuously being developed and refined. This allowed the developers to

- Spot potential problems early in the development process
- Get feedback from the users, used to refined the next phase of development
- Break down the system into modules that where developed separately

Future work

Email schema
Extending the email schema to represent the fact that emails belong to the same mailing list. This would require a mailing list resource. In the current system all the emails are disjoint. Also in future a more plug-in architecture for the schema would provide greater flexibility, where the properties are dynamically generated from the schema the system is working with.

Java classes and methods
An ‘is method’ in the logic layer to check the value of the fields may prove useful and avoid repetition.

Additional features
There are many additional features that the system should incorporate. These include:

- Send people reminders of there uncompleted actions
- List actions with the user specifying the type of action to list
- Undo facility

Threading
Work needs to be done on providing correct and consistent threading information to the user. At present the system does not provide any threading information because this information is not always available from the original archives.

Optimisation
There is a lot of looping in the system. Cutting down the number of loops is directly related to reducing the amount of time it will take to process requests.

Error checking and validation
There is no error checking or validation of the user’s input at all in the current system. Also, feedback from the object server when executing user commands should be sent back to the user for them to take any necessary action. It is important to realise how easy it is to make a mistake when using the bot. Even though the command syntax is fairly concise, URIs and IDs are not at the best of times. In the present system commands are sent to the list, which can be misleading if the corresponding task is not completed due to an error.

Synchronisation
The tasks specified need to be synchronised so the user isn’t trying to update and action that hasn’t been created due to a delay in the email system. At present the email archive protects the bot against this as email are stored in the archive in time order. If the bot access email before the archive does in future, this will be essential.

Cache management
All the messages in the database are put into the cache and in a large system this approach is not practical. There must to be an algorithm for memory management. One way of doing this would be to have a finite amount of memory allocated to the cache and to fill it with the most recent emails in the database regardless of the list they originated from. This approach would not have any impact on lists that have low email traffic but are searched frequently. A way round this latter problem is to have a separate cache for the different lists.

Help facility
The user needs an easy and flexible way and accessing information on the commands available. A proposal for this is a command requesting help that returns a list of all the commands directly to the user.

Data access
Multiple accesses to the data can be very dangerous and the current system does not cater for this occurrence.

The system also needs to know when things have changed in the object server via another bot. The object server could notify the bot of any changes or the bot could check it see whether it's view was update before making any changes.

Reliability
As discussed earlier the system is dependant on a working email archive. To increase reliability this dependency should be removed.

Speed
In a fully functioning system the machine it is currently running on will need upgrading.

Demonstration and feedback
It would best if the system where used by exiting workings so there feedback can be used to enhance the system further. To do this the configuration file will have to include their working groups so the bot will monitor their list and could use the search engine and give commands to the bot. For this project to work the group chairman and one or more of it's members must agree to use the facilities available and give feedback at regular meetings with the developers. Potential demonstration users include Sweb-apps contact Andy Seaborne, RDFCore contact Brian McBride and WebOnt contact Jeremy Carroll.

Appendix

Setting up the email bot

1. Install Jetty 4.0 (3)
2. Install MySQL 3.23 (4)
3. Copy the jar file emailbot.jar (4) and rdfobject.jar (1)
4. Set the classpath to include the jar files
5. Create a database for storing email messages
6. Create the polling program configuration file

Running the email bot

1. Run the MySQL server
2. Run the object server (1)
3. To start the email bot type java code.boot passing it the parameters
    a. The URL of the email message database
    b. The URL of the configuration file database sever
    c. The URI of the configuration file database
    d. The URI of the polling program configuration file

Example: >java code.Boot jdbc:mysql://sweb-prj-6:3306/email http://sweb-prj-2:9003 urn:db:poller urn:olu:poller

Commands available

All subjects that have commands for the bot must start with 'emailbot:'. The next parameter is the name of the command, followed by the arguments for the command. All the command each of the command arguments should be in between single quotes.

Example of email subject specifying command to the email bot

emailbot: CreateAction 'Harrods' 'olu_ibidunni@hpl.hp.com' 'Olu goes shopping' '03-07-02'

This command will create a new action with the ID Harrods, the actor Olu (identified in the syntax by her email address), the short description 'Olu goes shopping' and the java date 03-07-02.

The following commands have been implemented

1. CreateAction <action ID> <recognised email address of the person assigned to> <short description> <date dd-mm-yy (optional)> to create a new action. If a date is not specified the current date will be used. Remember this current date is the date the email is processed and not the date the email was sent. If the email was sent but the system is not run these dates could be very different. The best thing is to specify the date if the system is not running at the time the email is being sent. The recognised email address of the actor is the address that can be used to retrieve their actor resource in the actor database.

2. ChangeID <old ID> <new ID> to change the ID of an existing action

3. ChangeState <action ID> <new state> to change the state of an existing action. The state transition machine of the action defines the transitions that can be made from one state to the next.

4. AddRefernce <action ID> <any number of references separated by spaces> to add new references to any existing action

5. AddComment <action ID> <any number of comments separated by spaces> to add new comments to an existing action

6. Regret <actor email address> adds a statement to the meeting graph to says actor not attending next meeting

Java classes and Javadocs

The message database class
This class provides access to the message database and useful iterators such as iterators over all the message ids, all the messages in a list, all the mailing lists and so on. The constrictor takes a model as it's parameter and this means it can work both with memory and relational database models. The iterators are used both by the polling program and the search engine. There is also a method for adding new messages that makes use of the methods in the message class. The code for this class is in

`\\0-data-2\shared\Labs\Bristol\PSS\DMSD\Semantic Web\Applications\project\email\code\MessageDB.java`

Message class
This class provides access to the message header fields with a set and get method for each field. The code for this class is in

`\\0-data-2\shared\Labs\Bristol\PSS\DMSD\Semantic Web\Applications\project\email\code\EmailMessage.java`

The main cache
This class has a method for adding message to the memory model and the java class for this can be found at

`\\0-data-2\shared\Labs\Bristol\PSS\DMSD\Semantic Web\Applications\project\email\code\CacheModel.java`

Caching previous query results class
This cache is maintained by the search engine itself and keeps all the previous results of queries in the cache. The partial implementation for this can be found at

`\\0-data-2\shared\Labs\Bristol\PSS\DMSD\Semantic Web\Applications\project\email\code\Cache1.java`

Caching the message Ids in the database class
The polling program maintains this cache putting the message IDs into a cache that can be used by the search engine. The partial implementation for this can be found at

`\\0-data-2\shared\Labs\Bristol\PSS\DMSD\Semantic Web\Applications\project\email\code\cache2.java`

The javadocs for all the classes can be found at

`\\0-data-2\shared\Labs\Bristol\PSS\DMSD\Semantic Web\Applications\javadoc\index.html`

Program listings
The source code is on the shared directory on the Hplabs-bristol network at

`\\0-data-2\shared\Labs\Bristol\PSS\DMSD\Semantic Web\Applications\project\email\code\`

Other references

1. HP technical report RDF Object by Alex Barnell
2. `http://jetty.mortbay.org/jetty/`
3. `http://www.mysql.com/`
4. `\\0-data-2\shared\Labs\Bristol\PSS\DMSD\Semantic Web\Applications\project\email\archive\emailbot.jar`