The ePerson Snippet Manager: a Semantic Web Application

Dave Banks, Steve Cayzer, Ian Dickinson, Dave Reynolds
Digital Media Systems Laboratory
HP Laboratories Bristol
HPL-2002-328
November 27th , 2002*

semantic web, knowledge management, personal information management

In this report we describe the lessons and experiences from developing a substantial semantic web application in the domain of community knowledge management. This application, the *Snippet Manager,* is built upon our ongoing *ePerson* investigation. An ePerson is a personal representative on the net that is trusted by a user to store personal information, and make it available under appropriate controls. Our prototype Snippet Manager is a tool into which a community of users can deposit small items of information (e.g. notes, bookmarks, news items) and annotate, structure and share them with others in the community. The infrastructure and architecture we developed, and the insights arising from this work, are applicable to many semantic web information management applications.

The ePerson Snippet Manager: a Semantic Web Application

Dave Banks, Steve Cayzer, Ian Dickinson, Dave Reynolds

14/11/02, v.1.0

Keywords: semantic web, knowledge management, personal information management

Abstract

In this report we describe the lessons and experiences from developing a substantial semantic web application in the domain of community knowledge management.

This application, the *Snippet Manager,* is built upon our ongoing *ePerson* investigation. An ePerson is a personal representative on the net that is trusted by a user to store personal information, and make it available under appropriate controls. Our prototype Snippet Manager is a tool into which a community of users can deposit small items of information (e.g. notes, bookmarks, news items) and annotate, structure and share them with others in the community.

The infrastructure and architecture we developed, and the insights arising from this work, are applicable to many semantic web information management applications.

Table of Contents

1 Executive Summary

In this report we describe the lessons and experiences from developing a substantial semantic web application in the domain of community knowledge management.

This application, the *Snippet Manager,* is part of our ongoing *ePerson* investigation. An ePerson is a personal representative on the net that is trusted by a user to store personal information, and to make it available under appropriate controls. Our prototype Snippet Manager is a tool into which a community of users can deposit small items of information (e.g. notes, bookmarks, news items) and annotate, structure and share them with others in the community.

We have built a complete prototype implementation of this application together with a set of supporting semantic web infrastructure components. This exercise has demonstrated feasibility of the approach and given some hints of the utility of the application, even though it is not yet suitable for routine practical use. We have identified many lessons and design issues for each of the system components and for the overall prototype. The information gained and captured here is relevant both to any future development of ePerson tools and to many related semantic web applications.

The key features of the components we developed include:

- transport-independent addressing and message API with support for broadcast/multicast and for message signing;
- distributed RDF query, based on a query-by-example pattern matching style;
- a name resolution and service discovery infrastructure based on broadcast discovery of RDF self-description records;
- a personal knowledge-based hosting infrastructure, supporting role-based access control;
- networked RDF sources providing access to the DMOZ classification hierarchy and the DMOZ page classifications themselves;
- a vocabulary for representing user profile information with interoperability hooks to related profile schemas (which is populated both manually and from automatically inferred interest vectors derived from browsing records);
- a set of data models for representing items of user information ranging from generic snippets through to specific items such as bookmarks in which the users themselves and personal collections of items (workspaces) can also be treated as information items for annotation and indexing;
- a functioning user interface and prototype application allowing creation, organization and navigation of personal information repositories driven by the DAML files that define the data model;
- tools for import and export of bookmarks and drop-able items into the information repositories;
- an extensible tool for exploration of a linked community of information stores.

2 Background and objectives

In this section we present our perspective on *personal knowledge assistants*, and our hypotheses for their key characteristics. Our work in this area is set within the context of the ePerson project [1].

An ePerson is a user's personal representative on the net, which is trusted by the user to store personal information, and to make it available under appropriate controls. Such personal information includes user profiles, user interests and preferences, opinions and ratings, bookmarks and directly shared items such as pictures, music and documents. These ePerson nodes are then linked into a peer-to-peer network to facilitate sharing of opinions and information across ad hoc communities.

While there are many possible applications for such a concept, the focus of the work we report on here is the development of personal knowledge assistants that can help individuals to manage the knowledge and information they need. In particular, to help them:

- *filter* information, to track changes and developments in a given area without excessive information overload
- to *forage* for information, to investigate and research new areas[1]
- to *organise* their information for use by both themselves and their peer community

These goals are not novel. Many tools have been developed over the years to cope with information overload and assist in organisation and sharing of information. Some of the latter have been successful (e.g. Lotus Notes) but largely the problems of information overload have not been solved. So what angle do we have to offer to address this?

We believe that the key is a combination of three principles – *social filtering*, *structured knowledge* and a *person-centric* approach.

Social filtering. Our belief is that it is the people in these connected communities that will do much of the collecting, filtering, qualifying and organising of information. The techniques of pattern recognition and information retrieval have a useful role to play in automated knowledge assistants but don't provide a complete solution. Our tools should primarily support people in performing such filtering, and as far as possible automate the sharing and reuse of this human work. In summary, harness the people in the network to help sort the signal from noise.

Structured knowledge. If the annotations and organisational structures that are built up by these communities are machine accessible we can provide useful automation or automated assistance. If the metadata concerning individual items and the links or relations between items are explicitly captured we can provide more flexible and powerful tools for visualising, exploring and combining such information. If communities agree on particular structured terminologies, or

[1] Simply looking up individual facts does not seem to pose much of a problem for people. It is the more substantial activities of deeper investigations into a given topic area that appear ripe for greater assistance [27].

ontologies, to use in annotating and linking items we have a greater chance of discovering common patterns and automating reuse.

The semantic web [2] provides a set of standards and tools that directly address this issue of accessed to structured, shared metadata. In particular it defines a common data model for metadata RDF [3] that supports free combination of metadata from many terminologies and sources. The next layer up of the semantic web, intended to explicitly model these terminologies, is still under development [4] but is likely to be based upon existing proposals such as DAML+OIL[2] [5]. We chose to use DAML in the work reported here.

Person-centric. This principle has several facets – ownership, vocabularies and scaling.

People should own their own information. This is important both because we believe, a priori, in respect for the individual and their freedom and privacy, and because it is more effective. If I create a knowledge base which I use regularly for my own purposes I am more likely to keep it up-to-date than if I create an index solely for use by others. Similarly, it is one thing to allow my close colleagues to see all of my bookmarks and browsing patterns to assist them in following up my investigations, it is another thing to imagine anyone being able to see them. We have to balance the benefits of connecting people together against the problems of intrusiveness. The right balance point, in our view, is that individuals control the data, annotations and structuring that they put into the information infrastructure and have complete control over the visibility of this information.

Second, consider the notion of knowledge structuring, such as the use of ontologies for metadata tags and relational properties. This is typically applied top down. Some large organisation, e.g. a professional or standards body, spends much time developing standard terminologies that can then appear cumbersome and inflexible when applied to a given specific, personal problem instance. We invert this and allow individuals and small work groups to chose their own terminologies (and how rigidly these should be applied) but then support them in discovering existing ontologies which might be adapted for reuse or in linking their own terms to ontologies developed by other groups. This bottom up approach, sometimes dubbed *emergent ontologies,* allows us to make better trade-offs between flexibility and reuse.

Thirdly, we believe person-centric approaches are inherently more scalable – both in the networking sense of decentralised architectures and in the organisational sense. An information portal that is maintained by a centralised group is limited by the resources of the group in how effectively they can maintain it, change its structure, imagine and develop new tools. An information infrastructure that provides everyone with the building blocks for adapting and extending the tools to their own needs should be able to grow and adapt.

[2] This will henceforth be abbreviated to DAML for narrative convenience; the contribution of the OIL researchers to the DAML+OIL standard is fully acknowledged.

Our hypothesis is that combining these three principles in the design of a personal information infrastructure will indeed lead to substantial improvements in the way people filter, forage for and organise information.

As a first test of this hypothesis we have chosen an example information assistant – the shared Snippet Manager – and developed a working implementation of it, based upon semantic web representations, guided by our three principles. In the rest of this report we describe, in some detail, the design and implementation of this first test application and the lessons we have learnt along the way.

3 Introduction to the Snippet Manager application

The *Snippet Manager* is a tool for organising collections of small arbitrary information items (news articles, papers, pictures, bookmarks, notes) and sharing them across ad hoc peer groups. These items may be self-contained units like pictures or small extracts from larger documents (such as email messages and reports).

Each user has a *knowledge base* to store their user profile, preferences and the information snippets of interest to them.

The user can then view, organise and add to the pool of items in their knowledge base through a collection of user interface tools (implemented in Java™) that comprise the Snippet Manager.

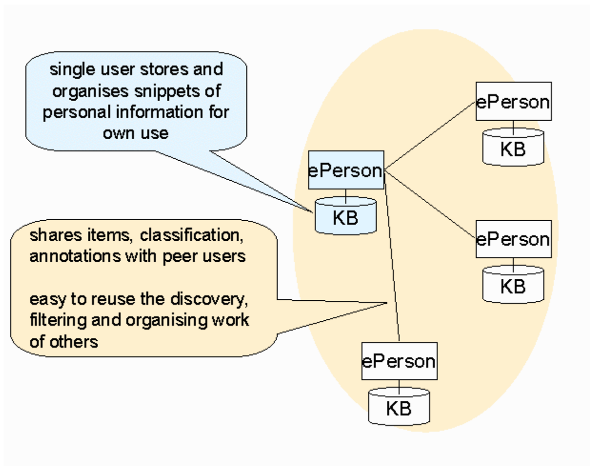

single user stores and organises snippets of personal information for own use
shares items, classification, annotations with peer users
easy to reuse the discovery, filtering and organising work of others
ePerson
KB
ePerson
KB
ePerson
KB
ePerson
KB

Figure 1: Overview of Snippet Manager

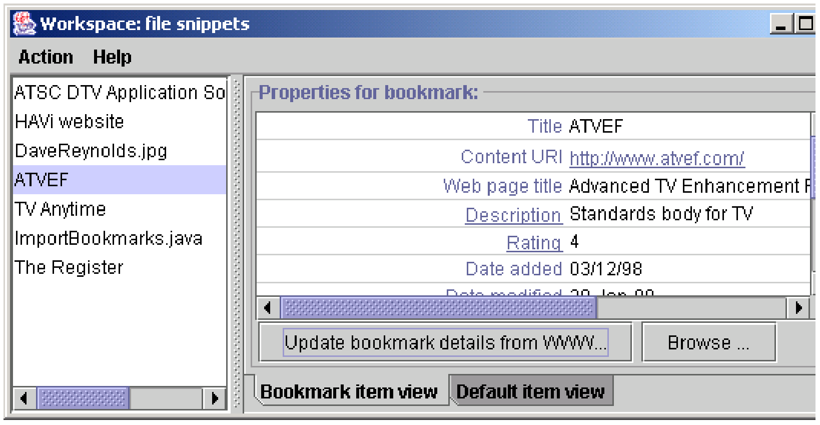
These items are all sharable with other peer Snippet Managers known to the user. Peers can be discovered using local broadcast or by receiving a reference to an ePerson through some other medium like email. A community browser tool within the Snippet Manager allows a user to explore this space of shared information – for example discovering the comments, ratings and classifications that other users have applied to a given item.

Since each ePerson node has knowledge of peer ePersons, they could be linked more widely into a decentralized network where the links follow the social links of the users rather that some notion of network locality. This provides a platform for future extensions to community browsing, exploiting topic-based query routing.

When a new user wishes to start using the Snippet Manager they first need to create a personal knowledge base on some knowledge base host. This *kbhost* may be run locally on a personal computer or run centrally on a managed server. As part of setting up this initial knowledge base a set of cryptographic credentials[3] are created to identify the user to the kbhost. These credentials enable the user to access their knowledge base securely from multiple locations.

When the Snippet Manager is started (supplying the appropriate credentials file and password) the user can then create visual collections of information *items* which we call *workspaces*. Each *item* represents information on some snippet and consists of an internal id, a URI reference to the item itself and an arbitrary collection of metadata. Some metadata is appropriate to particular types of items (for example web bookmarks will often have a *title*), some metadata is generic (e.g. *comments*, *ratings*).

[3] A public/private key pair and an eperson-identity based on a hash of the public key for the person and a similar key pair and hash-based id for the knowledge base.


Workspace: file snippets

Action Help

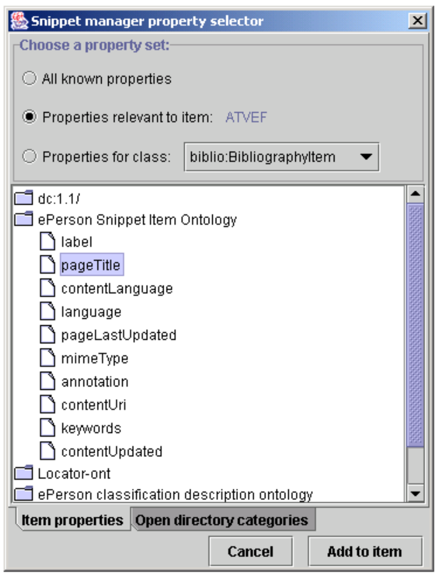
ATSC DTV Application So
HAVi website
DaveReynolds.jpg
ATVEF
TV Anytime
ImportBookmarks.java
The Register

Properties for bookmark:

Title	ATVEF
Content URI	http://www.atvef.com/
Web page title	Advanced TV Enhancement F
Description	Standards body for TV
Rating	4
Date added	03/12/98

Update bookmark details from WWW... | Browse ...

Bookmark item view | Default item view


Snippet manager property selector

Choose a property set:

All known properties

[✓ Properties relevant to item: ATVEF]

Properties for class: biblio:BibliographyItem

dc:1.1/
ePerson Snippet Item Ontology
label
pageTitle
contentLanguage
language
pageLastUpdated
mimeType
annotation
contentUri
keywords
contentUpdated
Locator-ont
ePerson classification description ontology

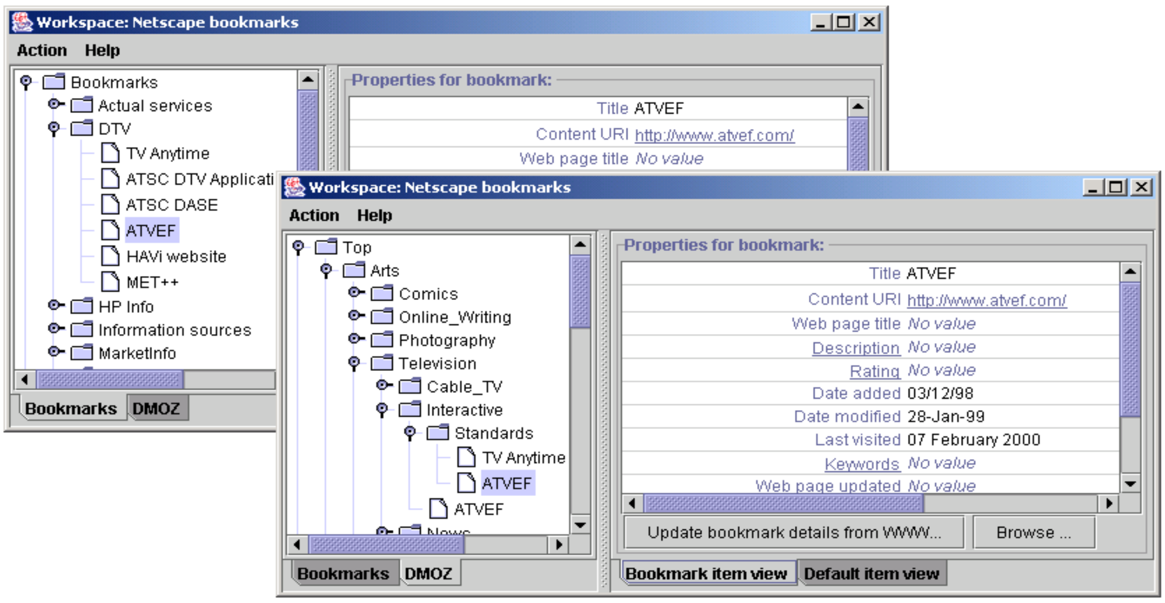
Item properties | Open directory categories

Cancel | Add to item

Figure 2: Simple workspace and vocabulary-driven property editor

Through the UI, all of the current metadata values can be viewed and edited and new annotations can be added. The set of properties available for attaching such annotations is drawn from a set of vocabularies that are loaded into the application at start up.

A particularly common and important form of metadata is the classification of an item into one or more categories in one or more classification schemes. Classification schemes may be shared between users – a user workspace can export a classification scheme as a `classificationService` that can then be discovered and reused by other users in the network. Amongst the standard classification schemes provided are the DMOZ open directory scheme [6] and the scheme generated from the bookmark/favourites folder structure when the user imports their web bookmarks. The same workspace can be viewed organized according to any of the classification schemes currently available.


Workspace: Netscape bookmarks

Action Help

Bookmarks
Actual services
DTV
TV Anytime
ATSC DTV Applicati
ATSC DASE
ATVEF
HAVi website
MET++
HP Info
Information sources
MarketInfo

Bookmarks | DMOZ

Properties for bookmark:

Title	ATVEF
Content URI	http://www.atvef.com/
Web page title	No value

Workspace: Netscape bookmarks

Action Help

Top
Arts
Comics
Online_Writing
Photography
Television
Cable_TV
Interactive
Standards
TV Anytime
ATVEF
ATVEF
News

Bookmarks | DMOZ

Properties for bookmark:

Title	ATVEF
Content URI	http://www.atvef.com/
Web page title	No value
Description	No value
Rating	No value
Date added	03/12/98
Date modified	28-Jan-99
Last visited	07 February 2000
Keywords	No value
Web page updated	No value

Update bookmark details from WWW... | Browse ...

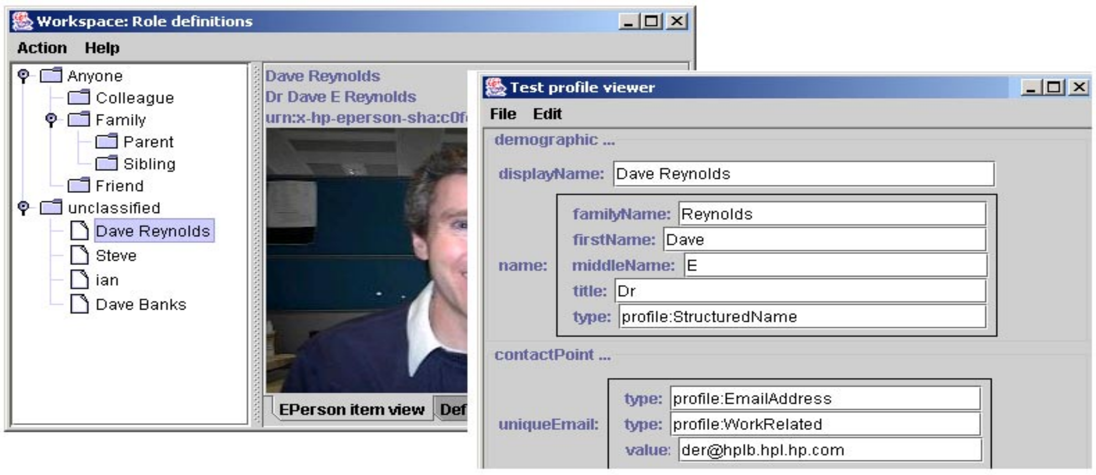
Bookmark item view | Default item view

Figure 3: Switch between bookmark and dmoz view of item

New items can be added to a workspace by importing them from some external source (such as the bookmarks or favourites file from a web browser) or by dragging them from the desktop. It is easy to drop such items into a workspace and later on go back and add annotations and classifications – *capture now, organize later.*

Items that are already on the web are referred to by URI and only their metadata is stored in the ePerson network. Items such as local text snippets are copied to a web server integrated with the knowledge-base host and assigned a URI so that the content data is also accessible to other users.

In addition to the *items* and their organizing *workspaces* the ePerson knowledge base stores a user profile. Part of this profile is manually entered using a structured editor.



Dave Reynolds
Dr Dave E Reynolds
urn:x-hp-eperson-sha:c0f

demographic ...

displayName: Dave Reynolds

name:
familyName: Reynolds
firstName: Dave
middleName: E
title: Dr
type: profile:StructuredName

contactPoint ...

uniqueEmail:
type: profile:EmailAddress
type: profile:WorkRelated
value: der@hplb.hpl.hp.com

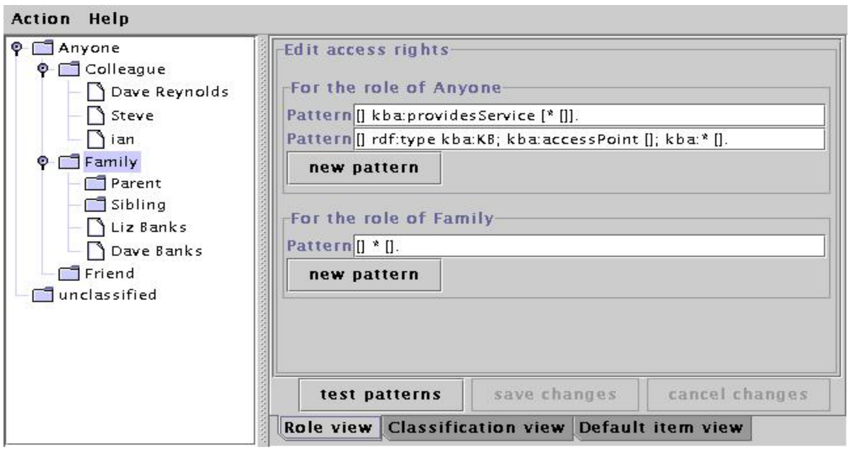
Figure 4 User profile – embedded view and manual editor

This part of the profile includes name and contact information, organizational membership and contact information for other colleagues.

Another part of the profile is a weighted-vector of user interests organized according to the DMOZ topic hierarchy. This is automatically derived from the logs of a proxy server that a user can choose to use when browsing. This *history server* offers users the ability to search over previously visited web sites[4] as an incentive to using the service.

The set of other ePersons visible to the user can be derived either from the colleague information explicitly entered into the user profile or by automatic discovery of ePersons within the local network region. The user is able to classify these other ePerson identifiers to according to the roles they can play.

[4] "I know I saw something about *x* just recently …"



```
Anyone
  Colleague
    Dave Reynolds
    Steve
    ian
  Family
    Parent
    Sibling
    Liz Banks
    Dave Banks
  Friend
unclassified
```

Edit access rights

For the role of Anyone

Pattern [] kba:providesService [* []].

Pattern [] rdf:type kba:KB; kba:accessPoint []; kba:* [].

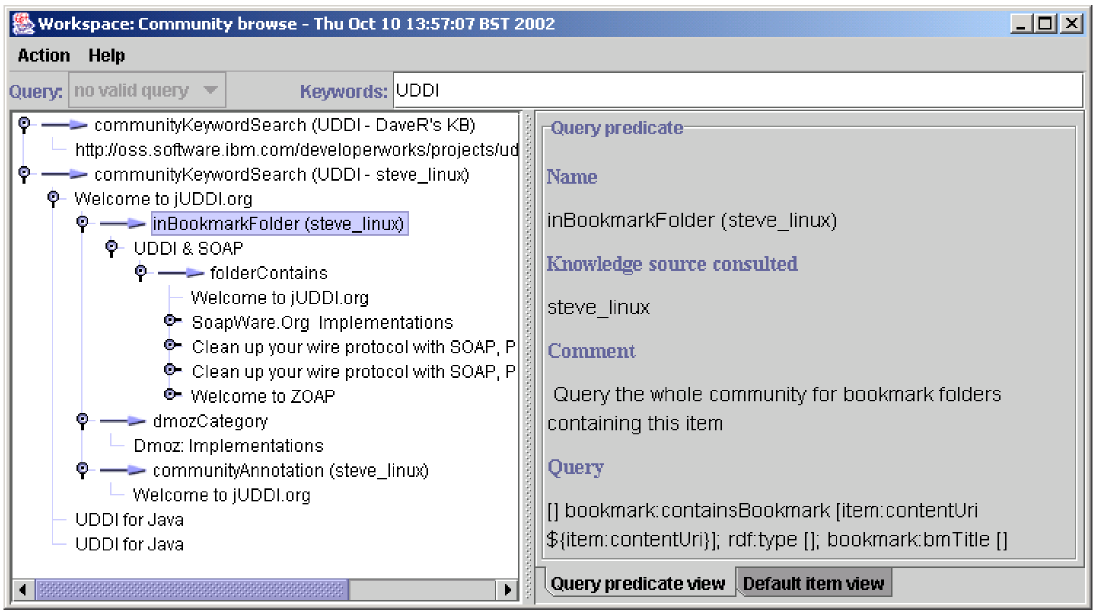
For the role of Family

Pattern [] * [].

Figure 5: Role classification and access control patterns

The user can then base access control on this role classification. They can choose what patterns of profile and stored data, if any, are visible to people in each role.

Finally, the community formed from the collection of these known ePersons can be interactively explored using the *community browser* tool. This can be seeded with information items using drag-and-drop or by keyword search. The community can then be explored for further information on the seed items. For example, it is easy to take a web item you are interested in, from there find other users in the community who have bookmarked the same item and view their annotations and ratings of it. From the classifications they have assigned the web item you can then view other items they have classified similarly and thus discover related items that may be of interest to you.


Workspace: Community browse - Thu Oct 10 13:57:07 BST 2002

Query: no valid query Keywords: UDDI

```
communityKeywordSearch (UDDI - DaveR's KB)
  http://oss.software.ibm.com/developerworks/projects/ud
communityKeywordSearch (UDDI - steve_linux)
  Welcome to jUDDI.org
    inBookmarkFolder (steve_linux)
      UDDI & SOAP
        folderContains
          Welcome to jUDDI.org
          SoapWare.Org Implementations
          Clean up your wire protocol with SOAP, P
          Clean up your wire protocol with SOAP, P
          Welcome to ZOAP
    dmozCategory
      Dmoz: Implementations
    communityAnnotation (steve_linux)
      Welcome to jUDDI.org
  UDDI for Java
  UDDI for Java
```

Query predicate

Name

inBookmarkFolder (steve_linux)

Knowledge source consulted

steve_linux

Comment

Query the whole community for bookmark folders containing this item

Query

[] bookmark:containsBookmark [item:contentUri ${item:contentUri}]; rdf:type []; bookmark:bmTitle []

Figure 6: Community browser

4 Overview of architecture

In this section we give an overview of the architecture of the ePerson infrastructure and the Snippet Manager application. In the next section we will then explore each component in the architecture in detail looking at design and API issues.

4.1 Design principles

A basic principle of the ePerson infrastructure design is to use the RDF language to represent all stored information. The *item* metadata itself, the user's profile, the addressing and capability descriptions of services, the user's preferences and configurations and the current state of the application UI are all represented in RDF and stored in the knowledge base. In this way the same set of data manipulation and query tools can be reused at many parts of the architecture. The openly extensible nature of RDF makes it easy to add new properties to objects within the system as new requirements or opportunities are discovered.

A second key principle in the architecture is to clearly separate the details of the messaging layer and physical hosting of services from the application code. Thus all services (knowledge base hosts, ePersons, supporting services) are identified by location independent names – URNs. In this way services can be freely relocated – not only to different network hosts but also to different connection types (for example Jabber or TCP/IP).

4.2 Deployment architecture

Conceptually each ePerson node comprises a knowledge base together with an application UI. In practice it is more flexible to separate the knowledge bases from the application UI, so that the users can access their knowledge base from different locations and so that the knowledge base remains available online even if the user's access machine is out of contact.

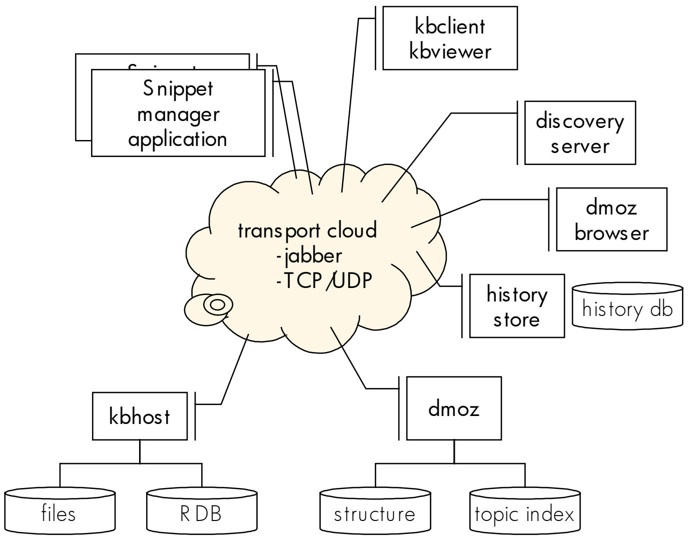


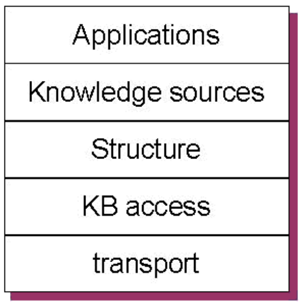
Figure 7: Deployment architecture

In addition to the application UI and the knowledge base host the ePerson infrastructure includes:

- a gateway onto the history logs of a proxy server – used for inference of user interests;
- a copy of the DMOZ open directory database [6] which can map URIs onto DMOZ topics;
- a set of supporting graphical and console based tools to allow developers to access and test the network, and

- a discovery server that can probe the services available on the local ePerson network.

4.3 Layered architecture



The ePerson infrastructure is designed as a series of layers, each of which provides an external API which abstracts the implementation details away from the client application.

The lowest layer, the *transport layer*, hides the details of the message transport machinery. Currently, we support raw TCP/UDP and Jabber [7] transports. This layer supports broadcast/multicast as well as unicast messaging to enable discovery of local services. As mentioned above, all end-points accessible over the transport layer are identified by location-independent URNs. The transport layer itself does not resolve these address names into locations but relies on the distributed query machinery in the next layer up to discover these mappings and inform the transport layer accordingly.

The *knowledge base access* layer provides facilities for remote access to RDF stores and services. It includes support for distributed query, aggregation of results from multiple sources, provenance tracking and fine-grained access control.

The *structure layer* provides facilities for modelling the RDF vocabularies in the system (both internal vocabularies and application specific ones) using the DAML language. Both this and the previous layer were built using HP's *Jena* semantic web toolkit [8].

The *knowledge source* layer provides specific knowledge services such as the DMOZ classification server, importing profiles from the history server and a discovery server that maps the local network space on behalf of the applications.

Finally the *application layer* includes reusable UI components for viewing and manipulating RDF data, viewing tools for knowledge base access during development and the *Snippet Manager* application itself.

5 Details of the architecture layers

In this section we walk through each of the layers of the implementation (as introduced above) and describe in more detail the precise structure and design. We will not give complete API specifications (the Javadoc for the complete software suite is well commented and available online at [9]) but we will give enough details of the structure and API to illustrate the key features of our design approach. In each subsection we will summarize the lessons learnt from this exercise and highlight those features of the design that worked well and those that did not.

5.1 Transport layer

5.1.1 Overview of transport layer

The ePerson transport layer has the following goals:

- Provide a uniform XML oriented Java messaging API.
- Support the following messaging semantics:
  - Request-response (i.e. conventional RPC)
  - Request-multiple response
  - Asynchronous messaging (i.e. no response expected)
- Provide pluggable connectors to support a variety of underlying transport technologies, such as direct TCP/UDP connections and Jabber (an XML based Instant Messaging system [7]).
- Provide persistent names (URNs) for endpoints that are independent of the underlying connector. It is expected that these names are stable over time.
- Support for flexible mappings between a name and one or more connector specific addresses. It is expected that these mappings will change over time.
- Support a limited scope broadcast capability for the discovery of local peers.
- Support the signing and verification of messages without using a full *public key infrastructure* (PKI).

5.1.2 Transport-independent addresses

Transport-independent addresses[5] are used to represent the logical transport endpoints between which messages are sent. The idea is to have a stable identifier that represents the endpoint, regardless of how the endpoint is connected to the network.

In particular, consider the case where the endpoint is an ePerson Knowledge Base (KB) installed on someone's personal laptop. At times, that laptop will be in work behind a corporate firewall. At other times, it will be at home, possibly behind a home gateway that implements NAT. At yet other times, it may be directly connected to the public Internet. The preferred means of supporting

[5] It's slightly confusing that we use the term address here, since what we are really doing is providing a scheme for persistently naming endpoints. With hindsight, we should have called this a transport name. Generally, if we talk about the name of an endpoint, then we are actually referring to its transport-independent address.

inbound connections (requests to the KB) may well be different in each case. When behind a firewall, the KB may expose itself to the world through a public Jabber server. When not behind a firewall, it may allow direct connections. Over time, the set of connector specific addresses that are used by the KB will change. However, the transport-independent address for the KB should remain stable.

In general, the transport-independent address of an endpoint is treated as an opaque label. It must minimally conform to the URI syntax, so that it can be the subject or object of RDF statements. In practice, all of the transport-independent addresses we have used within ePerson application are also URNs.

There is an important subclass of transport-independent addresses of the form

```
urn:x-hp-eperson sha:10f93f6e431ff650b5b2b1b35ca79b2be9b0912e
```

or

```
urn:x-hp-kb-sha:739f6a3c541a9d18a588f48e9c403447a144c625
```

that support authenticated messaging without need for a PKI. §5.1.5 contains more details on this.

5.1.3 Connector specific addresses

One or more connector specific addresses can be bound to the transport-independent address of a transport endpoint. Each connector specific address represents a way to actually connect to the endpoint. In the transport API, a connector specific address is simply represented by a Java properties object. The only mandatory property is the *connector* property, which identifies the type (Jabber, direct, etc) of connector to used. All other properties are connector specific.

An address cache is used in each endpoint to hold the mappings from transport-independent addresses to connector specific addresses. The transport API provides a means to populate the cache directly, and a callback can be registered to handle the case were no mapping can be found. Other than this the transport architecture contains no machinery to discover and propagate these address mappings. This is the responsibility of the KB access layer, where several techniques are used, including hardwiring them into personal KBs, querying the discovery service, and using broadcast queries.

Multiple address mappings may be defined for a transport endpoint, as long as each of these uses a different connector type. When sending a message, the transport implementation will try each mapping in turn, in no particular order[6]. Mappings may exist in the address cache for connectors not available to the sender; these are simply skipped. The result of this behaviour is that multiple copies of the message may be delivered to the recipient, hence a means of detecting and discarding duplicate messages is necessary. To achieve this, the sender labels each message with a unique `messageID` and the recipient maintains a cache of recently seen `messageID`s. A message is discarded by the recipient if the `messageID` is already in the cache.

[6] Strictly, a correct algorithm should verify successful receipt before terminating a scan of mapping options. The current implementation only approximates this behaviour.

5.1.4 Authority

In some cases, the endpoint sending a message may be doing so on behalf of another entity. In ePerson this is the case with Snippet Manager, since multiple copies of Snippet Manager may exist on different machines all owned by the same ePerson. So, in a transport message we make a distinction between the *authority* on whose behalf the message is sent, and the endpoint actually doing the sending. In general, this is would usually be combined with the message actually being signed using the private key of the authority. This implies that the user would need to distribute their private key to any endpoints acting on their authority. The means for this distribution is outside the scope of the transport architecture. In ePerson, we achieve this using credentials files.

5.1.5 Authentication and end-to-end security

Message signing is linked to the notion of the message *Authority* (see earlier in this section). The transport can sign a message on behalf of an *Authority* using the private key of that *Authority*.

The Authority element in the message includes the transport-independent address of the Authority (i.e. the name!) and their public key.

For signing to work (without need for a PKI) the transport-independent address must take one of the following forms:

```
urn:x-hp-eperson-sha:10f93f6e431ff650b5b2b1b35ca79b2be9b0912e
urn:x-hp-kb-sha:739f6a3c541a9d18a588f48e9c403447a144c625
```

In the ePerson system, these are used to represent individual ePersons and their personal KBs.

The hexadecimal part is a 160-bit SHA-1 hash of a full 1024-bit DSA public key. By deriving the transport-independent address from the public key, we avoid the need for any PKI like infrastructure with certificates to map between names and public keys. In effect, the public key (or a hash of it) is the identity of the endpoint. This is very much along the lines of the SDSI/SPKI model [11].

The scheme we use for actually calculating the message signature is currently non-standard because of performance problems with the Apache XML signature implementation [10].

Our algorithm is:

- Insert an empty signature element (i.e. `<eps:signature/>`) into the message to be signed. This should be placed at the end of the header section.
- Perform a rudimentary canonicalization by removing all white space, and merging any adjacent CDATA sections. The standard `eps:` element prefix must be used.
- Serialise the XML to a byte-stream using the UTF-8 character encoding.
- Compute a 160-bit SHA message digest over the serialized XML byte stream.
- Calculate a DSA signature over the digest.
- Insert it as content into the signature element using Base64 encoding.

This works and is approximately ten times faster than the Apache XML signature implementation (30ms per signature rather than 300ms on a PIII 700).

The disadvantages are it is non-standard and the canonicalization is less robust to minor changes in the message format (e.g. white space).

5.1.6 Transport layer – assessment and lessons

The use of URNs to name endpoints in the system seems on balance to have been a good choice. The original goal was to enable portability of knowledge bases without ending up with stale links. This was achieved, however the infrastructure for binding names to dynamic addressing information is not really in place. Worse, the bindings are not secure, and this represents a weak point in the security of the whole system. A good solution to these issues is the CNRI handle system [12]. This provides a general-purpose global name service enabling secure name resolution over the Internet, and supports custom data records. The performance of the handle system could be improved with client-side caching, which is not currently implemented.

For more details on the transport layer implementation, see Appendix 6.

5.2 Knowledge base access layer

5.2.1 Overview of the KB access layer

The job of the *knowledge base (KB) access layer* is to provide remote access to collections of RDF. As well as the ePerson knowledge bases themselves, which *are* collections of RDF statements, we also found it useful to expose other services as though they were RDF and make them accessible via the KB access machinery.

The design approach is to provide a coarse grain remote access API and let application code perform fine grain extraction and modification of the resulting RDF data using native Jena API calls. Thus the KB access layer is designed to retrieve subsets of RDF that match some form of pattern, the application might then use fine-grain RDF API calls or query languages such as RDQL [36] locally to extract the salient values from this retrieved RDF subset.

For the client side of the KB access layer we provide an abstraction called a *knowledge source* that can be implemented in many ways. In the simple case, one knowledge source is a remote interface onto one RDF store. We also support the notion of *composite* knowledge sources. These act as if they were a single RDF source that contains the sum of all the statements available from a set of RDF sources. In the implementation, we use the transport layer multicast mechanisms to distribute the query and results are delivered incrementally.

One special case composite knowledge source is the *cloud*. This is the aggregate of all RDF sources accessible by local broadcast, again with results returned incrementally. One use of the cloud is to support the discovery of local (peer) services. Each service is self-describing: it exports a set of RDF statements that describes its identity, supported connection types, location and capabilities. An RDF query pattern that matches the description of the required service can be issued to the cloud. If only one instance of the service is required, the discovery can complete as soon as the first response is received. We discuss this in more detail in section 5.2.3.3.

The server side of the KB access layer provides the dual abstraction to knowledge source, which we called *knowledge provider*. Several implementations of knowledge providers were developed, which interfaced onto file-based and database-based RDF stores. In addition, we built gateways onto existing database formats by building custom implementations of the knowledge provider interface. See section 5.4 for an example of this.

5.2.2 QBE – query by example

A critical design decision in developing the KB access layer was how to specify the patterns used to select subsets of RDF from knowledge sources.

Two existing solutions were already available to us: triple patterns and RDQL, which solve part of this requirement. We developed a third general pattern expression approach, which we term *Query by Example (QBE)*. At the API and network API levels, we support a general and extensible notion of a *Pattern* that encapsulates all of these approaches. In practice, we found that QBE offered the most useful trade-off between complexity and power and used it almost exclusively.

Triple patterns are simply RDF subject/predicate/object triples where any of the three arguments may be replaced by a wildcard. These are simple to encode in a Network API and natively implemented in most RDF toolkits including Jena.[7] Unfortunately they are not very selective and for the queries we needed to express we would either be faced with retrieving unmanageable fractions of the RDF source or making many smaller queries each costing a network round trip.

RDQL is a query language built into Jena, which supports conjunctions of triple patterns with variables allowed at any place in the triple pattern. The set of all matching variable bindings is returned, from which the matching subgraph can be reconstructed. This is more than powerful enough to express all the queries we expected. However, the approach of returning variable bindings and the complexity of supporting arbitrary variable patterns does lead to some implementation overhead.[8]

The QBE approach is a trade-off between these two alternatives, which offers pattern selectivity close to RDQL but with enough restrictions to allow very simple and high performance implementations.

The core idea is to specify the pattern itself as another RDF graph in which bNodes are treated as unnamed variables. We restrict the pattern to those graphs that are expressible using the current RDF/XML or N3 syntax, i.e. where the bNodes (variables) can only form trees[9]. This restriction simplifies the matcher from a subgraph isomorphism problem (NP) into a tree match problem (polynomial) with a simple recursive implementation.

In our data stores themselves we also use bNodes, typically as a means of storing structured data. This raises the question of how we should return bNodes in the query responses since they have no directly addressable identity. In our current design any result subgraph that has a leaf-bNode is expanded to include all statements in the source graph that have that bNode as subject, iteratively. This avoids the need to re-query the source for more information on a bNode. An alternative approach would be to exploit the newly proposed ability to label bNodes so that queries can be issued against a particular bNode – though the integration of that world view with the bNode-as-anonymous-existential-variable world view is tricky.

In practice, some embellishments to this basic approach were needed. Firstly, RDF does not have a notion of blank property nodes, so we allowed reserved property names to act as wildcards, or as wildcards restricted to a specific namespace. Secondly, we introduced a restricted literal notation to indicate regular expression matching of literal strings in place of equality matching. More details on the pattern match algorithm are given in Appendix 3.

[7] Using the `listStatements(new SelectorImpl(s,p.o))` idiom.

[8] Without direct support for subgraph extraction the list-all-variable-bindings approach can lead to combinatoric overheads when dealing with multiple-valued properties. Even small, 50 triple, test examples were enough to indicate that in-memory query processing could dominate over network latency. The current Jena RDQL implementation constructs the matching subgraph on the fly to avoid some of this overhead.

[9] The W3C RDF Core working group is now proposing a named-bNode notation that would enable arbitrary bNode graphs to be expressed in future RDF/XML notations.

We can construct the QBE pattern graph using any RDF notation or via Jena API calls. In practice, we found that a simplified N3 [37] subset provided a compact and relatively understandable syntax and most often used that.

Putting this together, an example query which looks for all services which are of type `kba:KB` and returns their access information (address, port number, connection type) would be:

```
[] rdf:type kba:KB; kba:accessPoint [].
```

where we are assuming that namespaces `rdf` and `kba` have been defined appropriately (the latter being the namespace of the ePerson knowledge base access layer properties). The `[]` symbol is the N3 notation for a bNode so that the first clause "`[] rdf:type kba:KB`" will match any resource that has a property `rdf:type` whose value is `kba:KB`. The semi-colon operator indicates a new property/value pair that applies to the same subject as the previous pair. Thus the second clause "`; kba:accessPoint []`" indicates that this `kba:KB` resource should also have a `kba:accessPoint` property with any value. If this query is issued to the cloud Knowledge source, the resulting set of RDF statements would give the identity, type and access information for all KB services accessible in the broadcast region.

5.2.2.1 QBE – assessment and lessons

This QBE approach worked well for us. The simplified N3 syntax was compact enough to be usefully embedded with code and sufficiently developer-friendly to be usable in the UI of various internal developer tools. The high performance of the matching algorithm meant that reasonable structured queries against in-memory stores took only a few milliseconds on modest PCs, so that the network latency and serialize/parse times dominated over query times. This encouraged us to reuse this machinery at several points in the design. For example, QBE patterns in the access control system specify the sets of statements accessible from a given role. In the notification machinery, QBE patterns identify sets of changes to a knowledge base that should trigger a notification event.

The biggest problem with the design was the decision to automatically expand leaf bNodes. We wrote queries that relied on this feature and then at a later time found a need to refer directly to the node and so gave it a label This then broke the earlier queries, which would no longer auto-expand. More explicit expansion operators would be an alternative solution or to simply drop this feature and expect all queries to explicitly match all the levels to be retrieved.

5.2.3 Network API

The communication between the KB clients (as represented by *KnowledgeSource* objects) and the KB servers takes place using an RDF network API which builds on top of the transport layer described in section 5.1.

5.2.3.1 Operations

The key operations in the kb access API were:

- `add(Model)`
  add a set of RDF statements to a KB

- `list(Pattern)`
  return the set of all RDF statements in a KB which match the pattern
- `update(deletePattern, addModel)`
  delete all statements from the KB that match `deletePattern` and then add all statements from `addModel` into the KB
- `replace(Model)`
  for each (*subject*, *property*) pair in `Model` the corresponding subject resource was found in the KB, all its property values were removed and replaced with those from `Model`.
- `setNotifier(uri, Pattern)`
  Arranges for the KB to monitor if any changes take place that match the given Pattern. If this happens an asynchronous message will be sent from the server to the client that requested the notifier. This message will contain both the changed RDF statements and the *uri* which acts a label for the notification request.

We later extended these operations with two more to support the representation of provenance information – these are discussed below in section 5.2.5.

5.2.3.2 Transport syntax

Each message from client to server, and each response back, is packaged as a small XML fragment comprising:

- an outer element (KBAccessMessage) identifying this as a message of relevance to the KB access layer
- an operation name defining the action to be performed
- a list of arguments, each a string or an RDF model as indicated by a type attribute, note that patterns are themselves either models (in the case of QBE) or encodable as models
- for encoding RDF models we currently use a CDATA section containing RDF statements in NTriple [10] syntax we chose this for ease of processing but could use full RDF/XML syntax by extending the range of type tags supported.

For example a request to list all resources of type `presentation:Desktop` present in a knowledge source is packaged as:

```
<kba:KBAccessMessage xmlns:kba="http://w3.hpl.hp.com/eperson/daml/eperson/kbaccess#">
      <kba:operation> list </kba:operation>
      <kba:arg type="rdf-n-triple">
        <![CDATA[
              _:A29f3b5X3aXf00ef5102eX3aXX2dX51e8
              <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
              <http://w3.hpl.hp.com/eperson/daml/eperson/presentation#Desktop>
              .
        ]]>
      </kba:arg>
</kba:KBAccessMessage>
```

5.2.3.3 Network API – assessment and lessons

The general coarse-grain approach for the network API was effective. Performance was adequate and it was generally easy to define a query that would delimit the useful subgraph. However, it sometimes led to a clumsy programming style. If the application is actually trying to extract a few specific embedded items of information then it would have to issue a general query and then perform additional low level RDF navigation operations to extract the relevant information from the retrieved subgraph. An alternative solution that simulated, for the application programmer, a fine-grained access API but implemented it using the coarse-gain approach via caching would have been more comfortable. This raises challenges such as generalising network queries to ensure adequate cache hits, efficiently determining cache hits for rich pattern languages and maintaining cache consistency if multiple updates are allowed.

As far as the detailed API design was concerned the main area that required some iteration was that of updating models. Initially we expected a combination of *add* and *replace* (update specific property values) to be the primary update mechanisms. However, the *update* (delete pattern + add statements) proved very useful and we tended to batch up a set of changes as a single *update* in preference to using *replace*. Using patterns instead of explicit statement sets to define the subgraph to be deleted worked well. It might sound more dangerous (because an error in the pattern could cause a lot of damage) but in practice the same pattern used earlier to retrieve a subgraph was then reused as the delete pattern in a subsequent *update.* In principle the delete-by-pattern approach should be less brittle in the face of multiple updaters but this was not stressed by our application since we typically only had one updating client at any one time.

5.2.4 Discovery server

As well as its primary role of providing remote access to RDF knowledge bases, the KB access layer also provides a service back to the transport layer.

We described earlier how and why we use location-independent names for all services and then need to be able to map these names to specific connector types (e.g. TCP/UDP or Jabber) and to specific connector-specific addresses. The knowledge base access layer is used for this.

All servers describe themselves (supported connectors, connector addresses and capabilities) in RDF using a built-in *kbaccess* ontology. Applications can then perform searches for these service description statements, convert them into name/location maps and register them with the transport layer. Thus for example a query issued to the cloud of the form:

```
[] rdf:type kba:KB; kba:accessPoint []; kba:* [].
```

would pull back location and capability[10] information on all knowledge bases[11] which would then allow the transport layer to send messages to any locally-known knowledge base.

There is, however, a problem with this.

To discover a single specific service, the broadcast query can be aborted as soon as the first response is received. However, to locate *all* services of a given type (as is the case in the sample above), then the query issuer has to wait long enough to be confident that all of the recipients have responded. This waiting for timeouts is fragile and introduces a substantial slowdown when booting up an application like the Snippet Manager.

[10] The *kba* namespace is used as the basis for all capability descriptions hence the "*kba:* [] "* sub pattern.

[11] I.e. items of *rdf:type kba:KB*.

We addressed this by noticing that the RDF statements that describe a service are, like all RDF, standalone and so can be freely cached and replicated. We could simply have arranged for applications to create a local RDF file cache of important name/address mappings. However, in our dynamic situation, where the service environment can change substantially between application runs, that was not appropriate.

Our solution was to create an additional network entity – the *discovery server*. Each instance of a discovery server does a periodic local broadcast query of the form above, using a long timeout, and caches all the results in a local database. A client application can then simply ask the same query of the discovery server, instead of the whole local cloud, and a get a reasonably up-to-date local network map with much higher performance.

The remaining problem is then for the client to discover the discovery service. The discovery server is intended to change only slowly, so we do cache the last known-good address for the discovery service on each client device. Alternatively, if the last known address does not function, then we use a broadcast query to the cloud looking just for a discovery service. This is a query that we can terminate as soon as the first discovery server responds and so we do not need to wait for a timeout. If no discovery server is found the application can continue to function transparently by using the cloud in place of the discovery server.

5.2.4.1 Discovery server - assessment and lessons

The discovery server was, in retrospect, not a good solution.

The primary problem is clearly that it turns a decentralized network into a centralized one. In principle, the discovery server is only mapping its local network region and a full network could have many local discovery servers doing such mapping, which then link together. In practice the notion of network region is not well defined in the system and we did not build in any mechanisms for selective query routing between linked discovery servers. The notion of separate, self-organizing, index nodes is a common design pattern in peer-to-peer networks [14][15] and in retrospect we should have treated this is a substantial design issue and solved it properly rather than patched around it.

Such centralization manifests in fragility. If the discovery server fails completely, the clients can adapt. However, if the discovery server appears to be working, but is not giving complete information, then some services will be undiscoverable. One manifestation of this occurs if a knowledge base sets an access control pattern that prevents the (anonymous) discovery server from even seeing its self-description records. In this case, the discovery server will not index the knowledge base whereas a sufficiently privileged client would be able to.

The second apparent issue with the discovery server approach is that of security. An attacker could attempt to masquerade as someone else’s knowledge base by placing a bogus advert in the discovery server. We do not have a way of signing sets of RDF statements in a knowledge base, so the discovery server entries are unsigned. In principle, we have protected against this attack with our self-certifying names. The client would find there are two addresses for the attacked knowledge base and when it contacts the bogus one the message-level signatures would not be signed by the private key corresponding to the public-key hash embedded in the knowledge base name. In practice not all of the requisite client-side checks are enabled in our running code.

One could imagine having the discovery server itself perform these signature consistency checks and so only hold valid address maps. However, this would only be of use if the application can trust the integrity of the discovery server would require a different form of certification, perhaps closer to a full PKI system. Having the address map entries self-certifying, or separately signed, is preferable if it can be made to scale.

5.2.5 Provenance representation and API

The Snippet Manager application allows users to view comments, ratings and classifications for an item that could have come from many different contributors. We would like, therefore, to track the *provenance* (i.e. the origin) of such annotations. This is an issue that we tried several solutions for, none of which were entirely satisfactory.

Provenance cannot be handled at the message level. For example, in querying the community for some information, we could implicitly note which data sources provided which statements and so track (at least partial) provenance. This has no particular value in our application, because each data source is free to cache information it has found from other data sources (e.g. the discovery server, or a user copying an item from another person's workspace into their own). The provenance we need to track is authorship not "most recent supplier of this copy".

Our first approach to provenance representation was to embed it within the data model itself. We replaced value literals by bNode structures as illustrated in Figure 8:

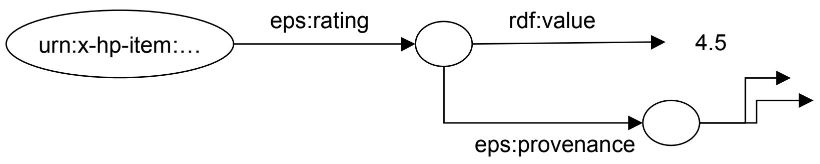


Figure 8: Representing provenance using an intermediate bNode

This is an adequate representation but is difficult to work with at the API level. If a property like a rating could have either a simple literal value or a structured provenanced value then we have to ensure that both query patterns and navigation code deal with both cases. The bNode-closure rule on QBE was, in part, designed to make the query support transparent but the application is still faced with traversing one of two different structures. This could have been simplified by hiding the provenance structure behind an API but that API would have been very specific to this non-standard method of provenance representation.

We revised our approach to use the official RDF technique for this problem: *reification*. In this case, the information in Figure 8 would be represented as shown in Figure 9:

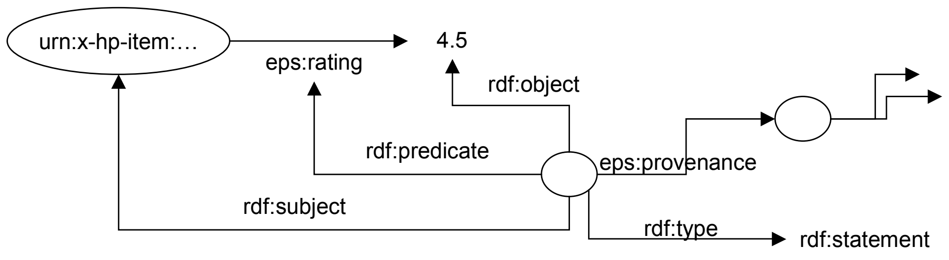


Figure 9: Representing provenance using RDF reification

This removes the problem that code that does not care about provenance information has a more complex structure to navigate. Code that does need to process provenance information is now more complex but we mitigated that by adding two additional Network API operations:

- `addWithProvenance(Model, provenance)`
  This uploads to the remote store the statements in *Model* plus a reification of all the statements in the *Model* along with a link from each reified statement to the *provenance* resource. In the current implementation this transformation is done client side and translated into a normal *add* operation but we could modify the network API to directly support it which would reduce the size of the upload message.
- `listWithProvenance(Pattern)`
  Returns the set of all RDF statements in a KB that match the pattern, along with any provenance information available for any of the statements to be returned.

5.2.5.1 Provenance representation and API – assessment and lessons

This approach of using explicit reification (with some supporting utility functions) to represent provenance information did work, but it is still not an ideal solution.

The primary problem is that of bulk – six statements are needed in this approach for each unadorned statement. In terms of transport syntax the overhead could be reduced by intelligent use of rdf:ID tags to carry implicit reification but our RDF tools did not fully support this mechanism. Similarly the storage overhead could be reduced by custom storage solutions but the current Jena implementation of this is limited.

This sense of high overhead meant that our application code tended to avoid using these *withProvenance* methods, which meant that we captured inadequate provenance in our knowledge bases. This in turn reduced the incentive to start adding better convenience tools for provenance support.

The lesson we take from this is that it is important to determine a provenance mechanism up-front, make sure it works from a practical application developer viewpoint and a theoretical viewpoint, and then use it everywhere.

5.2.6 Access control for RDF data stores

An important aspect of person-centric knowledge management is the notion that individuals own their information and can control who has access to it. This principle drives a need for access control within RDF data stores.

Within the ePerson framework, we implemented a fine grain role-based access control system, for personal knowledge bases. The fine grain nature of our design allows the user to control, down to the level of individual statements, what is visible to others. The complexity of this is balanced by assigning permission to roles, rather than to specific individuals, as there are likely to be fewer distinct role permissions than individuals in a community.

To configure the access control system the user must:

1. Define a set of roles
2. Assign patterns (defining what information is visible) to each role
3. Assign roles to known users

When handling a query for information, the knowledge base must:

1. Authenticate the identity of the user making the request
2. Map the user to a legitimate role (or set of roles)
3. Perform the query on the underlying RDF model
4. Filter the results according to the patterns defined for each role

In the rest of this section we will provide some of the details of the implementation of this scheme, and what we learned.

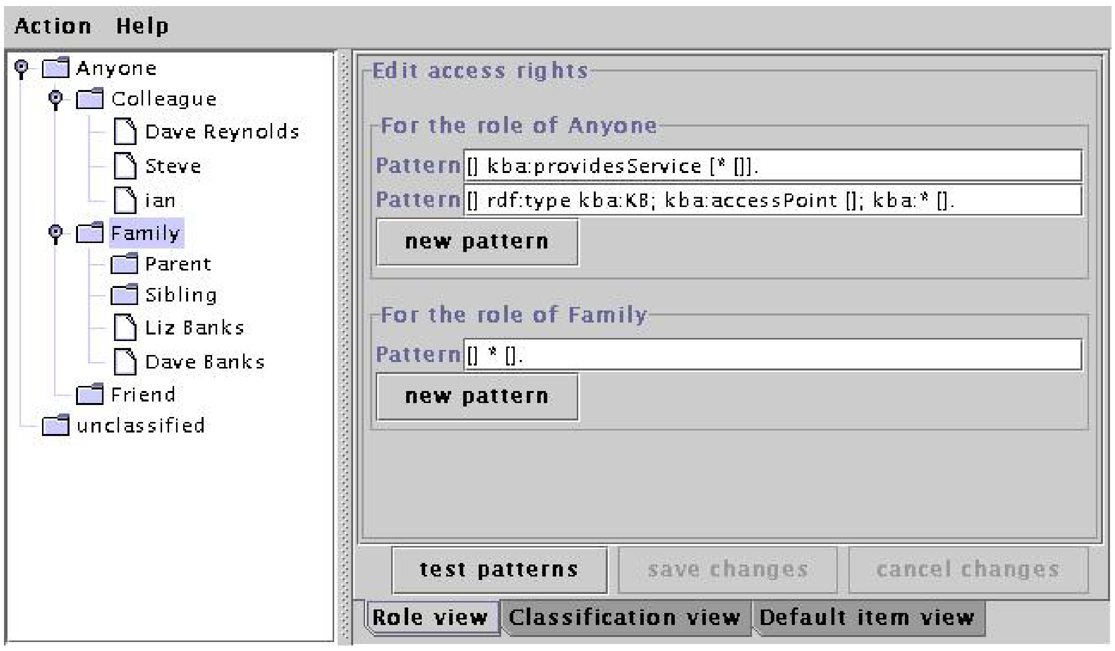
Defining role hierarchies and permissions

A role hierarchy defines a set of roles, and the relationships between them. In our system, we have defined two example hierarchies – *Simple Roles* and *HP Roles* – to experiment with. More can easily be added. The Simple Roles hierarchy is:

```
AnyoneRole
     FamilyRole
            ParentRole
            SiblingRole
     FriendRole
     ColleagueRole
```

One or more patterns can be added to each role to define the set of statements visible to that role. The patterns are actually QBE patterns as described in section 5.2.2. This is a form of positive selection; if a statement is not selected by any of the QBE patterns it will be invisible to the role.

Within Snippet Manager there is a Role Editor, which is used to manage roles and patterns. A screenshot of this is shown in Figure 10:



Pattern: [] kba:providesService [* []].
Pattern: [] rdf:type kba:KB; kba:accessPoint []; kba:* [].
Pattern: [] * [].

Figure 10: The Snippet Manager role editor

A role will inherit the permissions of its parent. Thus, at each level in the hierarchy it is only necessary to specify patterns for the additional information that needs to be made visible.

The Role Editor is also used for assigning users to roles, by simply dragging the user and dropping them on the appropriate role. Users can be assigned to multiple roles; in which case the permissions associated with the different roles are aggregated together.

It is also possible to use several role hierarchies concurrently; the role editor queries the local knowledge base for any resources of type RoleRoot, and builds a role hierarchy view for each. Again, the same user may be assigned to multiple roles across the different hierarchies.

Although provided primarily as a toy example, the SimpleRoles hierarchy includes one special role - the AnyoneRole. This role provides a placeholder for permissions for unknown users. It's also used when the authentication of a user fails; say because the message signature is invalid.

Authenticating requests

The transport layer provides the basic infrastructure for signing and verifying messages. This works without any need for certificates, because we have a convention that the ePerson URN is derived using a hash of their public key.

The Authority structure in the message header includes the claimed identity (ePerson URN), and the public key needed to verify the message. Authentication involves checking the message signature is valid, and then checking the relationship between the claimed identity and the public key.

At the completion of this stage, we have verified that the sender of the message was in possession of the private key belonging to the claimed identity. That's about as good as it gets.

Mapping to a set of roles

The next step is to determine the set of roles to which the requestor is entitled.

The algorithm is as follows:

1. Search the local knowledge base for a list of legitimate roles for this user (as defined by the knowledge base owner).
2. If the requestor has specified a set of desired roles, then form the intersection of the desired roles set and the legitimate roles set. If no roles have been requested, then skip this step. Note that by specifying roles, a requestor can only reduce their access rights.
3. Expand the set of roles by adding all parent roles; this set is termed the set of granted roles.

At the completion of this stage, we have the set of granted roles that will be used to filter the results set.

Filtering results according to role patterns

For each role, we periodically (every 30 seconds) build a query by aggregating the QBE patterns associated with the role, and execute the query against the RDF model for the knowledge base. This query returns the set of statements visible to the role (but not its parent). This set is cached (in a Java hash set) for use as a statement filter. If new data is added to the knowledge base, or the role patterns are updated, there will be a lag of at most 30 seconds before the statement filters are consistent with the changes. This was deemed acceptable.

The process of filtering the results of the user's query involves iterating through the individual statements and testing if they are present in any of the statement caches associated with any of the granted roles. If not, the statement is deleted from the results set.

5.2.6.1 Access control for RDF data stores - assessment and lessons

The access control scheme seemed to work well; by maintaining per-role statement caches (filters) we achieved a good balance between memory usage, query performance, and correct handling of updates. The only apparent performance impact of running with access control enabled was an increase in start up time for Snippet Manager (an extra 5-10 seconds), which is common in Java applications that use Sun's JCE (Java Cryptography Engine). Once running, there seemed to be little degradation in query performance.

This high performance is made possible by our policy of caching statement filters for each role. The current cache implementation might need some refinement if required to scale to a large number of roles.

A more general concern with this type of access control is the way in which users must specify the permissions using QBE patterns. These are not really user friendly. It would be nice to present a higher level user interface, possibly using pre-formatted queries for specific cases, rather like the way our community browser works (§5.5.7).

Another shortcoming of the fine grain scheme presented here is that it only works when reading back information. For write access, we adopted a very coarse grain model; the KB has one owner, and only that owner can make updates. Our intent was to extend this to be a list of owners, but the granularity is likely to remain the same.

We also encountered some system wide issues When access control is enabled on a KB, it ceases to be discoverable by the discover service, since this does not currently sign its requests. Clients rely on the discovery service for access point information, and so were unable to connect to the KB. The fix was to add the following pattern to the AnyoneRole:

```
[] rdf:type kba:KB; kba:accessPoint []; kba:* [].
[] kba:providesService [* []].
```

Finally, we should note that the access control is built upon the message signing provided by the transport layer. Even minor re-formatting of messages (such as Windows style new lines being morphed to Unix style new lines) can break the signatures. We hit several such problems; each of which was hard to diagnose. Whilst some of these would have been prevented if we had used the full cannonicalization that is part of XML signature, many would not. Application writers beware!

5.3 Structure layer

A key user value for the envisaged ePerson applications in general, and the Snippet Manager in particular, is to provide a coherent, integrated view onto the plurality of the user's information sources. Ultimately, a Snippet Manager might be expected to assist in the management, sharing and discovery of the user's email, web bookmarks, citation lists, MP3 playlists and many other resources. To facilitate this integrated view, an essential architectural component is conceptual model of these disparate data forms. This section discusses the data modelling layer of our architecture, and the use of ontologies and other semantic web technologies. Both general principles and the specifics of the prototype application are outlined.

The first component of the conceptual model is a top-level abstraction that encompasses all of the application-layer modelled entities: web addresses, bookmarks, files, lists, etc. The abstraction `Item` is used as the top-level concept. Specific sub-types of Item are then used to model other forms, for example `BookmarkItem` represents web addresses. Items can have structure, so an item can contain other items, and can have a variety of properties asserted about it. It is important to note that an item is the proxy for the original content; rather is the locus of knowledge about that content. So, for example, an item representing a web-address will have as one of its properties the web URL, but the identifying URI of the item is in a completely separate namespace from standard WWW URL's (see below). Among other consequences, this allows us to model a situation in which both Fred and Jane make different, possibly incompatible, assertions about the same web page. Each user would create a separate `Item`, each of which would point via `contentURI` to the web page under discussion, but would provide separate assertions about it (Figure 11).

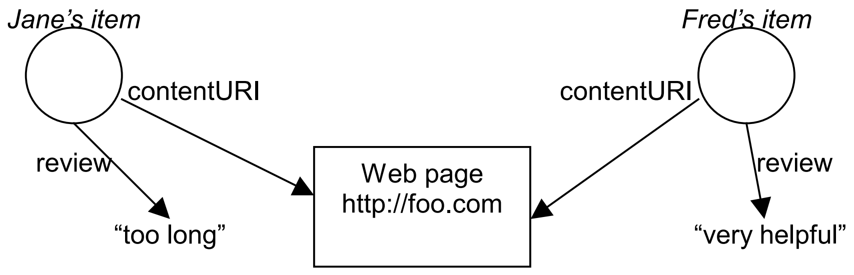


Figure 11: Separate items refer to the same content

Since items are first-class objects in their own right, other users can also refer to them, in the same way as other forms of content. So we can envisage networks of knowledge-sharing building up, as users refer directly to others' assertions (Figure 12):

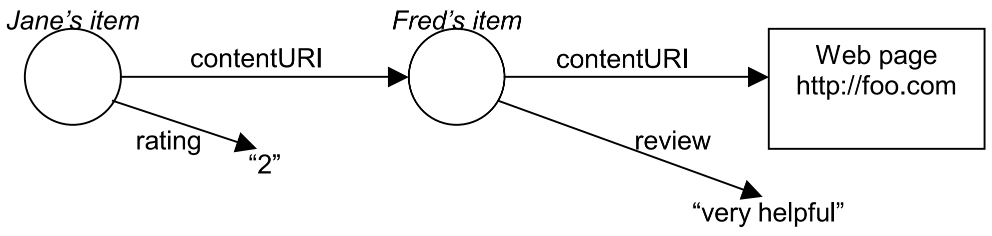


Figure 12: Items can refer to each other

The Snippet Manager conceptual model is defined in terms of DAML ontology classes and properties. Each item type corresponds to a class, and the standard properties that we expect to associate with that item are defined as properties with the class as domain. Extensive use is made of DAML's sub-class and sub-property capabilities for defining the conceptual model. Appendix 1 shows the hierarchy of classes in our current collection of DAML files.

In order to manipulate the elements of the conceptual model in Java, we use the Jena toolkit [8] to expose RDF and DAML values as Java objects. The full contents of the ontology defined by the various DAML files is imported into an ontology model at the start of the Snippet Manager application, and is then available to the rest of the application. In addition, a tool automatically generates Java vocabularies that encode the well-known names from the ontologies as Java constants.

So far, we have discussed the use of the data-modelling layer to store the domain information the ePerson application will process. A key benefit of RDF is that other data formats can be defined and freely mixed in with other data. This is in contrast to, say, a database, in which the relational tables must be carefully designed *a priori*, and cannot be easily extended at run-time. The knowledge base structure described in §5.2 is, therefore, used to store other aspects of the application. In particular, the way that the user has configured their view onto the items they have stored is itself stored as a first-class RDF structure, encoded according to a model defined by a DAML ontology. This view includes both the structuring of items into classifications, and the grouping and presentation of items (for example in workspaces, below). The benefits of this de-segregation between the application data model and the application presentation model include:

- the user's working space is stored on the server, and so is accessible consistently from any instance of the client;
- the user's organisational view of the content (i.e. item) space is itself a shareable commodity, which facilitates a further degree of collaboration between members of a community.

5.3.1 Of names …

Items themselves must be able to record metadata about many different kinds of entity, many of which don't have well-known names. Furthermore, even where well-known names do exist (for example web pages), the item recording assertions and metadata about the named resource has a separate and distinct existence. We must, therefore, name each item using a name from a distinct namespace. The solution we adopted was to create a URI from a GUID: a string that is highly likely never to be generated by another processor, and can therefore be safely regarded as a globally unique identifier. This URI is encoded using an unregistered URN namespace:

```
urn:x-hp-item:7aef341a-272d-51b2-8000-d1afd2ea63a4
```

This scheme works well for generating names that reliably do not accidentally clash with other item names in the set of all workspaces. The drawbacks include the fact that the name is highly non-user-friendly, and so generally needs to be hidden by the user interface, and that there is no obvious way to resolve such names to the resources that they represent. The discovery server provides some assistance for resolution, but has its own design limitations as described in §5.2.4. In general, a scalable decentralised search mechanism would be needed to resolve item URIs to items.

5.3.2 Encoding classifications

A key part of the vision of the ePerson activity is to support the re-use of *organizations of data* by other members of the community. Thus if person A creates a useful set of categories for classifying links about Java programming, there are ways in which this classification can be re-used by other users interested in Java programming. This implies that classifications must be first class objects that can not only be created and refined by one person, but discovered and re-used by others.

To encode a classification, a DAML class `ClassificationService` is defined as a sub-class of `ServiceDescription`. This positions a classification as an active entity – a service in the ePerson network – rather than a purely declarative data structure. Of course, there is a declarative component to the classification, including the categories themselves, as discussed further below. However, this positioning allows us to model dynamic classifications, such as a user's Java taxonomy, in just the same way as more static classifications such as the DMOZ classifier (see 5.4.1).

To model a classification, we must record a number of key properties about it. For example, noting the root node or nodes allows the UI to present the classification in a tree-like or graph-like view. For a given `Item`, a property will relate that item to the class that contains the item in the classification[12]. For different classifications, that property will be different. For example, a classification according to the user's bookmark structure may use a `bookmarkInFolder` property to classify bookmarks, whereas a DMOZ classification may use `dmozTopic`. Thus, to record the classification itself in the KB, we must record the particular property that is used to map items to classes. Specifically, a property `classificationProperty` has a domain of `ClassificationService` and a range of `rdf:Property`. A similar encoding is used to record the property that determines the sub-categories of a given category, which again may be different for each classification. This ability to mix levels of description – `ClassificationService` is from one perspective a first class knowledge item in the KB, while from another it is a meta-description of a

[12] This is a slight simplification, in that a single item may be in multiple categories. This does not change the essence of this description.

characteristic of the user's KB – is a feature of RDF. It is undoubtedly convenient to be able to so freely mix levels of description, but it remains to be seen whether in the longer term this introduces hidden complexities.

5.3.3 Structure layer – assessment and lessons

A design goal at the start of the project had been to provide a highly flexible model for encoding many different types of content items. To this end, we made the representation very sparse: item class and a small number of sub-classes, and RDF properties. This is a very flexible design, but suffers from placing too much burden onto the programmer to manipulate the contents of the items (e.g. getting, setting and processing item properties). On reflection, it may have been that a rather richer representation formalism would have provided more productivity for the application design programmers. For further comments on the interoperation of Java and RDF, see §5.1.2.

Encoding the ontology information in DAML was straightforward. We tried a number of editing tools for DAML files, but did not find any of them to add sufficient value above a capable text editor. Problems included difficulties with namespaces and multiple imports, and a tendency to define range constraints locally to classes using DAML restrictions, rather than global `daml:range` statements. Thus, ontology editing was performed exclusively in standard text editors, with the result that syntax errors were detected fairly late in the process. The DAML validator [16] was helpful in detecting some flaws in the ontology files, but had some notable limitations. For further comments on the use of DAML in the project, see §5.1.3.

We lacked the computational tools to make use of all of the information that was encoded in the ontology. In particular, where the type of a given instance, or the applicability of a property, might have been inferred from the sub-class and sub-property hierarchies, we had no direct access to that information and had to manually code (in the KB or the Java accessors) the closure of the hierarchy.

The flattening of the various layers of the data model into a single RDF KB had positive and negative features. It did provide a very flexible means of storing a large variety of different kinds of information, and this was repeatedly valuable. For example, a new item collection view could be defined that recorded particular parameters that it needed in the KB. No additional code was needed to make these parameters available to the Java code building the view. Among the drawbacks were that resetting the KB during development was hard (without losing valuable information such as a the user's registration details), and that care had to be taken to hide some properties from the user interface (because they were distracting and unhelpful, or dangerous to modify manually). The latter requirement was addressed by adding a class `EPersonInternal` to mark intended-to-be-hidden information, an inelegant solution at best.

5.4 Knowledge source layer

5.4.1 DMOZ Server

5.4.1.1 Introduction to the DMOZ dataset

The DMOZ (Directory Mozilla) Open Directory Project [17]is an open content Internet directory. Although it is similar to Yahoo! in directory structure, DMoz.org offers its directory for free, open use.

The thousands of DMoz.org editors are all volunteers, and DMoz.org has only a handful of official staff members. It is currently owned by Netscape under the Mozilla umbrella.

The RDF version of DMOZ comes as two large RDF files:

- The RDF file `structure.u8` defines the topic hierarchy. There are 428,590 topic categories[13], with an average depth of 6.89 levels below root and a maximum depth of 16 levels below root. The file is 289 Mbytes and contains about 3 million RDF statements.
- The RDF file `content.u8` provides URLs and descriptions of 3,005,746 web sites, including where they fit into the DMOZ hierarchy. The file is 939MB and contains about 16 million RDF statements.

Our interest in DMOZ is as a tool for classifying web pages. This was useful in a couple of contexts in ePerson; for organizing a set of bookmarks into a standard hierarchy and for building a profile of someone's interests from the web pages they view.

Given the size of the DMOZ dataset, it was preferable to make the data available as a shared resource on the network, rather than replicate it at each node. The ePerson knowledge base was the ideal abstraction for this data.

5.4.1.2 Parsing the DMOZ Structure dataset

The structure dataset was parsed directly into a Berkeley DB (BDB) backed Jena model. This took approximately 16 hours, and resulted in a 3.7GB file. This ten-fold expansion in space comes from three things:

1. BDB database files are not compact; typical utilizations are around 60% leaving some room for expansion. (1.7x)
2. Jena stores each RDF statement three times, for fast lookups on subject, predicate or object. (3x)
3. Jena stores each RDF statement as an expanded triple; this less efficient than the same statements serialized to XML/RDF since subjects are replicated and prefixes expanded. We verified this overhead with a fragment of the DMOZ structure dataset. (2.2x)

5.4.1.3 Parsing the DMOZ Content dataset

The content dataset is much (5x) larger than the structure dataset, and needed to be handled differently because of its size. Instead of storing this in a

[13] On the 17th April 2002.

(generic) BDB backed Jena model, we designed a custom BDB database to allow efficient lookups keyed on either URL or DMOZ category. This database was loaded using a custom SAX based parser, which took about 24 hours to run and resulted in a 1.6GB database.[14]

We then wrote an ePerson *KnowledgeProvider* backed by this custom database that answers query-by-example (QBE) queries. We have, in effect, built a gateway making a legacy database appear as RDF.

A gateway based solution for presenting the content dataset has both strong and weak points. It results in a smaller faster database than would be the case if we had used a BDB backed Jena model directly. However, the query handling side was more complicated; we explicitly coded for the types of QBE query we expected, and returned an error in other cases. With hindsight, a better approach would have been to write a wrapper for the custom database that implemented the Jena *Store* interface. This could then have been used transparently by the standard QBE machinery. Future versions of Jena are planned, which will introduce a simpler interface onto storage sub-systems. Such simplifications will make the development of gateway solutions much easier.

5.4.1.4 Classifier for mapping from URLs to DMOZ categories

Our DMOZ classifier works only on the web page URL; all page content is disregarded. The algorithm starts with the URL and looks this up in the DMOZ content dataset. If there isn't an exact match, then the URL is truncated at the last "/" character and the lookup retried. This process is repeated until a match is found, or there are no more "/" characters. At this point, a few variants of the base name are tried before giving up.

Multiple matches are possible; this results from editors categorizing the same site in several places in the DMOZ hierarchy.

5.4.1.5 DMOZ with ePerson

DMOZ category data is used in several places in ePerson:

- The items in a workspace can be classified using the DMOZ service.
- A collection view of items within a workspace can use DMOZ category data to present the items hierarchically.
- When importing bookmarks, each one can be classified using the DMOZ service.
- The Community Browser allows a user to lookup the DMOZ category of an item, and to search for other items in the community in the same category.
- There is a dmozViewer tool allowing the DMOZ structure to be browsed. The viewer connects to the DMOZ service to access the structure.

[14] Doing the same with a Jena model would have taken an estimated 60 hours and resulted in a 12GB database.

- A users interest profile (extracted from the history store) is represented as a weighted vector of DMOZ categories.
- A peer-to-peer routing network could be built using a similarity metric between DMOZ interest profiles.

5.4.1.6 Issues

Although the DMOZ datasets are in an RDF-like format, we were unable to parse them without extensive modifications. We wrote Unix `sed` scripts to correct the many syntactic errors. The most common problem was the presence of control characters in literals (the files were not even legal XML!). Another problem was the use of an earlier version of RDF, resulting in an incorrect namespace for the `rdf:` prefix which confused our RDF parser. The final problem was due to the lack of a schema for the DMOZ dataset, leading to uncertainties about the correct namespaces for the DMOZ properties and topic classes.

Once we were able to cleanly parse the data, we were surprised by the relative ease of with which Jena was able to deal with very large RDF models. The long parse times result from the systems being memory limited (256MB); the BDB databases seemed to exhibit little locality of reference, and so caching was generally ineffective.

In general when dealing with large Jena models, it's quite easy to write "pathological" QBE queries that need to touch every statement. These would take many hours to execute, giving the appearance that the system had hung. Interrupting (with ^C) would often cause the Java VM to crash (if the interrupt hit when executing the native BDB code). For some time we suspected some kind of database deadlock was occurring. Eventually we realised that these hangs were caused by pathological queries. The solution was to pre-parse the query, try to detect the pathological cases, and return a "query too complicated" exception. This problem was made worse by the need to serialize all accesses to the Jena model, since Jena's BDB implementation is not thread safe.

5.4.2 History server

A potentially rich source of user profile data is available from the history of user's web browser interactions. We make this available through a *history store*. This system operates as a web proxy, and maintains a keyword-indexed cache of all of the web pages browsed by each user. It is built on top of the Jakarta Lucene search engine [18], and uses PostgreSQL for storing page-hit logs and BerkeleyDB for storing cached copies of web pages. Members of the project team have been using the history store (somewhat sporadically) for about a year now.

The approach we have used to mine the History Store system results in an interest profile for a user that is a set of weighted DMOZ categories, where the weights across all of the categories sum to unity at the root of the DMOZ hierarchy. An example of the RDF data structure used to represent this is shown in Figure 13.

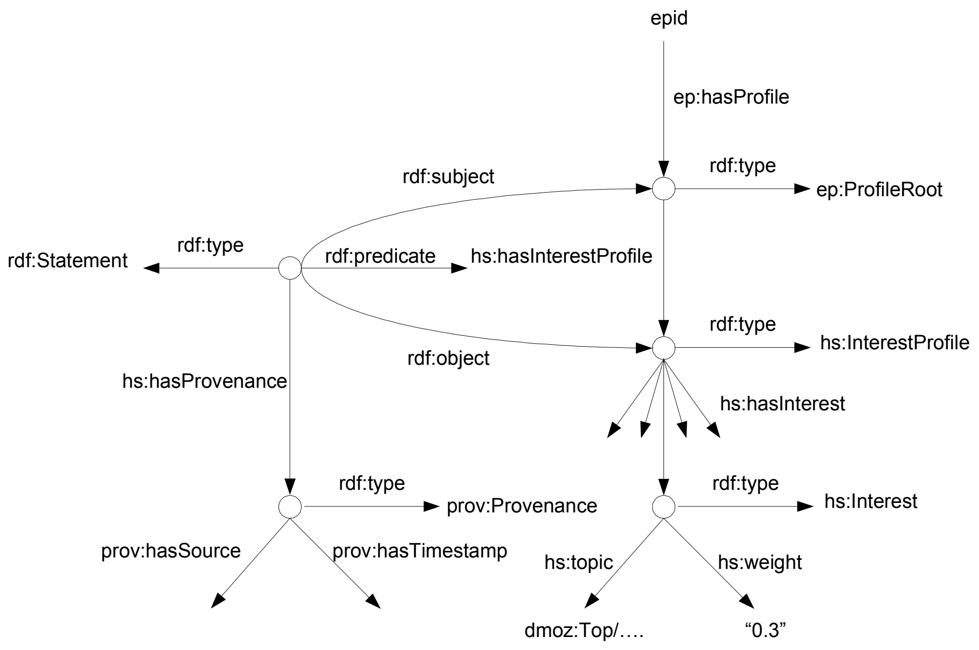


Figure 13: The RDF structure of history store entries

A `hs:hasInterestProfile` property links from the profile root to the interest profile node. This link is reified, allowing statements about the interest profile instance to be made. This is used to record the origin of the profile (i.e. the history store system) and the date the profile was generated.

The set of user interests is represented using `hs:hasInterest` properties to link from the interest profile node to individual interest nodes, which record a DMOZ category and the weight associated with the category. Only categories with non-zero weights are included.

The process for generating an interest profile involves looking at the set of web pages browsed by a user over a rolling time window - typically the last six months. A weighting function can be added to this window, so that pages browsed recently have more significance (linear weighting). Alternatively, all page hits within the window can be treated equally (flat weighting). The result is a score in the range 0.0 to 1.0 for each page hit. The page URL is then mapped to a DMOZ category, using the mechanism outline in §5.4.1.4. This is repeated for all page hits in the time window, and the page scores are accumulated by DMOZ category.

Once all of the page hits have been processed, a two-step normalization transforms the accumulated scores into the final category weights. The first step is to propagate scores from leaf categories back towards the root. The idea here is that the score associated with a category should represent all page hits that mapped onto that category and all of its children. The second stage is to divide all scores by the value on the DMOZ root node. This ensures the final category weights are independent of the number of pages browsed.

Clearly, a significant amount of work is involved in generating an interest profile. Consequently, we have adopted a batch model for processing. The interest profile is re-generated for each user on a weekly basis, and the RDF dataset (typically several hundred statements) is added to the user's personal KB. Since generating an interest profile may involve mapping many thousands of page hits to their DMOZ categories, it is advantageous to execute this on a machine with a local copy of the DMOZ database.

There is much more work that could be done on interest profiles. To date we have not implemented any high level tools for visualizing interest profiles. The data model outlined above allows many interest profiles to exist in a KB, allowing changes in users interests over time to be monitored. Again, we have not exploited this. Finally, there is scope for developing generalized query routing protocols that select the next destination based on a similarity metric applied to interest profiles.

5.5 Application layer

5.5.1 Overview

The Snippet Manager application sits at the top of the ePerson technology stack. It represents the primary means of interaction between the user and the knowledge sources. In a sense, the lower layers provide the key functionality for our ePerson hypotheses, and the application layer allows the user to test these in a simple and intuitive way.

One goal for this first prototype is to produce an application that we could (at least in principle) see ourselves using. For example, a useful outcome would be a tool that enables a user to manage their own data effectively; to merge data from a number of sources and attach useful metadata to it in order to make it meaningful. This general aspiration can be expressed as a couple of specific design objectives:

(i) The application should provide a lightweight affordance for data collection : "*capture now, organize later*".

(ii) The application should enable the user to generate new metadata in a painless and intuitive manner.

The base UI addresses, to a greater or lesser extent, both principles. A set of data import tools is also provided for lightweight capture.

We would also like the prototype to allow use of other people's information – not just the content itself, but also the annotations (metadata) and organisation. We need to do this in a p2p manner, allowing flexible decentralized control while respecting privacy. Application extensions such as a community browser, a classification service and a profile management facility address these needs.

The application thus consists of the base UI itself and a number of tools and services. These will be examined in turn, but first we introduce the key abstractions.

5.5.2 User abstractions

The application is based around some key user abstractions. Firstly, there is the notion of an *Item*. We deliberately avoid the use of 'Snippet' as our base abstraction, for while snippets are certainly items, our intention (as we shall see) is for the 'Item' concept to cover other types of information and content.

A second user abstraction is the *Workspace*, which acts as a container for items. The idea here is that a workspace contains and organises items that are in some sense related. Workspaces can be used in a number of ways. For example, in an information foraging task, they might be convenient places to 'dump' items that can be filtered, annotated and organised later. Alternatively, a workspace can be created specifically to capture the refined output of an investigation, or to accept data from an import tool. Workspaces can be viewed remotely (subject to access permissions), thus providing a means of sharing item information across the peer network. The organisation itself (for example, folder hierarchy) can also be made available for sharing as a *Classification Service* (§5.5.5).

A workspace can be examined using a *Workspace View*, which presents an overall summary of its contained items (an *Item Collection View*) and a context specific view onto a single item (*Item View*).

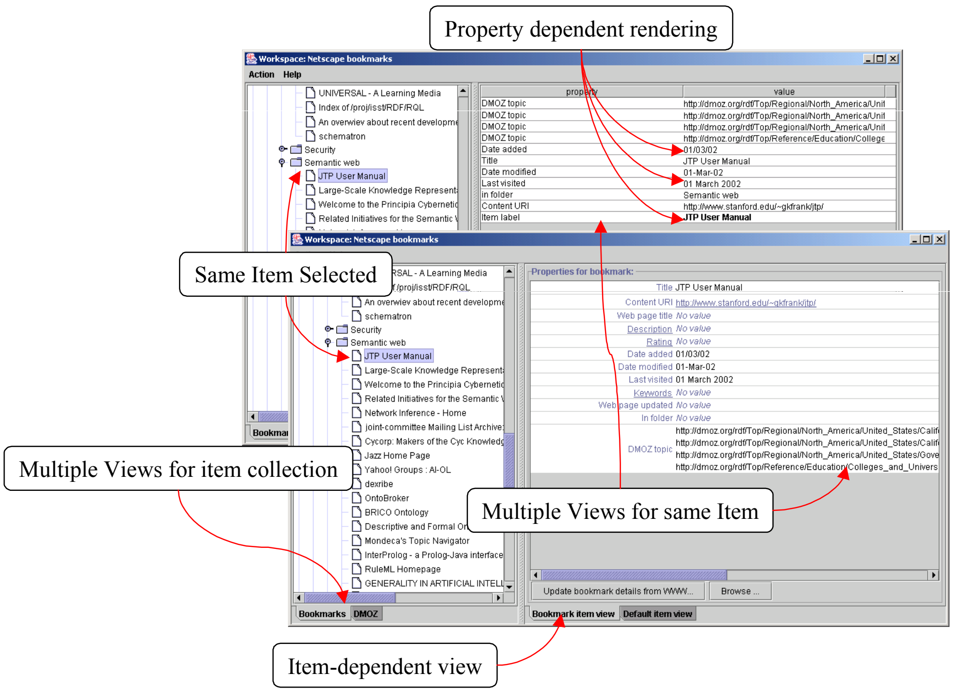
With these abstractions in place, we are able to provide some custom extensions. For example, the profile management tools (§5.5.6) treats users as specialized items and is thus able to present them in the familiar workspace context. The community browser (§5.5.7) presents queries as items, and similarly provides a specialised workspace view for navigation.

5.5.3 Application UI

In this section we describe the base UI itself. Associated application tools and services are described in subsequent sections.

5.5.3.1 Workspace Views

As mentioned above, the central user concept is that workspaces contain items. Thus, the main UI component is a workspace view, which contains item collection views in one (tabbed) pane, and item views in another. Figure 14 shows two such views onto the workspace "Netscape bookmarks". In both workspace views, an item collection view (left pane) presents a tree-like organisation of items[15], in which the item "JTP User Manual" (a bookmark) is selected. The difference between the views is that a different item view (right pane) is selected in each; the lower view shows a bookmark specific item view, whilst the upper view shows a default item view.



Property dependent rendering
Workspace: Netscape bookmarks
Action Help
Same Item Selected
Multiple Views for item collection
Multiple Views for same Item
Item-dependent view
Bookmarks
DMOZ
Bookmark item view
Default item view
Update bookmark details from WWW...
Browse ...

- UNIVERSAL - A Learning Media
- Index of /proj/isst/RDF/RQL
- An overwiev about recent developme
- schematron
- Security
- Semantic web
  - JTP User Manual
  - Large-Scale Knowledge Represent
  - Welcome to the Principia Cybernetic
  - Related Initiatives for the Semantic V
  - Network Inference - Home
  - joint-committee Mailing List Archive
  - Cycorp: Makers of the Cyc Knowledg
  - Jazz Home Page
  - Yahoo! Groups : AI-OL
  - dexribe
  - OntoBroker
  - BRICO Ontology
  - Descriptive and Formal Or
  - Mondeca's Topic Navigator
  - InterProlog - a Prolog-Java interface
  - RuleML Homepage
  - GENERALITY IN ARTIFICIAL INTELL

property	value
DMOZ topic	http://dmoz.org/rdf/Top/Regional/North_America/Unit
DMOZ topic	http://dmoz.org/rdf/Top/Regional/North_America/Unit
DMOZ topic	http://dmoz.org/rdf/Top/Regional/North_America/Unit
DMOZ topic	http://dmoz.org/rdf/Top/Reference/Education/College
Date added	01/03/02
Title	JTP User Manual
Date modified	01-Mar-02
Last visited	01 March 2002
in folder	Semantic web
Content URI	http://www.stanford.edu/~gkfrank/jtp/
Item label	**JTP User Manual**

Properties for bookmark:

Title	JTP User Manual
Content URI	http://www.stanford.edu/~gkfrank/jtp/
Web page title	*No value*
Description	*No value*
Rating	*No value*
Date added	01/03/02
Date modified	01-Mar-02
Last visited	01 March 2002
Keywords	*No value*
Web page updated	*No value*
In folder	*No value*
DMOZ topic	http://dmoz.org/rdf/Top/Regional/North_America/United_States/Calif http://dmoz.org/rdf/Top/Regional/North_America/United_States/Calif http://dmoz.org/rdf/Top/Regional/North_America/United_States/Gove http://dmoz.org/rdf/Top/Reference/Education/Colleges_and_Univers

Figure 14: Two workspace views, demonstrating a variety of key features

There are a number of other things worth noting about this diagram. Firstly, the item collection view (the left hand pane) is split into tabs. This allows the user to

[15] In fact, the organisation is imported from the user's Netscape bookmark folder hierarchy, a process explained in §5.5.4.1

view their items in a number of different organisational formats, depending on the context. In this case, a bookmark folder and DMOZ hierarchy view are offered. These are different organisational structures which both fit a tree like organisation. Other types of organisation (such as simple lists, graphs, or clusters) are also supported by this framework.

Secondly, the item chosen is a bookmark. Thus, whereas the default item view (top) presents all of an item's metadata[16], a bookmark view (bottom) shows the relevant data fields, sorted and organised in a consistent manner. An item dependent view is also a good place to present context specific functionality such as a link out to a browser (the button marked "Browse"). The query predicate (§5.5.7), ePerson (§5.5.6) and role (§5.5.6) views are other examples of item dependent views.

Thirdly, property dependent rendering is implemented. In the default item view, we can see that dates are rendered in a user-defined format (eg "01/03/2002" vs "01 March 2002"), item labels are presented in **bold**, and bookmark folders ("in folder") are rendered by folder label ("Semantic Web"). This last is an example of a general trick where one might want to render a resource using some property of that resource. This allows us to specify some perhaps more meaningful description of that resource (in the bookmark folder case, the folder 'value' itself is essentially a UUID which will carry no useful information for the user). Figure 15 shows how this technique is used to provide an indirect label for a bookmark's folder.

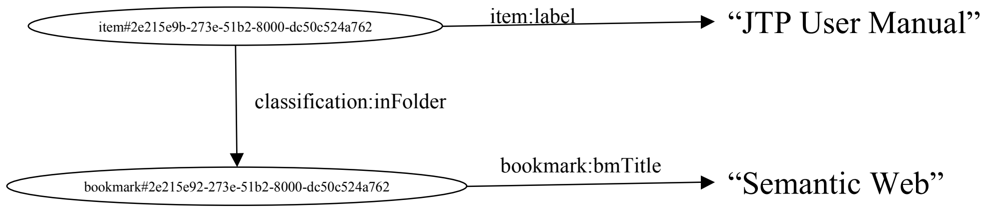

item#2e215e9b-273e-51b2-8000-dc50c524a762
item:label
"JTP User Manual"
classification:inFolder
bookmark#2e215e92-273e-51b2-8000-dc50c524a762
bookmark:bmTitle
"Semantic Web"

Figure 15: Indirect property labelling

5.5.3.2 Metadata Creation

Item annotations can be added manually through the item views, as shown in Figure 2. However, the UI also provides a number of lightweight capture[17] and markup mechanisms. For example, an item (or items) can be dragged from a browser (or from a file store) and dropped into a workspace. On drop completion, some information can be gleaned from its MIME type. Thus, bookmark items and file items are automatically assigned a different `rdf:type` property on import. Metadata can also be altered via drag and drop. When an item is dragged to a new place, the user is making an implicit statement about that item. For example, if a bookmark is dragged to a new folder, it is given a new `bookmark:inFolder` property corresponding to the new folder (and the old `inFolder` property is removed, unless the item was copied). A third mechanism is the bookmark item view, where a user can initiate an HTTP request that will collect any relevant metadata available at the bookmark's URL. Other

[16] Almost – there is a class of 'ePerson internal' metadata, which does not show up even in default item views.
[17] The other main capture mechanism, via import tools, will be described in §5.5.4

mechanisms for assisted markup include the classification service (§5.5.5) and community annotation (§5.5.7).

5.5.4 Import Tools

In this section we discuss the first of the application tools, the import tools. These were conceived and in some cases implemented to accept data from a number of different sources. Examples of such sources include bookmark data (described below), bibliography data (in BibTex or ProCite format), news articles (in RSS format) and user specific documents (in custom format, eg records in an Access database).

In order to provide more context for our discussion, we describe in more detail one of our implemented import tools, the bookmark importer.

5.5.4.1 Bookmark import tool

A key application of Snippet Manager is to enable the discovery, sharing and annotation of bookmarks within a community of users. A good source of such categorized bookmarks is the user's web browser favourites list. Hence, we looked into ways of importing this data into Snippet Manager.

An easy initial target was a batch import function that creates and populates a new "Bookmarks" workspace in the user's personal KB. That workspace is then visible to other users through the Community Browser, just like any other workspace.

The main issue here was for the import tool to be compatible with the many different web browsers in use within the group (IE5, Opera, Netscape 4.7 and Netscape 6.0). To achieve this, we used existing tools to first export the bookmarks to XBEL (XML Bookmark Exchange Language). We then wrote a tool that would read the XBEL file and translate this into an equivalent RDF representation that could be written into the users KB.

It's important to realize that a user’s favourites list really contains two different things: an organizational hierarchy (how the user sees their world) and the individual bookmarks that populate that hierarchy. When importing the data into Snippet Manager both of these concepts are carried across into Snippet Manager. Firstly, we create a custom classification scheme, where each of the bookmarks folders becomes a classification node within the scheme. A classification record (§5.5.5) is created to describe the scheme as a whole. Secondly, we create a new workspace, and add items to this to represent each of the user's bookmarks. An appropriate tree-like item collection view is then added to the workspace to allow the bookmarks to be viewed according to their original hierarchy. We may also classify each bookmark using the DMOZ server, yielding a second useful way to view the bookmarks. A walkthrough of the RDF data generated can be found in Appendix 2.

Note that both these import-time classifications add useful annotations to items and are thus an important mechanism for assisted metadata creation. Other annotations (for example, item type and label) are also added at import time.

One minor complexity is the need to include a RDF Sequence to retain the order of contents of a folder, which has resulted in some duplication of data. This could probably be avoided (and was not actually used by Snippet Manager).

5.5.4.2 Import tools – assessment and lessons

The import tools provide a convenient method for loading knowledge bases with non-trivial amounts of test data. However, they are time-consuming to produce; due to the diverse nature of input data formats, a tailor-made importer is necessary for each data source. An alternative approach would be to pre-process the data and then import using some generic importer. However, the pre-processing is such an overwhelming part of this task that the generic importer would be trivial, and hence the two approaches are in practice almost identical.

Although the bookmark import tool is useful for generating data for demonstrations, it is less than ideal for general use. The problem is that there are two places a user can manage their bookmarks - in their Web Browser and in Snippet Manager. Changes made in either tool really need to be propagated through to the other. This is actually a very hard issue to solve transparently, particularly in an environment with multiple browsers.

A more tractable solution could be to support the manual synchronization of bookmarks between the web browser and Snippet Manager. The synchronization process would involve exporting the current set of bookmarks from the Web Browser and Snippet Manager. These two sets would need to be merged, somehow resolving conflicting changes. Finally, the merged set would need to be re-imported back into the Web Browser and Snippet Manager. Because of the data caching performed by both of these applications, it is likely that a restart would be necessary to see the synchronized bookmarks - hence the need for the process to be instigated manually by the user.[18]

5.5.5 Classification Service

Classification services provide an application extension that enables peers to share organisational metadata. The DMOZ data source is exposed to the ePerson network as one such classification service. That means that, given a set of bookmarks, the service will annotate each with its appropriate DMOZ category (as explained in §5.4.1.4). Hence, bookmarks can be automatically classified according to the DMOZ hierarchy. Certain import operations (§5.5.4) also create a classification service. For example, the bookmark import creates a service that will assign known bookmarks into that user's bookmark folders.

Classification services can be advertised via the Discovery Server (§5.2.4) and thus made available to other peers. Such services are exposed by the "Apply Classification" function on an item collection view. This function allows the user to select from a list of discovered classification services. The chosen service is then invoked to classify (possibly hitherto unorganised) data, and a new item collection view is created for presentation of the new organization. This way of working conforms to our design principle "Capture now, organize

[18] This is current research into synchronization protocols such as [38] that might be applicable to the bookmark synchronization issue.

later", and also provides a mechanism to test our hypotheses about shared organisational structures.

5.5.6 Profile management tools

Another set of application tools provides the ability to manage user profiles, which are based on a rich schema. In the current schema, summarized in Figure 17 (page 62), we concentrate on descriptive information such as demographics, contact points, interests, roles and depictions. The user manually enters most of this information using an editor tool but this volunteered profile is augmented by the inferred interest profile discussed in section 5.4.2 above.

The profile editing tool, shown in Figure 4, is a generic structured value editor driven entirely by the profile schema written in DAML. The only additional information required is a designation of the top level categories and ordering preference hints to ensure that fields were displayed in a sensible order. These hints are provided as additional RDF properties on the classes defined in the profile DAML file.

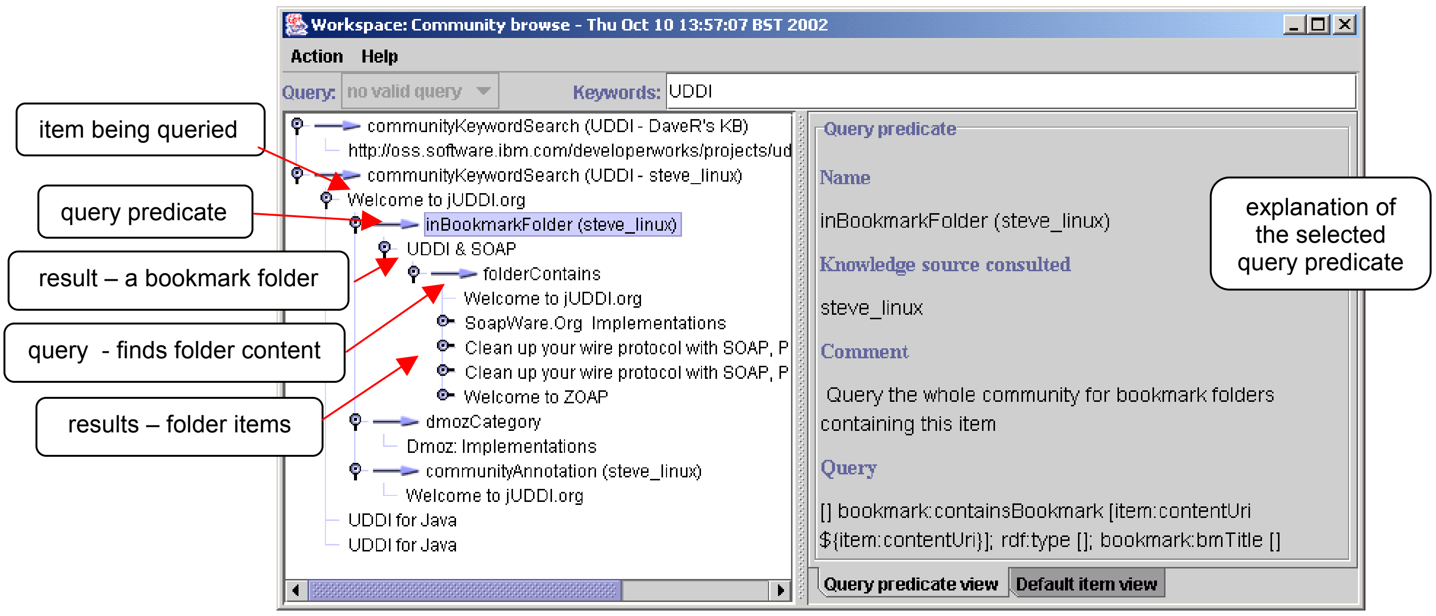
The primary challenge in this area was to develop a profile ontology which could be related to existing schemas (vCard [19], friend-of-a-friend [20] and DRC Orlando profile [21]) to support future interoperability. This proved non-trivial – see Appendix 4 for more details.

5.5.7 Community browser tool

Now that we have a connected network of RDF stores and services with all this imported data, how can we explore it? The main workspace UI tools provide a view a user's own knowledge base but a key objective of the application is to let users benefit from the work of colleagues. The primary tool for this is the *community browser.*

The community browser appears like any other workspace tool within the application. It displays a collection of items, new items can be dragged or copied into it, and when an item is selected an appropriate multi-tabbed item view is shown.

The key additional functionality of the community browser is that once an item is selected a set of predefined community-based queries can be selected from a menu. The results of these queries get added to the item collection view in a tree-like layout:



Workspace: Community browse - Thu Oct 10 13:57:07 BST 2002

Keywords: UDDI

```
communityKeywordSearch (UDDI - DaveR's KB)
  http://oss.software.ibm.com/developerworks/projects/ud
communityKeywordSearch (UDDI - steve_linux)
  Welcome to jUDDI.org
    inBookmarkFolder (steve_linux)
      UDDI & SOAP
        folderContains
          Welcome to jUDDI.org
          SoapWare.Org Implementations
          Clean up your wire protocol with SOAP, P
          Clean up your wire protocol with SOAP, P
          Welcome to ZOAP
    dmozCategory
      Dmoz: Implementations
    communityAnnotation (steve_linux)
      Welcome to jUDDI.org
  UDDI for Java
  UDDI for Java
```

Query predicate

Name

inBookmarkFolder (steve_linux)

Knowledge source consulted

steve_linux

Comment

Query the whole community for bookmark folders containing this item

Query

[] bookmark:containsBookmark [item:contentUri ${item:contentUri}]; rdf:type []; bookmark:bmTitle []

The tree is striped – it alternates between collections of viewable items and pseudo-relations that represent the community queries themselves. If the relation link is selected (as in the figure above) the query specification is displayed including the full QBE query used to generate the results.

- The queries are specified in a DAML file along with a schema for query description. In this way additional queries can easily be written and shared.

The queries we defined were:

- **hasWorkspace (ePerson)** – from a ePerson id find all of the workspaces that that person has allowed to be externally visible;
- **communityAnnotation (item)** – find all the properties of the given item which are defined across the community, this might include multiple annotations and ratings from different people;
- **communityKeywordSearch (keyword)** – this takes a text keyword and retrieves all items in the community which have a property whose value has the keyword as a substring;
- **ePersonKeywordSearch (ePerson, keyword)** – same as communityKeywordSearch but restricted to the particular ePerson currently selected;
- **inBookmarkFolder (item) –** find all the bookmark folders, across the community, which contain this item;
- **folderContains (folder) –** find all the items contained in this bookmark folder
- **inWorkspace (item)** – find all the user workspaces across the community that contain this item (note that a remote workspace can be opened directly from the workspace view pane so further queries to list workspace content are not needed);
- **dmozCategory (item)** – find the DMOZ categories into which this item could be assigned;
- **communityItemsInCategory (category)** - find all items across the community which have been classified as having this DMOZ category.

5.5.8 Application layer – assessment and lessons

Our “RDF everywhere” principle has a number of interesting implications for the UI. One obvious consequence is that the application represents persistent UI state in RDF. A rather more significant example is provided by the item dependent (and property dependent) views. These views (for example the bookmark view) can be determined by an item’s `rdf:type` via a set of UI presentation specifications (in RDF). Further details can be found in Appendix 5, but the key point is that the representation of item classes within a concept hierarchy allows a configurable and data-driven UI. This approach worked well for us, although there were some problems with event handling (see Appendix 5), mainly due to the interactions between Swing components and RDF wrapper objects (see below). We also encountered a need for richer inferencing in the underlying RDF toolkit, as explained in §5.3.3.

We did consider a number of other paradigms in this space. For example, Javascript could be used to provide a rich mechanism for sharing configuration machinery. Unfortunately the Javascript bindings are currently less than ideal, and this area remains a topic for future investigation.

We adopted as a user model an abstraction over RDF. Rather than presenting a property-centric view of the world, the UI in essence treats RDF as structured objects. It is possible that this limits the number of actions that have a natural UI representation, but we found that the user model was more than adequate for our needs. Indeed, even if we provided a property-centric view as an alternative model, it is far from clear that the benefits would outweigh the additional cognitive load.

A related topic is that of interface style. We tried a number of styles, some of which were particularly successful and were reused. Often, some evolution was required before we arrived at a sufficiently general design, but in general, the conceptual and engineering reuse of interface styles worked well. An example can be seen in the community browser, which, even though it represents quite different data, nevertheless has style consistent with the bookmark and DMOZ views.

Another issue is the use of intermediate models and caches. For example, UI components such as lists require some model behind them. This model needs to be in memory for the UI performance reasons. In addition, a raw RDF representation is not sufficient for certain representations (such as tables of properties) where we expect the properties to be returned in a consistent order. Finally, there are some properties that are in some sense privileged (e.g. item label, which should be exposed to the UI through an object’s `toString()` method) and would benefit from access via convenience methods. For all these reasons we built wrapper objects around items which solved the problem at the cost of extra UI complexity. This issue is also discussed in §6.1.2

An interesting issue arises from the concept of workspaces as containers. The advantage of this view is that the user is quite familiar with the idea from common file manager applications. Hence, an item can be selected, moved or copied somewhere else. However it is not the content that is manipulated but the metadata. A more refined concept would be to treat items as akin to UNIX soft links, or Windows shortcuts. Even this analogy is not perfect, however. Firstly, remote items are not treated as ‘real time’ links (indeed, the *automatic* propagation of metadata between peers is not really addressed in Snippet Manager[19]). Secondly, the application does not always make a distinction between container level operations ('delete from workspace') and more subtle metadata operations ('delete from folder')[20]. Even if it did, this distinction would further add to the complexity of the user model.

The community browser offers an alternative to the 'workspace as container' concept. It presents a filtered view onto the community owned items, and is set up for directed browsing, rather than manipulation and organisation of the contained items. One could imagine more radical extensions to the workspace concept, such as persistent queries (analogous to SQL views), in which the content can change without user intervention. Whether these shifts in working mode are intuitive to the casual user remains to be seen.

The Snippet Manager application allows the user to capture individual items in a reasonably lightweight manner. However, it would be better to integrate the data capture more with user activities (for example a 'snippify this item' button on a browser). The import tools are useful, but again it would be better to integrate them more tightly to a user's normal workflow. In general, integration with other tools is a weakness of the application.

The UI provides a number of mechanisms to generate new metadata, particularly via drag and drop. Whilst one could certainly imagine richer facilities (for example, content analysis of dropped text segments, metadata extraction from Word documents) the application illustrates the importance of the principle. Certainly our (user) experience suggests that adding data through the default methods is rather too tedious to be widely used, and that user assistance and automation are essential.

Finally, it is worth emphasizing the social filtering aspect of this work, in particular our aspirations for a richer model of sharing: "*share data, share metadata, share organization*". The application (and in particular the community browser) demonstrates an ability to share data and metadata. In addition, the classification service allows a user to discover and reuse a form of organisation. However, this sharing of organisation is rather coarse grain. In particular, it is not sufficient to really test our ideas about emergent ontologies.

In summary, it was challenging to implement a UI that both provided a natural interface and tested (some of) our intuitions. In fact, we have only partially achieved this goal, and richer user abstractions are still required, in particular to navigate larger communities and to clearly differentiate between metadata and content. Nevertheless, we feel that we have made a good first step in exploring appropriate user models for a semantic web application.

[19] For example, a copied remote item's metadata will stay in synchrony with its twin. But the same item (by URI) not wrapped by the same Java object (e.g. in a different remote location) will not receive metadata changes. There is partly a technical issue here (we had talked about, but not implemented, a 'lazy update' mechanism). But there is also a conceptual issue in that our user model does not clearly differentiate between identical (possibly remote) snippets, and distinct snippets 'about' the same content.

[20] Thus, for example, if an item appears in 2 folders, then deletion in one folder will remove the item from the workspace completely (and thus, apparently, both folders). Actually, the `inFolder` annotations are not removed, so that if the item is reintroduced to the workspace later, then the item will reappear in both folders.

6 Discussion: successes, issues and lessons

The Snippet Manager and ePerson infrastructure required the development a rich collection of system components. The work of developing a complete first cut implementation of this system has given us many insights into the design issues, along with several solutions and tools that are of practical benefit now.

The goal of the phase of the work reported here was to build a first complete, functioning implementation of the Snippet Manager application and supporting ePerson infrastructure. Against this goal we were very successful. The infrastructure and application are sufficiently functional and stable to allow our local work group to import bookmarks and other snippets and discover interesting relationships between items. This was a successful demonstration of feasibility.

This first implementation is not, and was not intended to be, a production system suitable for daily use. The plan was to build a test system, review it and then build a second-generation system, which could deliver on the original vision. The test of this second generation system would be to deploy it to users outside of our local workgroup and perform a user-based evaluation of whether our hypothesis, as described in section 1, has been validated or not.

We have tried to highlight the lessons and insights for each subsystem as we have described it above. In this section we discuss those system level issues that have not already been covered.

6.1.1 Overall system lessons

The overall system functionality, as exposed through the current UI, is sufficiently rich to show value in our approach and has enabled individuals to discover linkages between bookmarks that weren't apparent beforehand. However, this first prototype is not sufficiently functional to become an everyday tool for our workgroup.

Integration with current tools

The chief barrier to using the system in routine work is the lack of integration with existing tools. For example, in bookmark management, we support bulk import/export and drag-and-drop addition of single bookmarks but there is no live dynamic link between the ePerson bookmark store and a user's normal browser. The alternative bookmark organizations created or discovered within the ePerson framework are not visible from the browser only from the Snippet Manager; conversely new bookmarks or folders created from within the browser only become visible after a fresh import process.

Configuration complexity

The Snippet Manager, like many semantic web applications, will have greater value for individual users if there are more users of the system in total. This implies we should have as low a barrier to entry as possible – it should be possible for new users to start using such an application with as little initial investment in installation and configuration as possible. The current design requires installation of a Java application, locating (or running your own) kbhost service and creating an initial ePerson identity via a simple web form. The cost of this could be quite low given good installation scripting, but having to install any application can be a barrier. Switching to a web-based front end may lead to a less effective UI, but would make it easier for users to experiment with using the system.

For other workgroups to deploy the application, they would to need have access to a Jabber server and configure the transport layer to connect to that. Alternatively, on a local network configuration, the local UDP broadcast protocol is effective. Even if we didn't switch to a web-based UI, then switching to a more universally available transport like HTTP and changing our distributed query processing to not require a broadcast infrastructure would be worth considering.

Scaling

Whilst not an issue for the current deployment levels there are several scaling issues that would need to be addressed in a full scale implementation.

The key scaling issue is broadcast and discovery. We have already discussed the issues with broadcast, our attempted solution using a discovery server and the limitations of that design in section 5.2.4. A decentralized system with adaptive index nodes would permit greater scaling.

The second scaling issue is that of distributed query. Within a small-scale system, we can send an RDF query to the whole of the target community and aggregate the results. With a larger installation, a query routing which chose a subset of the community most likely to be able to add useful information on the query would be preferable. This is a key research topic for the future.

6.1.2 Consequences of the extensive use of RDF

We consciously and deliberately used RDF (including DAML) to store all of the persistent information in the Snippet Manager prototype. In this section we discuss the consequences, good and bad, arising from this discussion.

Many models

The ePerson architecture in general assumes that client applications and data-holding repositories are distributed throughout a peer-to-peer network. In practise, this means that the single set of data held by the repository has to be replicated on the client side for processing. We chose Jena's `Model` as the appropriate client-side container for the fragments of the server-side KB containing the full data set. Such fragments are created, for example, when the properties of an `Item` are cached in the client while the item in on display. This policy creates very many models on the client side, which has an effect on the efficiency of the client, but also complicates client-side programming. Several bugs were traced to a mismatch between models, for example:

```
localModel.getProperty( r, p ) vs.
r.getProperty( p )
```

produces different results if resource `r` is not currently bound to model `localModel`.

Models and resources as Java data-structures

It is convenient to think of an in-memory RDF model (Jena's `ModelMem` class) data-structure. For example, a Swing `Jlist` in the user interface has to be backed by a data structure containing the items to be displayed in the list. However, RDF models are unordered collections, so cannot be used, for example, to provide a collection of sorted items. In fact, there is no guarantee that the order in which resources are returned from a listing method (such as `listSubjects`) is consistent from one call to the next.

One solution to this need is to copy the resources into an ordered data structure. However, this then duplicates the RDF model as a data container, with the loss of the particular RDF capabilities from the model API. Alternatively, a wrapper using the delegate pattern could wrap the RDF model, and augment the normal access modes with modes that provide consistent ordering. This is problematic because of the complexity of the model API.

The solution we adopted was a compromise between these approaches. The `Item` class presents a view onto a model that contains the details of a particular item. An optional wrapper (`ItemWrapper`) wraps Item and delegates the interface methods to the Item class, with the addition that an ordering table imposes a fixed order on the resources, and access to resources by integer index. In general, we can see that many applications, but particularly including GUI presentations, need predictable, linearised access to the resources in a model.

Caching issues
Accessing information directly from the ePerson knowledge base is too slow to be useful to the client. Network latency and overhead simply do not permit sufficiently fast response times for an interactive application. We can view the client-side RDF models as caches. However, this raises the standard concerns about caches (invalidation, write-through, etc), and makes it the client programmer's responsibility to maintain the validity of the cache. It also does not allow caching between queries, unless the programmer chooses to cache the results of the query in a locally shared model. An alternative approach, that we discussed but did not implement, is to allow the client-side query processor to cache the results of queries and re-use these in subsequent queries. A challenge with such an approach is to recognise and re-use overlapping or subsuming queries, otherwise cache hits will be limited to instances of the re-issuing of a query identical to one in the cache, which will presumably be rare.

Flat database structure (see also §5.3.3)
In our current architecture, the knowledge-base that is the repository of the individual's knowledge is a single flat structure. No segregation exists between items, tool configuration, identification credentials, and personal profile. This flat structure was helpful during the development of the prototype, since a single set of query tools could be used to access all of the different stored data. There were costs to this choice, however, which became apparent during the development of the software. First, it was often useful or necessary to flush old or incorrectly structured knowledge from the KB (for example, while trying to get the right set of statements to record the hierarchy of the user's bookmarks in the KB). There was no easy way to selectively remove a region of the KB without affecting other persistent information, and we evolved a strategy of keeping one or more shadow copies of a clean knowledge base to allow re-starting the server but without the need to re-register each user. Secondly, in a flat structure it is not easy to optimise storage strategies according to the characteristics of the data. Finally, it was hard to inspect a dump of the

knowledge base while debugging, especially as Jena does not attempt to localise groups of related statements in the serialised output.

A future version of the ePerson architecture will probably segment the different categories of data in the KB, and furthermore should provide more tool support to assist with managing the data both during and after the software development cycle.

Inferencing models
Our ontology files (see above) although encoded in DAML, make relatively little use of the representation capabilities of the DAML language. This is largely because we don't, at the present time, have a tool that can make use of much of this information. Jena does include some support for processing DAML ontologies, and is able to follow links of transitive queries and recognise equivalences. However, the DAML support is disjoint from the query processing, and is limited to models that are stored in-memory. These two factors ruled out the direct use of the DAML support, which in turn meant that we had no way of inferring answers to queries that should be relatively straightforward. An example is sub-classing: assume that two classes, A and B exist, and B is a sub-class of A. Asking a standard Jena model to list all of the instances of A (for example, all `Item`s) will not deliver any instances of B, so a list of Items would not automatically include `BookmarkItem`s. This is clearly not desirable. In the absence of any inference support, we found it necessary to include extra information in the knowledge base, for example directly asserting the information contained in the closure under inference. So, a given item resource would be asserted to have `rdf:type BookmarkItem`, **and** `rdf:type Item`. This is expensive in storage and transmission costs, very error-prone, and largely impossible to update retrospectively should the ontology schema change.

An alternative strategy for inferencing over the class hierarchy, which was implemented on a limited basis in the current prototype, is to manually traverse the class- and property- hierarchies using custom queries. While this clearly works, it is too brittle and unwieldy to be generally useful.

6.1.3 Ontology issues

A number of issues relating to working with ontologies in general, as opposed to DAML/RDFS in particular, also became apparent during the project.

Firstly, we tried to re-use existing ontologies wherever possible. This is consistent with the vision that the use of an explicit ontology helps to build shareable knowledge structures. However, in practice, it is difficult to avoid getting locked-in to a closed world of ontologies whose existence is assumed. This is particularly so if the ontology is being used to provide a first-class representation of internal data structures. However, it is also true even when using ontologies as formalizations of concepts, such as the personal details in a user profile. Appendix 4 explores this issue in detail. An example of merging terms from similar, but not identical, ontologies is given in [23].

A related issue is that of needing access to concepts which are very general, and likely to be shared in many conceptualisations. A concept of a point in time, or a duration, is needed when making assertions about meetings, or the creation date of an item, or many similar applications. We can, and in fact did, define our own micro-ontology of instants and durations. However, such a local definition is likely to be inconsistent with definitions from other sources, and, in the case of time, rather limited with respect to the deep thinking that has been addressed to this question by philosophers and knowledge engineers. An alternative is to make use of a shared upper ontology of core concepts [24]. Such upper ontologies, however, are very complex and require detailed understanding.

Finally, we noticed a clear tension between the need to build general structures, and simple structures. For example, a common piece of metadata is a subjective rating by one person of an item. A typical rating scheme rates resources on a scale of 1 (bad) to 5 (good), and this would be very easy to define in an ontology. However, other rating schemes use different scales, sometimes reversing the polarity (so 1 is good and 5 bad), and sometimes using a different range (say, -1 to +1). It is easy to see that, given the appropriate information, it would be easy to translate between these rating schemes. However, a ontology structure that is general enough to record these variations appears complex and unwieldy for the simple case of standard ratings.

6.1.4 Performance issues

The ePerson system was our first large scale application of semantic web technologies in general, and the Jena toolkit in particular. The personal knowledge bases we are dealing with contain between 10,000 and 100,000 statements. It's not surprising, therefore, that we hit some performance issues along the way. This section highlights some of these, the techniques used to pinpoint the problems, and how they were overcome.

Opening a large workspace

The bookmark import tool allows us to create workspaces containing several hundred items. We initially found that opening such a workspace took about 40 seconds.

Some analysis of the transport layer debug logs showed that Snippet Manager was reading items from the remote KB one at time, hence the time to load large workspaces was dominated the remote query latency. It was possible to re-code the workspace item cache layer to use just two queries; one to identify the items in the workspace, and a second to retrieve all the statements about those items. This reduced the workspace load time from 40 seconds to about 8 seconds.

We then tried an experiment using a tool call AspectJ^{TM} [39]. AspectJ is a simple and practical extension to the Java that supports the aspect-oriented programming (AOP) paradigm. AOP allows developers to reap the benefits of modularity for concerns that cut across the natural units of modularity. In Java, the natural unit of modularity is the class. In AspectJ, aspects modularize concerns that affect more than one class. We defined an aspect to insert profiling code at key places in the ePerson stack. The advantage over traditional Java profiling tools (like prof, hprof and eprof) is the ability to be selective about where to add profiling. Being selective is important, as is easy for the overheads of profiling to overshadow the effects being measured.

Using AspectJ allowed us to accurately track where the remaining 8 seconds was being spent. The results are shown below:

Comment	*Time (secs)*
Various initial queries (5 small)	0.900
Idle while big query being done	1.497
All SAX Parsing (SAXBuilder.build - 0.263 is big query)	0.442
All RDF Parsing (Model.read - 3.889 is big query)	4.442
Loop creating 414 items in workspace	0.241
Loop adding 414 items to bookmark view	0.291
Loop adding 414 items to bookmark view	0.314
Total time to load and render the workspace.	8.127

The above sequence actually involves 14 separate queries to the KB. Most were to fetch workspace and item presentation information, and consequently were small. The one that fetched the item data was large (912KB). Almost half the time to load the workspace was spent in the Jena RDF parser, parsing this one query. Some minor tweaks to the parser (replacing Strings by StringBuffers) reduced this from 4.442 seconds to 1.093 seconds, roughly halving the time to load a large workspace.

The above exercise leads us to two conclusions:

1. *Transparent client-side caching is needed in a RDF network API*

   The latency when retrieving RDF statements from a remote KB is two to three orders of magnitude slower than when retrieving statements from an in-memory model. Caching is critical to improving system performance. Ideally, this caching should be implemented transparently in the client side of the RDF network API. For ePerson we had to code the caching explicitly in the application; consequently this was somewhat ad hoc and was sometimes ineffective.

2. *RDF parsers need to be better optimized for throughput*

   The throughput when retrieving large RDF result sets is an order of magnitude slower than 100 Mbps network speeds. Most of this overhead results from RDF parsing. Even after making some optimizations, the N-Triple parsing took significantly longer than the XML parsing.

Other areas for performance tuning

There are several other areas where we have seen performance problems. Although we have not analyzed these in detail, the approach outlined above could be used. These areas are simply listed here for completeness:

- Loading the system ontologies at start-up takes about 10 seconds. Batch or parallel loading of these may help.
- Enabling message signing increases start-up time by about 6 seconds. This may be an unavoidable overhead of initializing a Java Cryptographic Provider, but some vendor's providers may be better than others.

7 Related work

There is a substantial body of literature on the issues of personal and workgroup information management and knowledge management, it is beyond the scope of this report to review any significant fraction of this literature. Instead we will highlight a few of the existing systems and research endeavours that seem to share substantial overlap with the ePerson philosophy.

First, the name *ePerson* itself was inspired by the work of Martin Röscheisen [28]. In Röscheisen's conceptualisation, the ePerson was an active agent that stored user policies with respect to digital rights management issues. It would store the set of rights management contracts entered into by a user and could negotiate terms of such contracts. Whilst not directly related to the problems of information management, the vision of an active user agent that stores explicit user preference information to help mediate between the user and web-based services was part of the initial inspiration for the ePerson project.

Amongst current research activities, the most closely related project that we are aware of is probably the Haystack project at MIT led by Prof. David Karger (see, for example, [29]). Haystack uses a similar approach of representing all structured information in a common semi-structured data format and has similarly chosen RDF for this. They also have a rich user interface that is driven from both the RDF data itself and from ontology or schema files defining the data models. In Haystack, as in ePerson, the type of the item being displayed drives the rendering of items and the mapping from category to view is declaratively defined.

Haystack is attempting a more all-encompassing information management tool that the ePerson Snippet Manager. Whereas in our case we are primarily concerned with annotation, rating, classifying and filtering of small information items, Haystack seeks to offer a complete information management environment. Haystack supports traditional personal information management (PIM) tasks, such as calendar and task management, as well as snippet and document discovery. It has correspondingly less support for annotation, ratings and manual classification and organization of data, and more support for actions that can be performed on information items and full integration with all relevant data sources. One way of characterising this is that we are primarily focused on the use and exchange of metadata, whereas Haystack supports many forms of semi-structured data: both meta-level and content level.

However, the biggest difference is probably in research emphasis and community support. Our primary interest is in cross-community information management, hence the importance of discovery, distributed query, metadata and community browsing capabilities. We claim no strong research innovations in the area of UI design. In contrast, the primary focus for Haystack is the "semantic UI" approach to information management tools - issues of community sharing and exploration are a possible future direction rather than a core part of their work.

The Edutella project [32] is in many ways closer in spirit to the ePerson approach than Haystack. Edutella is aimed at the discovery of educational resources through the peer-to-peer (p2p) exchange of metadata expressed in

RDF. Like ePerson, Edutella is developing a peer-to-peer RDF query infrastructure on top of a transport abstraction. In the case of Edutella they use the Sun JXTA framework as their transport abstraction. In the early stages of our work we did experiment with JXTA but had problems making it work well in our local networking environment. In future versions, we would expect to be able to plug JXTA in as another connector type alongside Jabber and raw TCP within our transport API framework. The Edutella project has developed a hierarchy of query capabilities, with the base level corresponding roughly to our QBEHigher levels offer disjunction, negation, and eventually recursion. Our ability to quantify over properties and namespace-constrained properties, plus our regular expression matching, go beyond the base level offered in Edutella, and it is not clear where they fit within the hierarchy. Edutella goes beyond the existing ePerson infrastructure in defining two levels of wrapper-mediators. The base level is primarily a wrapper that translates to a common RDF data model and is similar in spirit to our `KnowledgeProvider` interface. The higher-level mediators support assembly and distribution of queries across sources, which go beyond our current functionality. The target application for Edutella appears to be primarily distributed access to institutional scale repositories, but the infrastructure could well be applicable to the personal scale repositories we are investigating.

Another related area of work we should highlight is the work of the digital library community in devising ways of aggregating metadata from several sources. A key innovation in recent years has been the introduction of very practical but scalable common interfaces for metadata aggregation, especially the Open Archive Initiative (OAI) [30]. Whilst this work was originally aimed at integrating data from small numbers of large library collections, the Kepler project [31] at Old Dominion University has explored its application to integration of large numbers of small personal collections. This system uses a centralized registration server to link the individual repositories to the service providers that perform the indexing. The data model for annotation and indexing is less flexible than the RDF-based approach advocated here. The primary interoperable metadata is Dublin Core and whilst other formats can be expressed and harvested, they need to be expressible as well formatted metadata packets, defined by some XML schema. It is unclear whether the Kepler *archivelet* implementations support more than the base Dublin Core data. The centralized registration and indexing approach of Kepler is appealing for mid-scale systems. There may be scaling problems for very large collections of sources though Google has demonstrated that centralized systems can be remarkably scalable given sufficient funding! Finally, given that a substantial fraction of the snippets we have actually stored in our repositories are web bookmarks, it is important to acknowledge existing work in bookmark sharing and organization tools especially the work of Soumen Chakrabarti on Memex [34], [35]. Memex uses a web proxy to collect browsing history (similar to our history store) and then offers automated clustering tools to allow visited sites and bookmarks to be organized into categories. This work has a much stronger emphasis than ours on automatic learning of categories and correspondingly little emphasis on sharing of direct user classifications, annotations and ratings.

8 Conclusions

In summary, we have developed a complete, functioning infrastructure for personal/workgroup information management building upon semantic web techniques, along with a functioning prototype application – the Snippet Manager.

The infrastructure supports the key features of:

- transport-independent addressing and message API with support for broadcast/multicast and for message signing;
- distributed RDF query, based on a query-by-example pattern matching style;
- a name resolution and service discovery infrastructure based on broadcast discovery of RDF self-description records;
- a personal knowledge-based hosting infrastructure, supporting role-based access control;
- networked RDF sources providing access to the DMOZ classification hierarchy and the DMOZ page classifications themselves ;
- a vocabulary for representing user profile information with interoperability hooks to related profile schemas which is populated both manually and from automatically inferred interest vectors derived from browsing records;
- a set of data models for representing items of user information ranging from generic snippets through to specific items such a bookmarks in which the users themselves and personal collections of items (workspaces) can themselves be treated as information items for annotation and indexing;
- a functioning user interface and prototype application allowing creation, organization and navigation of personal information repositories driven by the DAML files that define the data model;
- tools for import and export of bookmarks and drop-able items into the information repositories;
- an extensible tool for exploration of a linked community of information stores.

As a prototype implementation, this exercise has demonstrated feasibility of the approach and given some hints of its practical utility even though it is not yet suitable for routine practical use. We have identified many lessons and design issues for each of the system components and for the overall prototype. The information gained and captured here is relevant both to any future development of ePerson tools and to many related semantic web applications.

Our experiences from this experiment point to several avenues for future work. In terms of infrastructure, the critical area for further investigation is scalability. Our prototype solution is effective for a modest workgroup sizes but the barriers to a new participant joining the network should be lowered and the scaling of discovery (both server and data discovery) and distributed query would be worth further investigation. In terms of the application, it is the integration with other desktop tools (both at a technical level and in terms of UI metaphors) that would focus on. Finally, the ontology mapping issues revealed by the profile ontology experiment are important areas to address for many future semantic web applications.

9 References

[1] The ePerson project, HP Labs Bristol
http://w3.hpl.hp.com/eperson

[2] The Semantic Web Activity. **http://www.w3.org/2001/sw/**

[3] The Resource Description Framework. **http://www.w3.org/RDF/**

[4] The Web Ontology Working group.
http://www.w3.org/2001/sw/WebOnt/

[5] The DARPA Agent Markup Language. **http://www.daml.org/**

[6] The open directory project. **http://dmoz.org/**

[7] The Jabber software foundation. **http://www.Jabber.org/**

[8] The Jena RDF toolkit. **http://www.hpl.hp.com/semweb/Jena-top.html**

[9] The ePerson Javadoc archive.
http://w3.hpl.hp.com/eperson/infoAgents/apidocs/index.html

[10] Apache XML signature
http://xml.apache.org/security

[11] Simple public key infrastructure RFC2693.

[12] CNRI Handle System
http://www.handle.net/

[13] RDF Core working draft. N-Triples.
http://www.w3.org/2001/sw/RDFCore/ntriples/

[14] Ben Zhao et al, *Brocade: Landmark routing on peer-to-peer networks*
http://roc.cs.berkeley.edu/retreats/summer_02/slides/hling.pdf

[15] *Morpheus p2p file sharing*. See review at
http://www.ics.uci.edu/~isse/hollyshare/presentation4(survey).pdf

[16] DAML Validator
http://www.daml.org/validator/

[17] The Open Directory Project (DMOZ)
http://dmoz.org/

[18] Jakarta Lucene search engine
http://jakarta.apache.org/lucene/docs/index.html

[19] *vCard MIME directory profile*, RFC 2426.

[20] Dan Brickley, Libby Miller. *FOAF: the 'friend of a friend' vocabulary,*
http://xmlns.com/foaf/0.1/

[21] Dynamic Research Corporation
http://orlando.drc.com/

[22] DRC Orlando Person ontology.
http://orlando.drc.com/daml/Ontology/Person/current/

[23] Dejing Dou, Drew McDermott & Peishen Qi. *Ontology Translation by Ontology Merging* in Proc. EKAW Workshop on Ontologies for Multi-

Agent Systems, Sept 2002, Siguenza, Spain. pp3-18. Available from **http://cs-www.cs.yale.edu/homes/dvm/papers/DouMcDermottQi02.ps**

[24] IEEE P1600.1 Standard Upper Ontology (SUO) Working Group **http://suo.ieee.org/**

[25] Renato Iannella, *Representing vCard Objects in RDF/XML*, W3C Note 22 February 2001, http://www.w3.org/TR/vcard-rdf

[26] George A. Miller, *Wordnet a lexical database for the English language* **http://www.cogsci.princeton.edu/%7Ewn/**

[27] Sellen, AJ, Murphy, R and Shaw, KL. *How knowledge workers use the Web.* CHI 2002, April 20-25. 4(1), 227-234.

[28] Martin Röscheisen, *A Network-centric design for relationship-based rights management,* Stanford University PhD Thesis, December 1997.

[29] David Huynh, David Karger, and Dennis Quan. *Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF*. Draft submission to journal 2002. **http://haystack.lcs.mit.edu/papers/computer-network-2002.pdf**

[30] The Open Archives Initiatives, **http://www.openarchives.org**

[31] Kurt Maly, Mohammad Zubair, Xiaoming Liu, *Kepler - An OAI Data/Service Provider for the Individual*, D-Lib Magazine, April 2001, Volume 7 Number 4.

[32] Wolfgang Nejdl et al, *EDUTELLA: A P2P Networking Infrastructure Based on RDF,* WWW2002, May 7-11, 2002, Honolulu, Hawaii, USA.

[33] LiGong, *Project JXTA: A technology overview.* Technical report, SUN Microsystems, April 2001. **http://www.jxta.org/project/www/docs/TechOverview.pdf**

[34] Soumen Chakrabarti et al, *Using Memex to archive and mine community Web browsing experience*, WWW-9, 1998. **http://www9.org/w9cdrom/98/98.html**

[35] Soumen Chakrabarti, Yusuf Batterywala, *Mining themes from bookmarks,* KDD-2000 Workshop on Text Mining, August 20, 2000.

[36] RDQL: RDF Data Query Language, **http://www.hpl.hp.com/semweb/rdql.html**

[37] N3: Notation3 **http://www.w3.org/DesignIssues/Notation3.html**

[38] S. Agarwal, D. Starobinski, and A. Trachtenberg, *On the Scalability of Data Synchronization Protocols for PDAs and Mobile Device*s, IEEE Network Magazine, July/August 2002.

[39] AspectJ **http://aspectj.org**

Appendix 1 Ontology class hierarchy

The ePerson prototype made extensive use of DAML files to define an ontology of representation terms stored in the knowledge base. Reviewing our use of the DAML files, we can distinguish a number of uses for the ontology definitions:

- a namespace of well-known names for distinguished constants (classes, relationships and instances) that are referred to in the code base, and in pre-built queries;
- formally documenting the modelling assumptions that are used to build the conceptual model of the domain;
- formally documenting the associations between the RDF statements that form the modelling constructs embodying the conceptual model, of which a particular case is
- overlaying a frame-like view of structured objects formed from sets of RDF triples.

Given these multiple roles, it is perhaps unsurprising that our DAML files do not exploit most of the representational power available in the DAML language. In reviewing the class hierarchy (see below), we can see that most of the classes are distinct from each other. Clearly much of the modelling needs at the system level[21] could have been taken care of by RDFS.

It is also noteworthy that the prevalence of the 'RDF as structured objects' in modelling suggests that RDF toolkits, such as Jena [8], should offer direct support for this use case in their API.

The following two figures show the class hierarchy (i.e. the `rdf:subclass` and `daml:subclass` relationships) for the classes in the current prototype. In common with the other parts of the codebase, it is clear that a future version of the system could do much to regularise and improve the hierarchy; this is presented as is, rather than as an ideal standard. The property hierarchy is omitted due to lack of space.

[21] At the system level, but not at the user domain level, where the needs of ontology representation, dynamic development and transformation suggest that more representational power than DAML is needed, not less. A demonstration of this need can be found in Appendix 4.

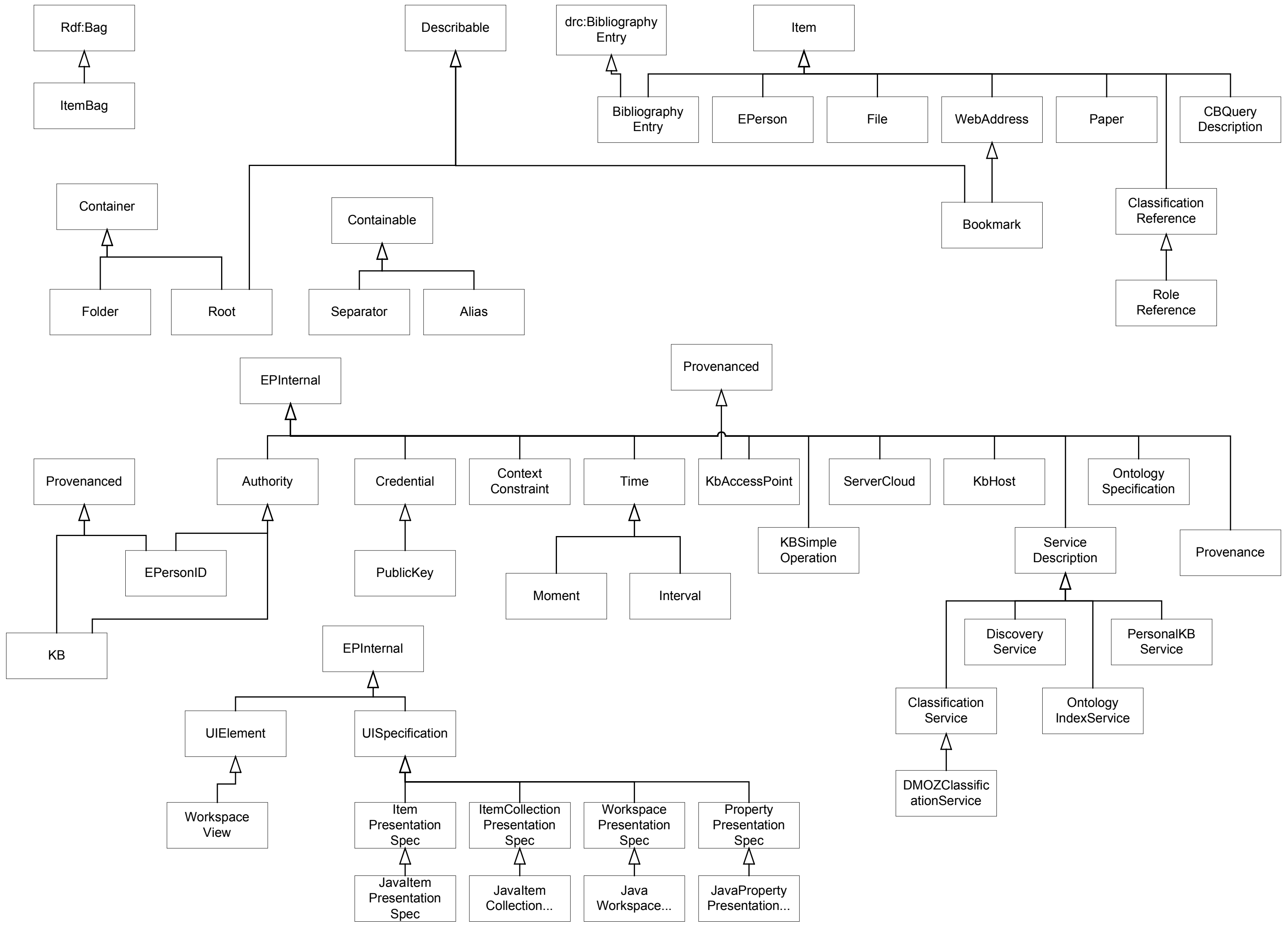


Figure 16 DAML class hierarchy part (i)

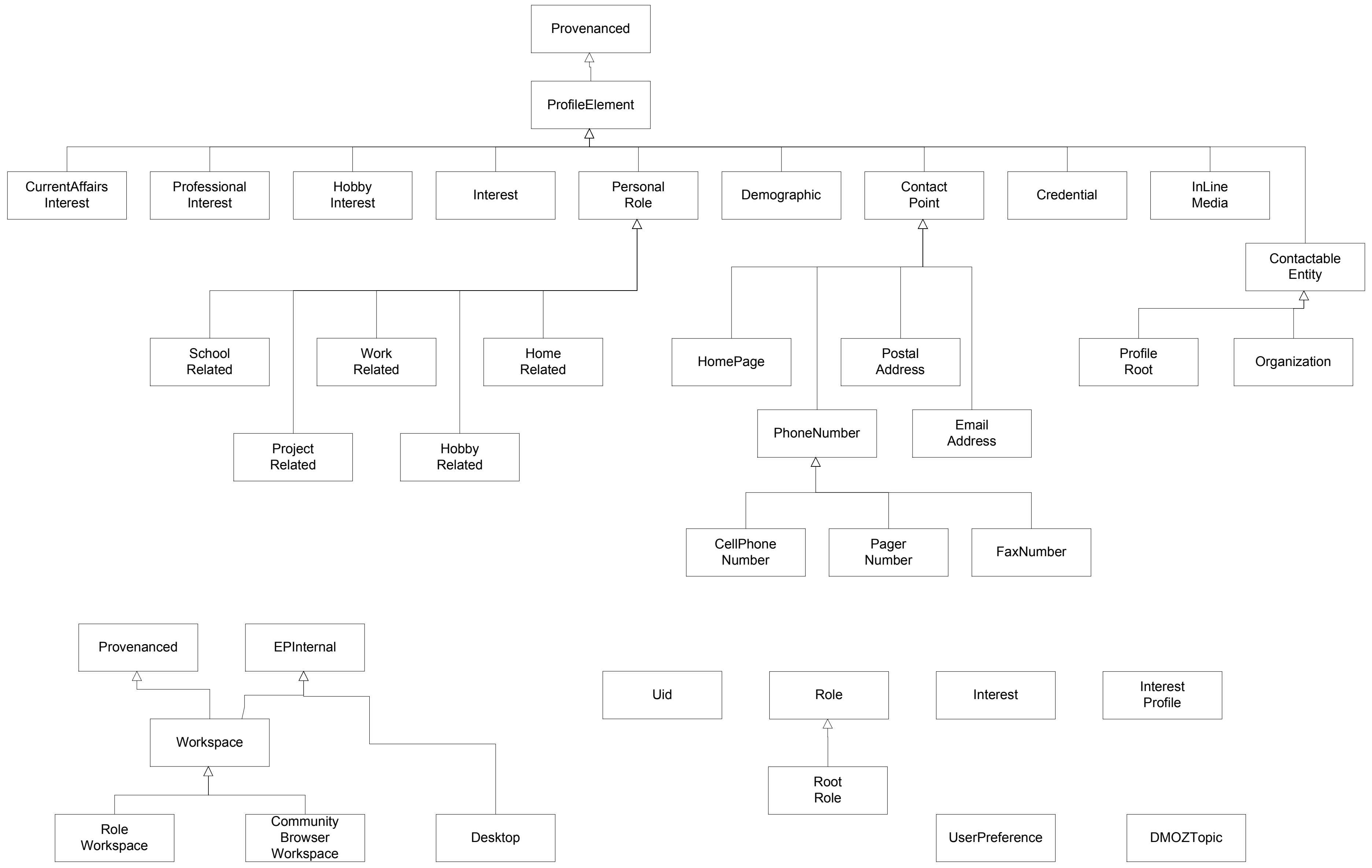


Figure 17: DAML class hierarchy part (ii)

Appendix 2 Example RDF for imported bookmarks

In this section we walk through the RDF generated when importing the following XBEL (XML Bookmark Exchange Language) file:

```
<?xml version="1.0"?>
<xbel>  <desc>No description</desc>
  <folder>
    <title>Starting Points</title>
    <bookmark href="http://www.google.com/" added="1019126208" visited="1022753710"
modified="994758191" >
      <title>Google</title>
    </bookmark>
  </folder>
  <bookmark href="http://quote.yahoo.com/q?s=hpq&amp;d=v1" added="1022749186" >
    <title>Yahoo! Finance - HPQ</title>
  </bookmark>
</xbel>
```

The RDF starts with various namespaces definitions:

```
1  <rdf:RDF
2  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
3  xmlns:item='http://w3.hpl.hp.com/eperson/daml/eperson/item#'
4  xmlns:classification='http://w3.hpl.hp.com/eperson/daml/eperson/classification#'
5  xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#'
6  xmlns:dmoz='http://dmoz.org/rdf/'
7  xmlns:kbaccess='http://w3.hpl.hp.com/eperson/daml/eperson/kbaccess#'
8  xmlns:presentation='http://w3.hpl.hp.com/eperson/daml/eperson/presentation#'
9  xmlns:bookmarks='http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#'
10 xmlns:workspace='http://w3.hpl.hp.com/eperson/daml/eperson/workspace#'
11 >
```

The next block declares that urn:x-hp-kb:dmoz provides a service (lines 12-14). The service described (lines 15-32) is the DMOZ classification service used when the bookmarks were imported. The service description of a classification service includes all the information necessary to build a ClassifiedItemCollectionView of the items - the root node, the classification property, the label property and the narrow property (lines 22-30). Finally, there is a pointer to a service, which can be used to classify more items in the future (line 31). This is a directClassifierKB, which means in this case the KB is expected to contain triples of the form: <URL> <property> <value>.

```
12 <rdf:Description rdf:about='urn:x-hp-kb:dmoz'>
13 <kbaccess:providesService rdf:resource='#A0'/>
14 </rdf:Description>

15 <rdf:Description rdf:about='#A0'>
16 <rdf:type rdf:resource=
17 'http://w3.hpl.hp.com/eperson/daml/eperson/kbaccess#ServiceDescription'/>
18 <rdf:type rdf:resource=
19 'http://w3.hpl.hp.com/eperson/daml/eperson/kbaccess#ClassificationService'/>
20 <rdf:type rdf:resource=
21 'http://w3.hpl.hp.com/eperson/daml/eperson/kbaccess#DMOZClassificationService'/>
22 <classification:narrowProperty rdf:resource=
23 'http://dmoz.org/rdf/narrow'/>
24 <classification:classificationProperty rdf:resource=
25 'http://dmoz.org/rdf/topic'/>
26 <classification:labelProperty rdf:resource=
27 'http://www.w3.org/2000/01/rdf-schema#label'/>
28 <classification:rootNode rdf:resource=
29 'http://dmoz.org/rdf/Top'/>
30 <classification:expandClassification>false</classification:expandClassification>
31 <classification:directClassifierKB rdf:resource='urn:x-hp-kb:dmoz'/>
32 </rdf:Description>
```

The next block declares that this KB provides a service (lines 33-35). The service described (lines 36-52) is a classification service derived from the user's bookmark folder hierarchy. Again, the service description of a classification service includes all the information necessary to build a ClassifiedItemCollectionView of the items (lines 41-49). Finally, there is a pointer to a service, which can be used to classify more items in the future (line 51-52). This is simply a pointer to back to this KB; however this time it is defined as an ItemClassifierKB, which means the KB is expected to contain triples of the form: <GUID> contentURI <URL> ; <property> <value>.

```
    33 <rdf:Description rdf:about='urn:x-hp-kb-sha:739f6a3c541a9d18a588f48e9c403447a144c625'>
    34 <kbaccess:providesService rdf:resource='#A5'/>
    35 </rdf:Description>

    36 <rdf:Description rdf:about='#A5'>
    37 <rdf:type rdf:resource=
    38 'http://w3.hpl.hp.com/eperson/daml/eperson/kbaccess#ServiceDescription'/>
    39 <rdf:type rdf:resource=
    40 'http://w3.hpl.hp.com/eperson/daml/eperson/kbaccess#ClassificationService'/>
    41 <classification:narrowProperty rdf:resource=
    42 'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#containsFolder'/>
    43 <classification:classificationProperty rdf:resource=
    44 'http://w3.hpl.hp.com/eperson/daml/eperson/classification#inFolder-82bd7c85-2738-51b2-
8000-8ef35df5bdbc'/>
    45 <classification:labelProperty rdf:resource=
    46 'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#bmTitle'/>
    47 <classification:rootNode rdf:resource=
    48 'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#82bd7c86-2738-51b2-8000-
8ef35df5bdbc'/>
    49 <classification:expandClassification>false</classification:expandClassification>
    50 <classification:itemClassifierKB rdf:resource=
    51 'urn:x-hp-kb-sha:739f6a3c541a9d18a588f48e9c403447a144c625'/>
    52 </rdf:Description>
```

To allow several bookmarks hierarchies to be held in the same KB, the classification property value used in the previous service description (line 44) must be different each time. Hence, it is GUID based. To allow consistent rendering of this property, the next block of RDF defines it as a sub property of a common parent (lines 60-61).

```
    53 <rdf:Description rdf:about=
    54 'http://w3.hpl.hp.com/eperson/daml/eperson/classification#inFolder-82bd7c85-2738-51b2-
8000-8ef35df5bdbc'>
    55 <rdf:type rdf:resource=
    56 'http://www.w3.org/1999/02/22-rdf-syntax-ns#Property'/>
    57 <rdfs:label>In folder</rdfs:label>
    58 <rdfs:range rdf:resource=
    59 'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#Folder'/>
    60 <rdfs:subPropertyOf rdf:resource=
    61 'http://w3.hpl.hp.com/eperson/daml/eperson/classification#bookmarkInFolderProperty'/>
    62 </rdf:Description>
```

The next block defines a Workspace to contain the imported bookmarks. The workspace is given a GUID based name (line 64) a type (line 65-66) and a label (line 67). It includes links to the contained bookmarks (lined 69-72).

```
    63 <rdf:Description rdf:about=
    64 'http://w3.hpl.hp.com/eperson/daml/eperson/workspace#82bd7c84-2738-51b2-8000-
8ef35df5bdbc'>
    65 <rdf:type rdf:resource=
    66 'http://w3.hpl.hp.com/eperson/daml/eperson/workspace#Workspace'/>
    67 <workspace:label>Bookmarks Example imported on Wed Oct 02 11:49:14 BST
2002</workspace:label>
    68 <workspace:layout rdf:resource='#A2'/>
    69 <workspace:contains rdf:resource=
    70 'http://w3.hpl.hp.com/eperson/daml/eperson/item#82bd7c88-2738-51b2-8000-8ef35df5bdbc'/>
    71 <workspace:contains rdf:resource=
    72 'http://w3.hpl.hp.com/eperson/daml/eperson/item#82bd7c89-2738-51b2-8000-8ef35df5bdbc'/>
    73 </rdf:Description>
```

The definition of how to render a workspace is held separately from the workspace itself, in a structure called a workspace presentation spec. The one used here is a more specific subclass - the Java workspace presentation spec - which allows the implementing Java class to be specified (line 80). In this case it is a DefaultWorkspaceView - a class that supports multiple tabbed views. There are initially two views defined (lines 82-83).

```
74 <rdf:Description rdf:about='#A2'>
75 <rdf:type rdf:resource=
76 'http://w3.hpl.hp.com/eperson/daml/eperson/presentation#WorkspacePresentationSpec'/>
77 <rdf:type rdf:resource=
78 'http://w3.hpl.hp.com/eperson/daml/eperson/presentation#JavaWorkspacePresentationSpec'/>
79 <presentation:JavaDefinitionClass>
80 com.hp.hpl.eperson.sm.ui.DefaultWorkspaceView
81 </presentation:JavaDefinitionClass>
82 <presentation:itemCollectionView rdf:resource='#A3'/>
83 <presentation:itemCollectionView rdf:resource='#A4'/>
84 </rdf:Description>
```

The first view is a ClassifiedItemCollectionView that presents the bookmark folder classification. Note the link to the classification spec defined earlier (line 93).

```
    85 <rdf:Description rdf:about='#A3'>
    86 <rdf:type rdf:resource=
    87 'http://w3.hpl.hp.com/eperson/daml/eperson/presentation#ItemCollectionPresentationSpec'/>
    88 <rdf:type rdf:resource=
'http://w3.hpl.hp.com/eperson/daml/eperson/presentation#JavaItemCollectionPresentationSpec'/>
    89 <rdfs:label>Bookmarks</rdfs:label>
    90 <presentation:JavaDefinitionClass>
    91 com.hp.hpl.eperson.classification.ClassifiedItemCollectionView
    92 </presentation:JavaDefinitionClass>
    93 <presentation:presentsClassification rdf:resource='#A5'/>
    94 </rdf:Description>
```

The second view is a ClassifiedItemCollectionView that presents the DMOZ classification. Note the link to the classification spec defined earlier (line 103).

```
    95  <rdf:Description rdf:about='#A4'>
    96  <rdf:type rdf:resource=
    97  'http://w3.hpl.hp.com/eperson/daml/eperson/presentation#ItemCollectionPresentationSpec'/
>
    98  <rdf:type rdf:resource=
'http://w3.hpl.hp.com/eperson/daml/eperson/presentation#JavaItemCollectionPresentationSpec'/>
    99  <rdfs:label>DMOZ</rdfs:label>
    100 <presentation:JavaDefinitionClass>
    101 com.hp.hpl.eperson.classification.ClassifiedItemCollectionView
    102 </presentation:JavaDefinitionClass>
    103 <presentation:presentsClassification rdf:resource='#A0'/>
    104 </rdf:Description>
```

Next, we get to root folder of the bookmark folder hierarchy. This contains one folder (lines 110-111) and one bookmark (lines 112-113). If you look at the classification spec (line 48) you will see this folder (line 105) is the root node of the bookmark classification hierarchy.

```
    105 <rdf:Description
rdf:about='http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#82bd7c86-2738-51b2-8000-
8ef35df5bdbc'>
    106 <rdf:type rdf:resource=
    107 'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#Root'/>
    108 <bookmarks:title>Bookmarks</bookmarks:title>
    109 <bookmarks:orderedContents rdf:resource='#A6'/>
    110 <bookmarks:containsFolder rdf:resource=
    111 'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#82bd7c87-2738-51b2-8000-
8ef35df5bdbc'/>
    112 <bookmarks:containsBookmark rdf:resource=
    113 'http://w3.hpl.hp.com/eperson/daml/eperson/item#82bd7c89-2738-51b2-8000-8ef35df5bdbc'/>
    114 </rdf:Description>
```

The next block (which we actually ignore) is an RDF Sequence, and is used to preserve the original order of the items in the root folder. In general one of these will overlay each imported folder.

```
    115 <rdf:Description rdf:about='#A6'>
    116 <rdf:type rdf:resource=
    117 'http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq'/>
    118 <rdf:_1 rdf:resource=
    119 'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#82bd7c87-2738-51b2-8000-
8ef35df5bdbc'/>
    120 <rdf:_2 rdf:resource=
    121 'http://w3.hpl.hp.com/eperson/daml/eperson/item#82bd7c89-2738-51b2-8000-8ef35df5bdbc'/>
    122 </rdf:Description>
```

Here's the definition of the sub folder ("Starting Points"). It contains one bookmark (lines 128-129) and is itself contained in the root folder (130-131). Note the use of the GUID based inFolder property.

```
    123 <rdf:Description
rdf:about='http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#82bd7c87-2738-51b2-8000-
8ef35df5bdbc'>
    124 <rdf:type rdf:resource=
    125 'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#Folder'/>
    126 <bookmarks:bmTitle>Starting Points</bookmarks:bmTitle>
    127 <bookmarks:orderedContents rdf:resource='#A1'/>
    128 <bookmarks:containsBookmark rdf:resource=
    129 'http://w3.hpl.hp.com/eperson/daml/eperson/item#82bd7c88-2738-51b2-8000-8ef35df5bdbc'/>
    130 <classification:inFolder-82bd7c85-2738-51b2-8000-8ef35df5bdbc rdf:resource=
    131 'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#82bd7c86-2738-51b2-8000-
8ef35df5bdbc'/>
    132 </rdf:Description>
```

Again, there is an RDF Sequence overlaying this folder to preserve the order of the items contained in the folder. This is redundant in this case, since there is only a single item.

```
    133 <rdf:Description rdf:about='#A1'>
    134 <rdf:type rdf:resource=
    135 'http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq'/>
    136 <rdf:_1 rdf:resource=
    137 'http://w3.hpl.hp.com/eperson/daml/eperson/item#82bd7c88-2738-51b2-8000-8ef35df5bdbc'/>
    138 </rdf:Description>
```

Finally, here's the definition of the first bookmark item - with type Bookmark (a subclass of Item). Note that as this is now an Item, it is referenced by a GUID and the original bookmark URL is retained using the contentUri property (line 146). Other bookmark metadata is retained (lines 142-145). Its classification in the bookmark hierarchy is defined on lines 148-149. Its classification(s) in the DMOZ hierarchy is defined on lines 150-153.

```
    139 <rdf:Description rdf:about='http://w3.hpl.hp.com/eperson/daml/eperson/item#82bd7c88-
2738-51b2-8000-8ef35df5bdbc'>
    140 <rdf:type rdf:resource=
    141 'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#Bookmark'/>
    142 <bookmarks:added>Thu Apr 18 11:36:48 BST 2002</bookmarks:added>
    143 <bookmarks:bmTitle>Google</bookmarks:bmTitle>
    144 <bookmarks:modified>Tue Jul 10 10:43:11 BST 2001</bookmarks:modified>
    145 <bookmarks:visited>Thu May 30 11:15:10 BST 2002</bookmarks:visited>
    146 <item:contentUri>http://www.google.com/</item:contentUri>
    147 <item:label>Google</item:label>
    148 <classification:inFolder-82bd7c85-2738-51b2-8000-8ef35df5bdbc rdf:resource=
    149 'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#82bd7c87-2738-51b2-8000-
8ef35df5bdbc'/>
    150 <dmoz:topic rdf:resource=
    151 'http://dmoz.org/rdf/Top/World/Chinese_Simplified/???/????/??/????'/>
    152 <dmoz:topic rdf:resource=
    153 'http://dmoz.org/rdf/Top/Regional/North_America/United_States/California/Localities/M/Mo
untain_View/Business_and_Economy/Industries/Computers_and_Internet'/>
    154 </rdf:Description>
```

And here's the definition of the second bookmark item.

```
    155 <rdf:Description rdf:about='http://w3.hpl.hp.com/eperson/daml/eperson/item#82bd7c89-
2738-51b2-8000-8ef35df5bdbc'>
    156 <rdf:type rdf:resource=
'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#Bookmark'/>
<bookmarks:added>Thu May 30 09:59:46 BST 2002</bookmarks:added>
<bookmarks:bmTitle>Yahoo! Finance - HPQ</bookmarks:bmTitle>
<item:contentUri>http://quote.yahoo.com/q?s=hpq&amp;d=v1</item:contentUri>
<item:label>Yahoo! Finance - HPQ</item:label>
<classification:inFolder-82bd7c85-2738-51b2-8000-8ef35df5bdbc rdf:resource=
'http://w3.hpl.hp.com/eperson/daml/eperson/bookmarks#82bd7c86-2738-51b2-8000-8ef35df5bdbc'/>
<dmoz:topic rdf:resource=
'http://dmoz.org/rdf/Top/Computers/Software/Operating_Systems/Unix/BSD/FreeBSD/Prominent_Users'/>
<dmoz:topic rdf:resource=
'http://dmoz.org/rdf/Top/Computers/Internet/Searching/Directories/Yahoo'/>
<dmoz:topic rdf:resource=
'http://dmoz.org/rdf/Top/Business/Major_Companies/Publicly_Traded/Y'/>
</rdf:Description>

</rdf:RDF>
```

Appendix 3 QBE matching algorithm

Recall that the Query By Example (QBE) algorithm is designed to extract a subset of the statements from some RDF source that matches a query that is itself expressed as an RDF graph. The bNodes in the query graph are treated as variables and can match any resource in the source graph. We restrict the pattern to those graphs that are expressible using the current RDF/XML or N3 syntax, i.e. where the bNodes (variables) can only form trees. This restriction simplifies the matcher from a subgraph isomorphism problem (NP) into a tree match problem (polynomial) with a simple recursive implementation. We also introduce reserved properties that act as property wildcards (either global or restricted to a specific namespace) and reserved literals that perform regular expression string matching.

An example query which looks for all services which are of type `kba:KB` and returns their access information (address, port number, connection type) would be:

```
[] rdf:type kba:KB; kba:accessPoint [] .
```

where we are assuming that namespaces `rdf` and `kba` have been defined appropriately.

The query-by-example pattern match algorithm is specified as follows:

Inputs:

- P a pattern in the form of an RDF graph in which bNodes are only linked into tree (not graph) structures
- G an RDF graph to be matched against the pattern

Outputs:

- MG a subset of the statements from G that matches the pattern P.

Algorithm:

- Split P into a forest of tree patterns P_i where
  - each statement in P is in exactly one P_i
  - each P_i is either a singleton ground statement or a single tree of statements linked by bNodes (for all bNodes in P_i there is no statement in another P_j which references that bNode)
- For each P_i compute the matching subgraph MG_i as follows:
  - if P_i is a singleton ground statement S then MG_i = S if S is present in G otherwise it is empty
  - if P_i is a match tree with root node r_i then find all nodes n_{ij} in G which *couldMatch* r_i and set MG_i = union { *treeMatch*(r_i, n_{ij}) }
- let MG = union of all MG_i
- let closedMG = *bNodeClose*(MG, G)
- return closedMG

treeMatch(r, n) returns the set of statements in G, rooted at n which correspond to the sub-pattern rooted at r; this can be empty if n does not in fact match r

```
• let res be an empty set
• for each statement $S_i$ in P such that $S_i$ =<r, *, *>[22] {
      let $SG_i$ = statementMatch($S_i$ , n)
      if $SG_i$ is empty the match has failed and exit treeMatch with empty set
      if $SG_i$ is not empty res = res $\cup$ $SG_i$ and continue
  }
  return res
```

statementMatch(S, n) returns the set of statements in G, rooted at n which matches pattern statement S, it may be empty if match fails

```
• S is statement <r, p, o>
• let res be an empty set
• for each statement $SG_i$ in G such that $SG_i$ =<n, $p_i$, $o_i$> {
      we say that $SG_i$ matches S if p matches $p_i$ AND o matches $o_i$
      ▪ p matches $p_i$
            if p is a named property and p=$p_i$, OR
            if p is a wildcard, OR
            if p is a wildcard over namespace ns and $p_i$ is in namespace ns
      ▪ o matches $o_i$
            if o is a regular expression literal and $o_i$ is a string literal and
            the regular expression matches the string, OR
            if o is a string literal and o = $o_i$, OR
            if o is a bNode and $TM_i$ = treeMatch(o, $o_i$) is not empty
      if $SG_i$ matches S by this definition then
            let res = res $\cup$ $SG_i$ $\cup$ $TM_i$
  }
  return res
```

couldmatch(r) : returns all the nodes in G which could match with the pattern node r, this is a heuristic function which uses the most ground constraint in the pattern to give the smallest set of candidate matches to r.
If r is a ground node then there will only be one match.

bNodeClose(R, G) adds to graph R enough statements from G to ensure there are no dangling bNodes.

```
• let B be the set of bNodes referenced as the object of a statement in R
  which are not the subject of any statement in R
• for each node $n_i$ in B {
        for each statement S in G where S = <$n_i$, p, o> {
            add S to R
            if o is a bNode add o to B
        }
  }
• return R
```

[22] We used <s,p.o> to represent an RDF statement with subject s, predicate p and object o where * is used to indicate *don't care*.

Appendix 4 ePerson profile ontology – design issues

Introduction

Our vision of ontology development in the ePerson framework is that users should find it easy to discover and reuse terms from existing ontologies in labelling and structuring their own information. This should be sufficiently easy that the pool of ontologies for a given topic area converges to some reasonably stable set, rather than growing unbounded as each group develops their own from scratch. We have plans for tools that would assist in this discovery and reuse process, but the current prototype application does not have explicit support for user ontology development[23]. However, in manually developing the ontology files that we include in the existing implementation, we tried to pay attention to issues of ontology development and ontology mapping that arose, to use as inputs to future tool development.

In this section, we report on one example of such ontology development, which illustrates some of the ontology mapping issues and support requirements. The example we have chosen is the development of the user profile ontology[24].

The job of the profile ontology is to represent a user in the ePerson network – the owner of an ePerson ID. The primary use of this information is for informing other users so it needs to include information such as name, organisation and role, interests, skills, background, some depiction of the individual and how to contact them. We also use this as the basis for discovering and exploring social networks so some explicit representation of other users in the ePerson network that this user is connected to is useful. We also want to be able to use the profile as one means of adapting and customising services so that demographic information and quite detailed interest information is useful along with relevant cryptographic credentials. Finally, the profile should cater to many aspects of the user's life – work, home, school and hobbies.

Existing ontologies and some implications of their modelling choices

Several existing user profile or person description ontologies already exist.

The DRC group at Orlando [21] have a developed a rich set of interconnected DAML ontologies which includes a *person-ont* ontology [22]. This covers name, gender, US Social Security Number and contact point information (phone number, mail, address). It lacks several key areas for us (interests, colleagues, organisation, role). Parts of the modelling, such as the social security number, are US specific. Those areas that are covered, such as contact point information, lack context sensitivity – there is no separation of home, work email for example though there is separation of home/work phone numbers.

[23] Well, it is easy enough to add new DAML+OIL files into the list of known ontologies that all Snippet Manager instances load up (via the discovery service) – it is "just" that there is no support for developing the extension ontologies.
[24] We will use *schema* and *ontology* somewhat interchangeably here. At one level all we are doing is describing a complex data structure and so should call this a *schema*. However, we are also building a conceptual model of part of the world of sufficient richness that the term *ontology* is also reasonable.

The interconnectedness of the DRC ontologies is also an issue for us. The DRC set of ontologies is a standalone set with many internal references – the *person-ont* ontology references a *locator* and a *universal-property* ontology, the *locator* ontology references a *GPS* ontology and forth. At no point in this structure are external schemas or ontologies referenced other than basic Dublin Core[25]. It forms a closed world that is nicely structured because it is under the control of one group but is isolated. This raises a core issue of semantic web ontology design. Ideally a *person* ontology should be able to reference some general concept such as *mail address* and users of the *person* ontology should be able to chose different modelling approaches for the detailed representation of *mail address* but these representations should all relate to the same abstract concept. In that way when a different mail address representation becomes standardised and generally accepted it can be used within profiles without having to change the *person* ontology itself. This decoupling of separable modelling choices via abstract concept mappings is a key requirement that has arisen from our empirical explorations to-date.

A second existing user profile tool is the vCard standard defined by IETF RFC2426. As well as name and contact information vCard also includes depictions (photos, sounds, logo), organisation, role, credentials (public key), and minimalist demographics (birthday). It thus covers, at some level, most of our requirements except for interests and colleagues. The vCard standard itself is a syntactic standard defining a specific text encoding. Attempts have been made to translate this standard to an RDFS schema [25] but there are many ways such a mapping could be done and no standardisation of it.

As an example of the modelling choices that must be made in doing this translation, consider the issue of locators (email addresses, physical mail addresses and phone numbers). The vCard specification supports multiple contact points that can be classified along several dimensions – context (home, work) and type (e.g. cell-phone vs. fixed phone numbers, international/domestic/parcel mail addresses) [26]. The vCard standard uses *type attributes* to model this. In the RDFS modelling of vCard, the different categories of locator are indicated through subclass relationships – thus a phone number can be given both an `rdf:type vcard:voice` and an `rdf:type vcard:work` to indicate the nature of the phone number and both of these are subclasses of a general `vcard:TELTYPE` class. This works, but is not the only choice – for example, a specific property such as `vcard:category` could be have been used.

Using subclassing is a problem when a category applies in several places – for example, `vcard:work` is a subclass of both `TELTYPE` and `ADDRTYPE` suggesting that an instance of `vcard:work` is in the intersection of these two classes which is not correct. This is a limitation of using RDFS. Using DAML this could have been more modelled using unions, which would capture the original intent better, but it is still awkward if we want to reuse the *home* label as a modifier for some

[25] And even there we encounter a problem in that the DRC Dublin Core references are not in the correct Dublin Core namespace, DRC changed the namespace to prevent the DAML validator complaining that there is no published DAML+OIL file at the namespace location.

[26] This is an important feature for us – for example users may include their home phone numbers in their ePerson profile but only want to share that with close colleagues but may be happy for anyone in the network to see their work phone number.

other concept such as a user interest. There are also situations with vCard where the range of type modifiers is not bounded by the vCard spec, for example UID (which refers to country-specific unique identifiers). In that case the RDFS mapping uses a custom property (`vcard:TYPE`) whose value is simply a literal.

There are several places in the vCard specification where a property can have multiple values – the ROLE property for example. In the vCard RDFS mapping this modelled by using RDF collections (bags, seqs, alts). This has the advantage that ordering and preference information can be given. It has the disadvantage that these properties cannot be given a range constraint because neither RDFS nor DAML can express composite types (e.g. "a bag of *role* instances"). In the case of vCard this is not in fact a problem because the values are simple string literals.

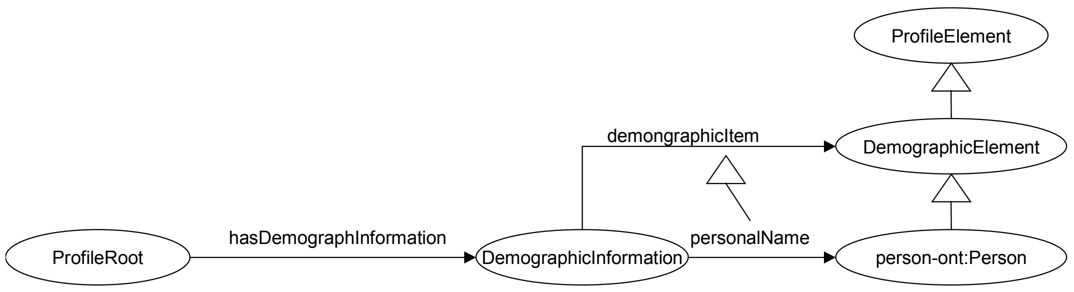
The third important existing profile modelling proposal is Dan Brickley's *friend-of-a-friend (foaf)* schema [20]. This is a much simpler schema than vCard and consequently has less deep coverage of some areas. It does offer name, depiction, interest and some contact information (just email and homepage). It goes beyond person-ont and vCard in having a direct representation of social networks in the form of the `foaf:knows` relation and in the of modelling of university related topics such as current and past research projects and funding agency. Apart from name information, which has some structuring, most elements are modelled as simple properties whose value is a uri (well, an `rdf:Resource`). This gives more structure than the literals typically used in vCard (for example for email address or organization) and is more open than the DRC approach of restricting the range to specific classes of resource. Foaf also goes some way to supporting the abstract concept mapping we mentioned above in that a few of key concept classes (*person, document, organization, project*) are linked to the corresponding concept in Wordnet 1.6 [26].

Compared to our requirements *foaf* has incomplete coverage of topics (omits demographics, credentials, role) and has limited depth of modelling in other areas (no mail address, interests unstructured). In particular it has no general mechanism for separation of home, work or school information apart from the special case of homepages – even there the concepts of, for example, `foaf:homepage` and `foaf:workplaceHomepage` are unrelated; there is no property or class hierarchy relationship between them included in the model.

ePerson profile solution – structured profile

Our first approach to creating an ePerson user profile ontology was to create a structured backbone and then refer to leaf concepts modelled via a mixture of external and locally developed ontologies.

The main profile elements were modelled as substructures rather then subclasses. Thus the root element was an object of class `ProfileRoot`, which then linked to objects such as `DemographicInformation` or `ContactInformation`. These second level nodes in turn linked to specific profile elements. A subclass/subproperty hierarchy was used to group the specific elements. For example, all links from the `DemographicInformation` node were subproperties of `demographicItem` whose range was `DemographicElement`. We then made existing externally defined classes subclasses of these third level nodes, for example the users name was stored as a structured object linked to the `DemographicInformation` node via a `personalName` property (a subproperty of `demographicItem`) whose range referred to the `Person` class in DRC Orlando person ontology.



This structured approach had strengths and weaknesses. A particular strength was that it enabled us to mix references to different ontologies. Because the second and third level nodes were distinct from each other, we could draw models for the third level concepts from different ontologies without creating conflicts. For example, in using DRC person-ont to model personal names we were not saying that the `ProfileRoot` or even a `DemographicInformation` node represented a `person-ont:Person` - that would have constrained the other information that could attach to those nodes. Instead `ProfileRoot` and `DemographicInformation` remained malleable abstract nodes that simply linked together separately modelled pieces.

The structured approach also had some advantages for query formation in that we could retrieve a required subset of the profile just by navigating the structural links without having to directly use subclass or subproperty relationships. Though this advantage would have been reduced had we had better inference support in our query processing.

One disadvantage of the structured approach was its sheer complexity; the mixture of separate nodes structures, property hierarchies and class hierarchies was difficult to follow. This complexity resulted in some errors and some redundancy.

Another concern was that in some sense we were modelling a data structure, which happened to be holding a user profile, rather than directly modelling a person. This meant that the semantic relationship between the information attached to different second level nodes was not captured in the ontology itself but implicit in the processing model. For example, consider two properties - personal email address and unique id (such as a social security number). Each of these is an unambiguous property of a person in that two people with different values for either should be treated as distinct. In the structural modelling approach the personal email address would be a property of a `ContactInformation` node and the unique id a property of the separate `DemographicInformation` node, each property is giving an unambiguous labelling of a different entity. Without composition operators (which are not available in DAML) we cannot express the notion that the unambiguous nature of these relations refers back up to the common `ProfileRoot` node (or actually the implicit abstract *person* concept behind the `ePersonID` to which the `ProfileRoot` is attached).

The structural approach also does not solve the question of how to relate our representation to the other schemas. For example, having chosen DRC person-ont for modelling personal names we have no explicit information on how this concept matches to the related concepts in vCard or foaf data sources.

ePerson profile solution #2 – mapping profile

For the second version of our ePerson profile ontology, we attempted to address these issues by removing the structural separation of profile elements (relying on subclassing to provide the separation). Our profile root element was intended to directly model a "person" in sense of `foaf:Person` or `person-ont:Person` so that unambiguous properties attached to that element uniquely identify the person. We also tried to explicitly map each ePerson profile concept to the related concepts in each of vCard, person-ont and foaf.

This led to a rich profile ontology with more complete and uniform coverage of our needs than any of these existing ontologies but without the structural complexity of our first design.

This synthesis attempt was mostly successful but the greatest problem was found in attempting to map the concepts we were modelling to the related concepts in the other ontologies. Without this mapping all we are doing is creating yet another profile schema.It may be a more complete and comprehensive schema, but it risks being just as standalone and isolated as all the other schemas we have discussed.

Amongst the mapping challenges we encountered were: structural dissimilarity, type annotations, conflicts between literals and resources, and range issues. We explore each of these further below.

structural dissimilarity

Consider a personal name. All the ontologies provide some breakdown of a name into components (surname, given name, title etc), some notion of a display name (often expected to be some concatenation of the components) and some notion of alias or nickname. However, they all differ in the precise breakdown and how it is achieved.

For example, in vCard the structured name is a resource of type `NPROPERTIES` attached to the root person via a property `N`. The name components (`Family`, `Given`, `Prefix`, `Suffix`, `Other`) then attach to the `NPROPERTIES` node. In contrast in foaf all the name components (`givenname`, `surname`, `title`, `firstname`[27]) attach directly to the root `Person` resource. The DRC person-ont takes a similar approach to foaf but with a different breakdown of components (`firstName`, `lastname`, `title`, `middleName`).

There are several structural problems here. First, even though all are just trying to break a name string into a set of concatenatable components they all have at least one option unique to them. For example, a person with a middle name is happy with person-ont, might use `Other` if forced to use vCard and within foaf

[27] We believe this may be a small bug in foaf in that `givenname` and `firstname` are the same concept.

might use a repeated `firstname` property or include their middle name in the `firstname` string. None of these sets can be mapped onto any of the other without loss. We made our structured name a superset of these components but could not directly express any consistency checks, for example that the concatenation of these string values in a given order should be an invariant.

However, even if we restrict ourselves to the common components (e.g. firstname/lastname pairs) we still have a structural problem. The vCard model has an explicit structured name entity that is distinct from the person being named. If we try to assert individual properties such as `vCard:Family` and `foaf:surname` as equivalent, this would then imply that our root entity is both a `foaf:Person` and a `vCard:NPROPERTIES` (since that is the domain of the `vCard:Family` property). If we mapped data from vCard to foaf and back we would end up with a single root node with the `vcard:N` property both starting and terminating on the same node. Whilst this doesn't lead directly to contradictions in this case (RDFS has insufficient representational power to ever lead to contradictions) it is clearly unsatisfying. This situation could possibly be handled with DAML if we had a property composition operator – that would enable us to map two property sequences together without having to map the intermediate classes.

type annotations

There are situations where the same concept (e.g. a phone number) needs to be tagged with some context modifier (e.g. work versus home) where the different modelling choices make the mapping hard. In person-ont there are distinct classes `WorkTel` and `HomeTel`, which then have the actual phone numbers attached to them as literals via distinct properties `workTel` and `homeTel`. In vCard there is no explicit type that is the intersection of the concepts "work related" and "phone number". Anything which is the range of a `vCard:TEL` property is a phone number, but its type may be restricted by also being of type `vCard:work`. In foaf there is little support for such categories with the exception of personal and work related homepages – that case is modelled by having two different properties with no explicit relationship between them.

We followed the vCard like approach and defined a range of type modifiers (`WorkRelated`, `HomeRelated`, `SchoolRelated`, `ProjectRelated`, `HobbyRelated`) that could be applied to any profile element.

This works reasonably and in most cases the mapping to other schemes is hard rather than impossible. Our concept `workRelated` is at an abstract level the same thing as `vCard:work` but we can't say they are equivalent because `vCard:work` is specific to addresses and phone numbers. This can be captured in DAML by saying that the intersection of `PhoneNumber` and `workRelated` is equivalent to the intersection of `vCard:TELTYPE` and `vCard:work` though this will be costly to reason with and is a slight mischaracterization of `vcard:TELTYPE`.

literals verses resources

In designing a schema, there is a choice about how external referenced entities are represented. Schemas like vCard tend to use literal strings to represent entities such as email addresses or types of unique identifier. In contrast, foaf uses URIs (`rdf:Resource`) for such leaf concepts and the DRC ontologies constrain the resources to be in specific classes. In some cases, email addresses for example, an agent with additional information on the syntactic form of the literal could map between the two but it is not possible to directly relate either the classes (literals and resources being distinct) or the properties. So while `vCard:EMAIL` (with range restricted to the class `vCard:internet`) and `foaf:mbox` are trying capture same high level concept we cannot directly relate the two.

This caused problems internally in the ePerson profile ontology. We wanted all properties of the profile to be subproperties of `prof:profileElement` and all property values to be subclasses of `prof:ProfileElement`. For example this would have enabled us to allow `prof:WorkRelated` to be a subclass of `prof:ProfileElement` and so be attachable to any value. However, many properties had non-resource ranges and so need to be modelled as either `range:Literal` or as datatype properties with range such as `xsd:string`. We compromised by having all properties be subproperties of `prof:profileElement` which then precluded us from having the class `prof:ProfileElement` as the range of `profileElement` since neither literals nor concrete datatypes can be subclasses of such a class.

range of literals

A small but annoying problem occurs when a property has a value that is "some sort" of string. RDFS schemas such as foaf typically express this by saying the property has range `rdf:Literal` whereas DAML schemas typically use a datatype property whose range is `xsd:String`. RDF Literals and XSD strings are different concepts and two such properties cannot be equated even though the user intent and the actual character sequence written down in each case will be identical.

Our interim solution to these problems was to introduce a set of properties to indicate a partial relationship exists between two elements in different ontologies but where the precise semantics is left undefined. We called them things like `relatedTo`, `nearEquivalent`¸ `specializationOf` and `generalizationOf`. This is not really a solution, in that the semantics of the relationship is not being captured. However, it is useful as a set of annotations that an ontology-mapping tool could use to create partial mappings – it is capturing information on the sorts of structural and term similarity that is often used for schema mapping.

A better solution that we would like to explore in the future would be to link the classes in each ontology to some grounding set of abstract concepts (such as the wordnet classes). Thus, instead of a direct mapping between two classes we say that each class is a partial attempt to represent the same underlying concept. We want this mapping to be valid even though there might be differences in representation choice. For example, two properties could be both mapped to the same abstract relationship even if different modelling choices are made for capturing the object of the relationship (for example one property might have range `Literal` and the other might have range `Resource`). The details of this is a topic for future research.

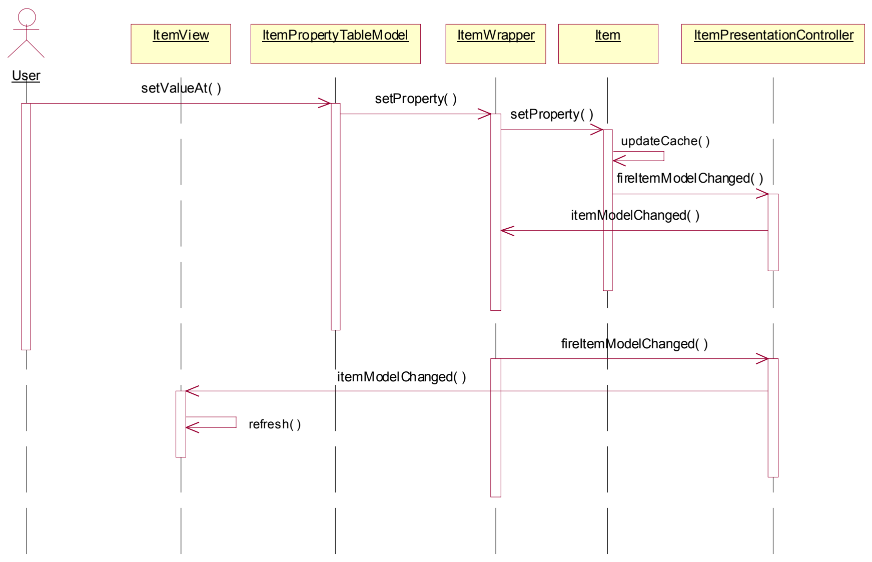
Appendix 5 Application Layer Implementation Notes

9.1.1 UI Java classes

We defined two Java interfaces, `Item` and `Workspace`, which allowed us to implement classes encapsulating the RDF data. Essentially, these classes wrap instances of (DAML) classes. Each of these Java classes was specialised as required (for example we implemented `EPersonItem` to encapsulate data about a peer). We also implemented an `ItemWrapper` that enabled us to provide useful services like property ordering and caching.

In order to provide a flexible framework for viewing data, we provided a set of view abstractions. The main three were `WorkspaceView`, `ItemCollectionView` and `ItemView`. The `DefaultWorkspaceView` class was composed of item collection views and item views, and the `CommunityBrowser` tool was implemented as an extension of this class. The `DefaultItemCollectionView` was a simple list. The view was specialised by allowing collections to be viewed hierarchically (as a tree), although other options (eg graphs) are also possible. The `DefaultItemView` was agnostic with respect to the item's class (a simple list of properties) or could be specific to the class – e.g. bookmark, ePerson, workspace. The way in which item views are chosen is described in the next section.

Event flow proved to be an issue, in particular the task of keeping the different views and the underlying data in synchrony. An example UML sequence diagram (below) shows how item data changes were propagated to all interested parties. The essential point is that data change requests are propagated down to the item, while modelChanged events are propagated up from the item to registered viewers. We implemented a chaining, with `ItemWrapper` listening to `Item`, and `ItemView` listening to `ItemWrapper`. This was achieved using a controller class (the `Item` and `ItemWrapper` in the diagram below broadcast their changes using different controller objects).

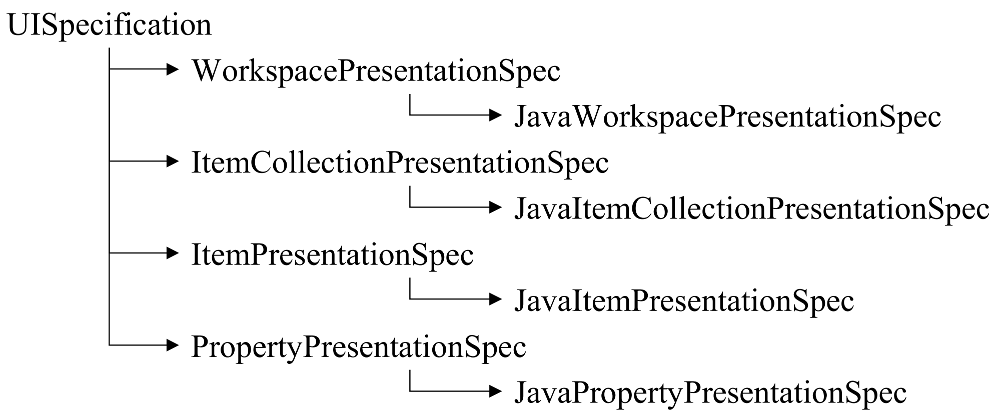


A similar scheme was implemented for `ItemCollectionView`, which listened to collection level changes (such as item selected/added/removed) and `WorkspaceView`, which listened to workspace level changes (such as workspace added/deleted/updated).

One advantage with this scheme was that data changes are reflected wherever viewers are listening to the same item. So, for example an item copied to a new (even remote) workspace will reflect the item changes in progress. Note that the changes are not (at this stage) written out to the underlying RDF store; rather, the `Item` (Java) object caches all changes and writes them out when the user selects 'Save Changes'. A disadvantage is that the viewers have to be listening to the same Java object. Thus, the same item (by URI), if represented by 2 Java objects, will not be updated until the changes are written out to store and the workspace refreshed. This could be addressed by using a 'push' model of updating and propagating changes, but such a mechanism would add to the complexity of the UI.

9.1.2 RDF for configuration

The state of the UI was stored in the RDF model. Thus, the currently open workspaces are displayed in their correct screen positions when the Snippet Manager is restarted. This is achieved by use of a `UIElement` (DAML) class which `presents` a workspace, and has attributes such as `width` and `height`. In order to specify the presentation in RDF, we defined a `UISpecification` DAML class, and specialised this as shown below.


UISpecification
WorkspacePresentationSpec
JavaWorkspacePresentationSpec
ItemCollectionPresentationSpec
JavaItemCollectionPresentationSpec
ItemPresentationSpec
JavaItemPresentationSpec
PropertyPresentationSpec
JavaPropertyPresentationSpec

As can be seen, this mechanism gave us a way to map from presentation specifications (coded in RDF) to Java classes, for example for the bookmark item view:

```
<ItemPresentationSpec rdf:ID="theBookmarkItemView">
    <rdfs:label>Bookmark item view</rdfs:label>
    <rdf:type rdf:resource="#JavaItemPresentationSpec" />
    <rdfs:comment>The item view for bookmarks</rdfs:comment>
    <pres:JavaDefinitionClass>
        com.hp.hpl.eperson.sm.ui.BookmarkItemView
    </pres:JavaDefinitionClass>
    <pres:viewOrder>2</pres:viewOrder>
</ItemPresentationSpec>
```

The appropriate presentation specification can then be passed to a factory, which will instantiate the required class through the Java Reflection API.

For workspaces, the specification is unambiguously defined via the `layout` property. The workspace presentation specification may define one (or more) item collection presentation specifications (these can be tabbed). Item collection presentation specifications may define a `preferred` item view. The item class (or one of its superclasses) will define one or more additional item views (the root class, 'Item' has its `itemView` property set to `theDefaultItemView`, so that view is always available for any item). Again, multiple item views can be tabbed, with the preferred item view (if defined) always coming first.

Property values could be viewed and edited using classes implementing the `PropertyComponent` interface. This allowed a variety of methods for the appropriate rendering of data (eg simple text, date formatting, highlighted as bold) or more complex formatting (eg if the property value is itself an RDF resource, render it as a selected property of that resource). The appropriate component is specified using a `PropertyPresentationSpec`. Unlike item views, property values are only allowed one view (renderer). Therefore some conflict resolution may be required. The pseudocode follows:

```
IF property (or superProperty) has preferredPropertyView defined
    use that
Elif range of property (or one of its superproperties) has propertyView defined
    use that
Elif property value has a class for which propertyView is defined
    use that
Else
    use default component
```

Appendix 6 Transport Layer Implementation Notes

Transport API

The *TransportFactory* allows new *Transport* instances to be created. A *Transport* instance is typically used to send and receive messages on behalf of one entity (e.g. an ePerson or a KB). The credentials of that entity are represented by an *Authority* instance, which can be created from an XML credentials file using the *Credentials* helper class. After creating the *Transport* instance, it should be bound to the *Authority* using the `setDefaultAuthority` method. The Snippet Manager and KBHost applications only operate on behalf of one *Authority*, so only need to create a single *Transport* instance.

The transport API supports the following messaging semantics:

Request-response

This looks like conventional RPC, where a single response is expected. In general, the destination endpoint will be a single node, rather than a broadcast address. This type of interaction can be achieved as follows:

```
sendRequest(Address dst, MessageBody body, boolean sign, int timeout)
```

The application thread is blocked until either a response is received, or the timeout occurs. The response is returned by the call. Any additional responses are dropped.

Request-multiple response

This mode allows multiple responses. There are really two cases where this is useful. The first is when the destination may provide a partial response quickly, and the remainder of the response some time later. The second is when the destination is actually a multicast or broadcast address, and so responses may be received from several different nodes. No attempt is made to keep track of the number of responses; it is down to the application to decide how long to wait and when to give up.

This type of interaction can be achieved as follows:

```
sendRequest(Address dst, MessageBody body, boolean sign, MessageListener listener)
```

The application thread is not blocked and each response results in the specified message listener being called (from a different thread). The application can cancel the message listener when it has received sufficient responses. Note that this mode can be used for non blocking RPC if the listener is cancelled when the first response is received.

Asynchronous messaging

In this mode, no response is expected. This type of interaction can be achieved as follows:

```
sendMessage(Address dst, MessageBody body, boolean sign)
```

Example protocol message

```
<eps:Message xmlns:eps="http://www.hpl.hp.com/eperson">
  <eps:Header>
    <eps:MessageType>
      1
    </eps:MessageType>
    <eps:SrcAddr>
      anonymous
    </eps:SrcAddr>
    <eps:DstAddr>
      urn:x-hp-kb-sha:739f6a3c541a9d18a588f48e9c403447a144c625
    </eps:DstAddr>
    <eps:MessageId>
      a913efa5-2732-51b2-8000-8ef35df5bdbc
    </eps:MessageId>
    <eps:Authority>
      <eps:Identity>
 urn:x-hp-eperson-sha:10f93f6e431ff650b5b2b1b35ca79b2be9b0912e
      </eps:Identity>
      <eps:PublicKey Algorithm="DSA" Format="X.509">
 MIIBtzCCASwGByqGSM44BAEwggEfAoGB
 AP1/U4EddRIpUt9KnC7s5Of2EbdSPO9E
 AMMeP4C2USZpRV1AIlH7WT2NWPq/xfW6
 MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f
 6AR7ECLCT7up1/63xhv4O1fnxqimFQ8E
 +4P208UewwI1VBNaFpEy9nXzrith1yrv
 8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmC
 ouuEC/BYHPUCgYEA9+GghdabPd7LvKtc
 NrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeO
 utRZT+ZxBxCBgLRJFnEj6EwoFhO3zwky
 jMim4TwWeotUfI0o4KOuHiuzpnWRbqN/
 C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/h
 WuWfBpKLZl6Ae1UlZAFMO/7PSSoDgYQA
 AoGARaATcxT/0UETKc4BxxcW9hRc8mOA
 LJXOy7qPpnVb+VvhWoNi7hvOn+BbF6qZ
 FOIj8OPF7mJdV776W8g0oYVX4WmSW8H3
 0RKF9JsIYdiRPKqc3PVbJEPB2thegx8y
 xiGoFH9m/wyMiKWsuIrNNEii+80azJPh
 +rhVXIC7G66NkxE=
      </eps:PublicKey>
      <eps:Role>
        http://w3.hpl.hp.com/eperson/daml/eperson/role#Individual
      </eps:Role>
    </eps:Authority>
    <eps:Signature>
 MCwCFHVJp4XhUDUBGpF+42/9jjGtFC3A
 AhRHqGjySGj4Y44/QrQEO7fsQTq+ZQ==
    </eps:Signature>
  </eps:Header>
  <eps:Body MimeType="text/xml">
    <kba:KBAccessMessage
xmlns:kba="http://w3.hpl.hp.com/eperson/daml/eperson/kbaccess#">
      <kba:operation>
        list
      </kba:operation>
      <kba:arg type="rdf-n-triple">
        <![CDATA[
_:A29f3b5X3aXf00ef5102eX3aXX2dX51e8
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://w3.hpl.hp.com/eperson/daml/eperson/presentation#Desktop> .
_:A29f3b5X3aXf00ef5102eX3aXX2dX51e6
<urn:dummy#_ANY_>
_:A29f3b5X3aXf00ef5102eX3aXX2dX51e5 .
_:A29f3b5X3aXf00ef5102eX3aXX2dX51e8
<http://w3.hpl.hp.com/eperson/daml/eperson/presentation#displays>
_:A29f3b5X3aXf00ef5102eX3aXX2dX51e6 .
        ]]>
      </kba:arg>
    </kba:KBAccessMessage>
  </eps:Body>
</eps:Message>
```

Message header description

`MessageType` - 0 Asynchronous message, 1 Request message, 2 Response message.

`SrcAddr` - Transport-independent source address; if *anonymous*, then consider the node as a client only, i.e. it can only accept responses.

`DstAddr` - Transport-independent destination address; optional `AddrType` attribute can take values *broadcast* or *community*.

`MessageID` - A unique GUID for this message; allows the recipient to drop duplicates.

`Authority` - Indicates on whose behalf (authority) the message is being sent; comprises and Identity, an optional public key and optional list of roles. This is not to be confused with a certificate authority.

`Identity` - the transport-independent address of the authority; for authentication to be possible, this must be derived from the public key as outlined above.

`PublicKey` - the public key of the authority; currently this must be a DSA public key in X.509 format.

`Role` - the role being claimed by the sender of the message; either the special *Individual* role, or one of the roles from a shared role ontology. Multiple roles may be claimed, and it is the responsibility of the recipient to validate that claim is legitimate.

`Signature` - the message signature (see §5.1.5).

Message body description

The format of the message body is really down to the application. Currently we provide special support for the text/xml mime type. In the example above, the message is a KB access message requesting a list operation be formed on the KB.

Address cache implementation

A single address cache is shared by all *Transport* instances. This maintains a mapping from transport-independent address (i.e. name) to connector type and connector specific address. Multiple mappings may exist, as long as they use different connector types.

Entries are added to this cache automatically whenever a message is received. Entries can also be added manually by the application. No background management of this cache is performed, so entries remain until there are deleted or overwritten.

Since multiple mappings are allowed for each endpoint, we should probably order these based on when they were added, and try the most recently added mapping first, when sending a message. We don't currently do this.

MessageID cache

Duplication of messages may happen because the transport supports multiple paths to the destination, over different connectors. When sending a message, there are cases where multiple paths are tried. To detect duplicates, each *Transport* instance maintains a cache of recently seen `messageID` values.

There may be an issue to address here. Currently, response messages are not subject to this check, since a node may legitimately send multiple responses. The `messageID` *of a request is copied to the response, so that the transport can match the response with the original request. Consequently, the current scheme cannot distinguish the case where the responder sends two separate responses from the case where a single response is duplicated in the network. A possible solution is to ensure that the* `messageID` *is set to a unique value in all messages, and adding a separate* `inResponseTo` *field to the response header.*

Threading model

The following threads exist:

- In general, all sending is handled by the application thread that called `send()`. Also, any resolution listeners will also by called by the application thread.
- Each `MessageID` cache (one per transport) has a garbage collection thread.
- Each `DirectConnector` has two server threads, one listening on a TCP port for unicast messages, the other listening on a UDP multicast port listening for broadcast messages. When an incoming message is received, a new thread is created to handle just that message.
- The `JabberConnector` is built on top of the JabberBeans API. Each `JabberConnector` creates a `ConnectionBean`, which create two threads: an `InputStreamHandler` and an `OutputStreamHandler`.

Transport instances and connector instances

An application may create multiple transport instances. Multiple source address mappings may then be added to each transport, causing connector instances to be created. What does the resulting set of objects look like? This actually depends on the source address mapping, and the rules implemented in the connector factory for each type of connector. In general, the connector factory for each type of connector will maintain a map from a part of connector specific address to the created connector, allowing connectors to be reused. If two different transports add the same source address mapping, then the connector will be shared between them. When a connector is shared by multiple transport instances, any incoming messages are passed on to all of the transport instances.

In the Jabber connector, the connector specific address contains the following fields:

`connector` = "Jabber"

`address` = local Jabber address (**user@server**)

`password` = password to user when connecting to local server

`resource` = (optional) used to distinguish several concurrent connections

`register` = (optional) if present, will attempt to create a new account on the server

`name` = (optional) when registering, allows a name to be specified

`email` = (optional) when registering, allows a email address to be specified

The Jabber connector maintains a map from address to created connector.

In the direct connector (TCP/UDP), the connector specific address contains the following fields:

`connector` = "direct"

`address` = local IP address (or name)

`port` = local TCP port to use to listen for incoming messages

The direct connector factory maintains a map from port to created connector.

In the direct connector factory, we should (but don't) check that the address property corresponds with one of the local IP addresses