Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF

David Huynh

MIT Artificial Intelligence Laboratory
200 Technology Square
Cambridge, MA 02139
dfhuynh@ai.mit.edu

David Karger

MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139
karger@theory.lcs.mit.edu

Dennis Quan

MIT Artificial Intelligence Laboratory
200 Technology Square
Cambridge, MA 02139
IBM Internet Technology Division
1 Rogers Street
Cambridge, MA 02142
dquan@media.mit.edu

Abstract

The Resource Definition Framework (RDF) is designed to support agent communication on the Web, but it is also suitable as a framework for modeling and storing personal information. Haystack is a personalized information repository that employs RDF in this manner. This flexible semistructured data model is appealing for several reasons. First, RDF supports ontologies created by the user and tailored to the user's needs. At the same time, system ontologies can be specified and evolved to support a variety of high-level functionalities such as flexible organization schemes, semantic querying, and collaboration. In addition, we show that RDF can be used to engineer a component architecture that gives rise to a semantically rich and uniform user interface. We demonstrate that by aggregating various types of users' data together in a homogeneous representation, we create opportunities for agents to make more informed deductions in automating tasks for users. Finally, we discuss the implementation of an RDF information store and a programming language specifically suited for manipulating RDF.

Introduction

The Resource Definition Framework (RDF) has been developed to provide interoperability between applications that exchange machine-understandable information on the Web [6]. In other words, RDF is well-suited for facilitating Web Services in resource discovery, cataloging, content rating, and privacy policies.

Of course, the expressive power of RDF is more far-reaching than just agent communication. We postulate that RDF can be well exploited for managing users' information. The semistructured nature of RDF lends itself well to the heterogeneous disposition of personal information corpora. In addition, since RDF provides a standard, platform-neutral means for exchanging metadata, it naturally facilitates sophisticated features such as annotation and collaboration. In this paper, we propose and demonstrate a personal information management system that employs RDF as its primary data model.

Motivation

The goal of the Haystack project is to develop a tool that allows users to easily manage their documents, e-mail messages, appointments, tasks, and other information. Haystack is designed to address four specific expectations of the user.

First, the user should be allowed maximum flexibility in how he or she chooses to describe and organize his or her information. The system should allow the user to structure his or her data in the most suitable fashion as perceived by the user. Section 2 elaborates on Haystack's support for user-defined ontologies.

Second, the system should not create artificial distinctions between different types of information that would seem unnatural to the user. This point is related to the previous point in that the system should not partition a corpus simply because different programs are used to manipulate different parts of that corpus. Rather, the system should store all of the user's information in one homogeneous representation and allow the user to impose semantics that partition the data appropriately.

Third, the system should allow the user to easily manipulate and visualize his or her information in ways appropriate to the task at hand. The user interface should be aware of the context in which arbitrary information is being displayed and should present an appropriate amount of detail. We address these issues later in Section 3 where we discuss Haystack's user interface.

Fourth, the user should be able to delegate certain information processing tasks to agents. Regardless of how powerful a user interface we provide, there will still be many repetitive tasks facing users, and we feel that users will benefit from automation. The details of Haystack's agent infrastructure are given in Section 4.

Contribution

By addressing these four needs, we show that Haystack is able to use RDF to extend several profound benefits to users. First, RDF can be readily exploited to add semantics to existing information management frameworks and to serve as a *lingua franca* between different corpora. On top of this, we provide an ontology that supports capabilities including collection-based organization, semantic categorization, and collaboration and trust management. By ontology we are referring to a vocabulary that specifies a set of classes and the properties possessed by objects of these classes. This ontology enables the user interface to present the user's information in a meaningful manner, and it also provides an abstraction on which agents can run.

Next, we show that RDF can be used to describe the means for visualizing heterogeneous data. In addition to the obvious registration metadata found in all component frameworks, RDF can be used to build user interface specification abstractions that can be directly manipulated by the user just as other metadata. This capability opens many doors for user interface engineering including the realization of a truly uniform interface.

Finally, we discuss the use of RDF for modeling imperative computational processes. We present a language called Adenine as a natural means for manipulating metadata and thus writing agents for Haystack. Adenine programs compile into an RDF representation, affording them the same ability to be annotated, distributed, and customized as other documents and information.

History

The information overload problem has become more and more evident in the past decade, driving the need for better information management tools. Several research projects have been initiated to address this issue. The Haystack project [8] [9] was started in 1997 to investigate possible solutions to this very problem. It aims to create a powerful platform for information management. Since its creation, the project has sought a data modeling framework suitable for storing and manipulating a heterogeneous corpus of metadata in parallel with a user's documents. With the introduction of RDF, a good match was found between the versatility and expressiveness of RDF and the primary need of Haystack to manage metadata. The project has recently been reincarnated to make use of RDF as its primary data model.

Related Work

There have been numerous efforts to augment the user's data with metadata. The Placeless Documents project at Xerox PARC [3] developed an architecture for storing documents based on properties specified by the user and by the system. Like Haystack, Placeless Documents supported arbitrary properties on objects and a collection mechanism for aggregating documents. It also specified in its schema access control attributes and shared properties useful for collaboration. The Placeless Documents architecture leveraged existing storage infrastructure (e.g. web servers, file systems, databases, IMAP, etc.) through a driver layer. Similarly, Haystack takes advantage of the same storage infrastructure, using URLs to identify documents.

On the surface, the main difference between Placeless' architecture and Haystack's is the adaptation of RDF as a standard for information storage and exchange. Although Haystack and Placeless share a lot of similarities in the data model layer, Haystack takes a more ambitious approach to the user interface problem. Placeless' Presto user interface focused on facilitating management of data in general using a predetermined set of interfaces. In developing Haystack, we are experimenting with ways to incorporate the customization of user interfaces into the bigger problem of personalized information management by providing a platform upon which user interfaces can be modeled and manipulated with the same facility as other metadata.

There are other systems, many in common use today, that permit arbitrary metadata annotations on files. The Windows NT file system (NTFS) supports file system-level user-definable attributes. WebDAV [2], a distributed HTTP-based content management system, also permits attributes on documents. Lotus Notes and Microsoft Exchange, two common knowledge management server solutions, both support custom attributes on objects within their databases. However, the metadata are not readily interchangeable among different environments. Further, the structure of metadata in these systems is highly constrained and makes the expression of complex relationships between objects difficult. For example, these systems do not have first class support for making assertions about predicates, making it difficult for the user interface and agents to analyze data conforming to a foreign ontology dynamically.

The Semantic Web project at the World Wide Web Consortium (W3C), like Haystack, is using RDF to address these issues of interchangeability [4]. The focus of the Semantic Web effort is to proliferate RDF-formatted metadata throughout the Internet in much the same fashion that HTML has been proliferated by the popularity of web browsers. By building agents that are capable of consuming RDF, data from multiple sources can be combined in ways that are presently impractical. The simplest examples involve resolving scheduling problems between different systems running different calendaring servers but both speaking RDF. A more complex example is one where a potential car buyer can make automated comparisons of different cars showcased on vendors' web sites because the car data is in RDF. Haystack is designed to work within the framework of the Semantic Web. However, the focus is on aggregating data from users' lives as well as from the Semantic Web into a personalized repository.

Describing and Organizing Heterogeneous Information

In this section we examine strategies for managing a heterogeneous corpus. First we examine how users can define objects using their own ontology. Then we discuss one means for aggregating objects—the collection—and how it is used in Haystack to help users organize their information.

Personalized Ontologies

One of Haystack's objectives is to facilitate the use of an ontology for organizing, manipulating and retrieving personal information. Some classes will be defined in the system ontology, such as those used for describing queries and collections of objects. Also, some properties, such as title, language, and description, will be defined by standard ontologies such as the Dublin Core. Other classes and properties can be defined by the user to suit his or her own organization method.

On the surface, the Haystack model may not seem very different from those of current systems. Programs such as Lotus Notes and Microsoft Outlook support classes such as e-mail message, contact, and appointment and provide default properties, including subject, from, and date. However, whereas the focus of these products is in providing an efficient and user-friendly means for maintaining objects with these standardized schemata, Haystack attempts to facilitate the entry of data using the user's own class definitions. This functionality is typically found in relational database products, where users first specify a structured schema that describes the data they are entering before populating the table. In reality however, schema design is usually left for database and system administrators, who circulate their designs toward end users of Notes or Outlook.

This "one size fits all" approach for schema design is far from perfect. End users are a fundamentally diverse populace, and people often have their own ideas of what attributes to store for particular classes of objects. A good example is an address book. Some people only care about storing one or two phone numbers and a mailing address, while sales managers may be concerned with a breakdown of all past contact with a customer as well as important dates in the customer's life, such as his or her birthday. The current approach of making more and more fields built-in to an address book product is problematic. Software adopting this approach is often overloading to the user who just wants a simple address book, yet perhaps not functional enough for the sales manager. Using a personal database such as Microsoft Access or FileMaker Pro only aggravates this problem, since users are forced to rebuild their address books from generic templates and generic data types.

To solve this mismatch problem, we must examine means for describing per-user customization. Technologies such as RDF provide flexible data frameworks upon which customized schema definitions and metadata can be specified. RDF's data model encourages the creation of custom vocabularies for describing the relationships between different objects. Furthermore, RDF's XML syntax makes these personalized vocabulary specifications portable between different systems. This will be important for allowing agents to enhance the user's information, as is discussed later.

The challenge exists in how to bring this ability to customize a personal information store to the end user who has no experience with database administration. To accomplish this, we have devised tools for helping people manipulate unstructured, semi-structured, and structured data, abstracting the details of schema management from end users. These tools are built upon a flexible, semi-structured RDF data store that Haystack uses to manage users' information according to ontologies they choose.

We generalize the problem of editing data based on arbitrary ontologies by providing a generic metadata editor. This editor takes advantage of the RDF Schema [7] data within Haystack's RDF store in order to present the user with a useful interface to their data. Future versions will allow users to assign arbitrary properties (not just those specified by the schema) to objects by simply typing the name of the property and its value. In this way users need not be conscientious about schemata, and incidental properties specific to one object and not to the class can be entered.

A customized view of an object will often provide a better interface to the user than a generic tool, when one is available. To support this we provide a sophisticated platform for describing these customizations in our prototype user interface tool called Ozone, discussed in Section 0.

In addition to editing properties of specific objects, it is often useful to the user to be able to manipulate the relationship between objects. A graph editor allows users to see a collection of objects and add and/or remove relationships between these objects. This idea has been popularized in tools such as Microsoft Visio, where structures such as flow charts, organization charts, and business processes can be modeled in a similar fashion. Further, tools for drawing semantic diagrams by making connections between concepts have become available. The Haystack graph editor provides these functionalities to the user but records this data in RDF, making the semantic diagrams true representations of "live" data.

Classifying Information

While users may be interested in customizing how their contact information is stored in their address books, some abstractions we argue are best defined by the base system. This prevents the user from being too bogged down with

details of semantic modeling, while providing the user with out-of-the-box functionality. Here we discuss one of these key classes, Collection.

A big problem with many document management systems, including paper-based ones, is the inability to conveniently file documents in more than one category. Although hierarchical folders are a useful and efficient means for storing documents, the hierarchical folder system presents challenges to users who attempt to use it to categorize documents. Does a document named "Network Admin Department Budget for 2001" belong in the "Budget" folder, the "2001" folder, or the "Network Admin" folder? Supposing an individual finds justification for placing it in just one of these folders, it is very possible that other users may have different views on classification and expect the document to be in a different folder. It may also be the case that some days a user will be interested in a time-based classification and other days a department-based classification.

Simply supporting files being in more than one folder at once is not sufficient. Commonly used modern operating environments such as Windows and MacOS already provide mechanisms (called "shortcuts" and "aliases" respectively) for placing objects in more than one folder. On UNIX systems users can create hard links to files in more than one directory at once. However, we find relatively little use of these features for simultaneously classifying documents into multiple categories.

We postulate that this is because the user interface does not encourage simultaneous classification. How many programs can be found whose file save feature prompts the user for all the possible directories into which to place a file? Many users place their files into a single directory because they are not willing to expend the effort to classify files. Of the fraction that are willing, there is yet a smaller fraction who would be willing to save their files in one place, then go to the shell to create the hard links into the other directories.

Collections, like folders, are aggregations of objects; an object may be a member of more than one collection, unlike folders, whose interface encourages a strict containment relationship between folders and objects. This flexible system for grouping objects together is an important tool for allowing users to organize objects in any way they choose.

Throughout the Haystack user interface, we provide simple facilities for specifying membership in more than one collection. Of course, we support direct manipulation schemes such as drag and drop between collections. However, as noted earlier, the system must itself facilitate the placement of objects in multiple collections in order to be useful. For example, Haystack generates collections to store the results of queries. Whereas in some systems, such as Microsoft Outlook, first class membership in multiple collections is only supported when the collection is a search result set, this functionality is supported naturally within Haystack. Still, we envision the greatest source of multiple classification will be from agents automatically categorizing documents for the user.

Semantic User Interface

In addition to modeling data, RDF is used as the medium for specifying Haystack's user interface and how Haystack presents the user's data. Haystack's prototype user interface, named Ozone, uses a custom component architecture, and the user interface is constructed dynamically at runtime based on metadata. In this section, we introduce this metadata-based component architecture, show how it helps construct a uniform user interface, and describe the benefits such an interface might bring.

Component Architecture

The Ozone user interface is constructed from a conglomeration of *parts*. An Ozone part is a code-backed component that can contribute to the user interface in some way. Attributes of each part are provided in metadata. In particular, the part's implementation, the types of data it can render, and the functionality it provides are described.

Types of Parts. There are four types of parts: layout parts, informative parts, decorative parts, and view parts. Layout parts are responsible for segmenting the visual display into distinct spaces and for positioning other parts in these spaces. Informative parts present textual and graphical information to the user. Decorative parts provide decorations such as white margins, line dividers, text spacing, list separators, etc. Finally, view parts use layout parts, informative parts, and decorative parts as building blocks in constructing a unified view of a single object to the user.

Figure 1 shows an example of how the various types of parts work together to present a meeting. Part (a) of the figure shows the end result while part (b) shows the internal wiring. The view part responsible for displaying the meeting employs a vertically splitting layout part to partition the display into two rows: the top row embeds an informative part that renders the title of the meeting; the bottom row contains the details of the meeting. The bottom row in turn contains a stacking layout part that stacks the three fields "Time," "Location," and "Attendees" vertically.

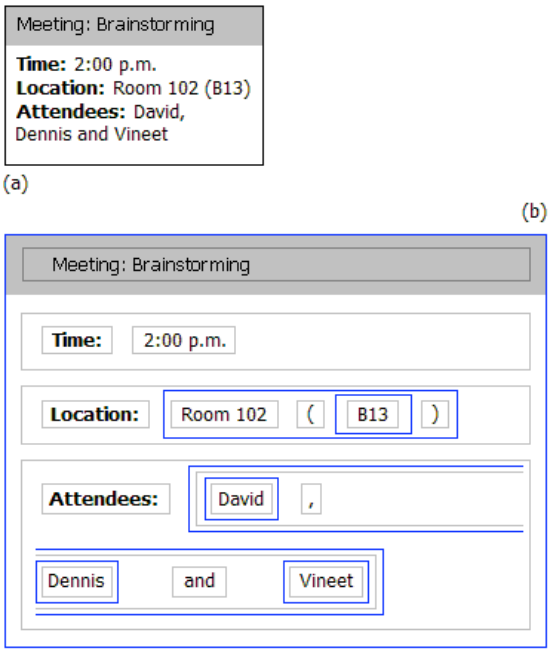

Figure 1. Example of different types of parts working together

The "Location" field consists of a decorative part that renders the label "Location:" and a view part that displays the room where the meeting is held. Note that because the room is a separate entity, the meeting view part does not attempt to present the room itself but rather employs another view part specialized to present the room. (The room is a separate entity because it has been modeled as a resource rather than as a literal property of the meeting.) The room view part includes an informative part to display the room's title "Room 102," two decorative parts to show the parentheses, and yet another view part to display the building where the room is located.

The "Attendees" field consists of a decorative part that renders the label "Attendees:" and a view part that shows the collection of attendees. The collection view part uses a list layout that positions the collection members sequentially, with decorative parts showing comma and "and" separators in between. The collection members are rendered by their own appropriate view parts.

Note that each view part is responsible for displaying exactly one semantic entity. In Figure 1, there are seven distinct semantic entities: the meeting, the room, the building, the attendee collection, and the three individual attendees. If a semantic entity is related to other semantic entities, the view part for that entity may incidentally embed view parts for the other entities. The parent view part determines the appropriate type of each child view part to embed, so that the nested presentation looks pleasing. For instance, a small parent view part embeds only small child view parts.

Part Metadata. Given a semantic entity to present, Ozone queries the RDF store for the part suitable for displaying the entity. Figure 2 shows an example of the metadata that links the entity to the suitable part.

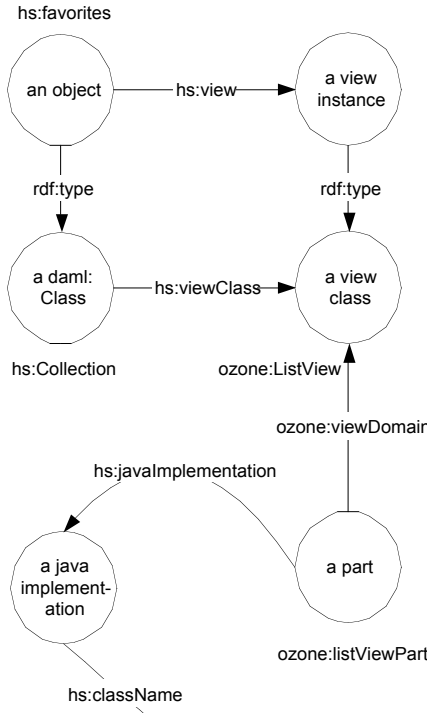

Figure 2. Part metadata example

In order to display the hs:favorites entity, Ozone queries for any view instance associated with the entity through the hs:view predicate.[1] If no view instance is found, Ozone determines the type of the entity (in this case, hs:Collection) and the view class corresponding to that type (ozone:ListView). Ozone then instantiates a unique resource that serves as a view instance for the hs:favorites entity and asserts that the view instance is of that view class type. The view instance will be persisted thereafter and it will serve to store custom settings applied by the user while viewing the corresponding semantic entity. In this example, such settings include the columns and their order in the list view. Each type of view instances persists its settings in its own custom ontology.

Once a view instance exists and its view class is known, Ozone queries for a part whose view domain (ozone:viewDomain) is the class of the given view

[1] The hs: prefix denotes a URI belonging to the Haystack namespace. Note that the idea of views is inherent to the Haystack ontology, whereas the view parts used to display them are inherent to Ozone.

instance. In this example, the part ozone:listViewPart is found. Ozone then finds the implementation of this part, instantiates the corresponding Java class, and initializes it with the semantic entity.

Benefits. The ability to encode the user interface in metadata and then render the interface from that metadata is appealing for a number of reasons. First, it allows the component architecture to construct the user interface dynamically at run-time; any change made to the metadata can be applied immediately. This is not possible with conventional user interface programming models for which user interface layouts are compiled into binary resources or code and loaded at runtime. Any change to such layouts requires recompilation unless the code is specifically parameterized to accept runtime customization. Even in rapid application development tools like Microsoft Visual Basic in which user interfaces can be built by drag and drop operations, there are two distinct edit and runtime modes. The user is only allowed to interact with the application in runtime mode when the user interface is already fixed and unchangeable. Skinnable applications also have a similar limitation. Skins are made by skin designers and then published to users. The users can select which skins to apply to the applications, but they cannot customize the skins themselves. Again, there are two distinct design and use modes that deny the users the power of customizing the user interfaces themselves. Likewise, web pages have two edit and view modes: in view mode, the user cannot make modifications. Our Ozone interface architecture imposes no such modes and allows the user to make changes to the user interface at runtime.

Note that user interface changes performed by the user are high-level: they are to programmers' user interface work as interior design is to carpentry. In other words, customizing the user interface is akin to editing a word processing document or manipulating a spreadsheet. The user is allowed to arrange parts and apply visual themes over them; all the necessary "carpentry work" is handled automatically by Ozone. Since arguably there is no universal interface that suits the needs of every user, this ability to customize one's user interface is desirable. In fact, such personalization features have been explored in simple forms on several portal web sites like http://my.yahoo.com. We would like to bring personalization to our information management platform. In addition to arranging pre-installed parts, the user is offered to select new parts from a part repository and drag them to desired locations in the Ozone interface.

One might argue that similar component architectures and personalization engines have been implemented without the need for RDF. In fact, anything implemented using RDF can be done with custom formats. However, what RDF offers is a unified and extensible framework much needed in the presence of several incompatible custom formats.

The second benefit of user interface metadata is that, like any other type of data, the user interface metadata can be saved, published, and shared. The ability to publish user interface metadata is particularly attractive. Consider a scenario in which a new employee enters a company with a large intranet. This intranet offers countless useful resources to the new employee, but because of its sheer volume, it is intimidating to get used to. Fortunately, other employees have over time collected a selection of Ozone parts that provide access to the most useful features of the intranet. The new employee can simply make use of these parts as a starting point. These parts are brought into the employee's Haystack and tailored based on his or her preferences. These parts can interact with the employee's Haystack and perform automatic customization that makes Haystack much more powerful than a static web page listing a set of useful links to the intranet.

User interfaces, hence, can be evolved by the users for themselves as their needs emerge and coalesce. Users with different needs tailor their interfaces differently. Those with similar needs share their interface layouts. This philosophy relieves the interface designers from the impossible task of making universal interfaces that satisfy both expert and novice users. Instead, we provide tools for the user to tailor his or her own user interface. In addition, by providing an ontology describing user interface interactions, such interactions can be tracked automatically and agents can apply machine learning algorithms to better fit the user interface to the user's implicit needs and preferences.

The third benefit of user interface metadata is that the user interface designer is encouraged to think semantically as he or she encodes the interface in metadata. Since the interface is rendered automatically by the component architecture based on a unified set of semantics, the barrier to creating user interfaces is much lowered. By removing the burden of fiddling with the interface "until it works" or "so it looks nice," we encourage the designer to think at the level of the user's semantics rather than at the level of how the user interface is syntactically constructed. Such abstraction leads to consistency, the most advocated property of user interface [10].

Finally, because the user interface is modeled in RDF just as the user's data is, the tools offered by Ozone for allowing the user to manipulate his or her data are the same as those used when manipulating the interface itself. For example, the mechanism for changing a contact's name is also used for changing the caption of a button. The mechanism for changing the font in an email message body can be used to change the font in which all labels are displayed. In fact, the mechanism for looking up a word in the body of a document is made available for looking up a menu command that one does not understand. This uniform fashion in which all things can be manipulated makes the interface of Ozone consistent and hence, natural and powerful.

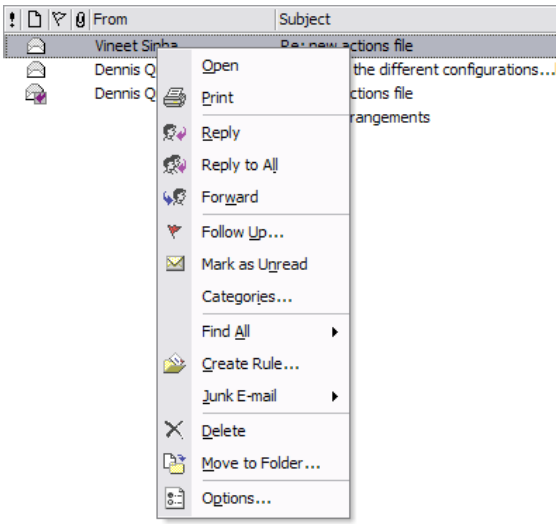

Figure 3. Actions for a contact name in a list view (Microsoft Outlook XP)

Uniform User Interface

Using the power of the component architecture, we explore the concept of a *uniform user interface* [10]. In such an interface, any two user interface elements that look semantically identical to the user afford the same set of actions regardless of context. For instance, a contact name shown in the "From" column for an email message in a list view (Figure 3) should expose the same actions as the same contact name shown in the "From" text field in the email message window (Figure 4). In Microsoft Outlook XP and other existing email clients, those two elements provide almost entirely different sets of actions. The former element is a dead text string painted as part of the whole list view item representing the email message. Right-clicking on it is equivalent to right-clicking anywhere in that list view item. The same context menu for the whole message is always shown regardless of the right-click location. The latter element is a control by itself. It represents a contact object and shows the context menu applicable to that object when right-clicked. To the user, both elements represent the same contact object and should give the same context menu. (Uniformity in this case implies modelessness.)

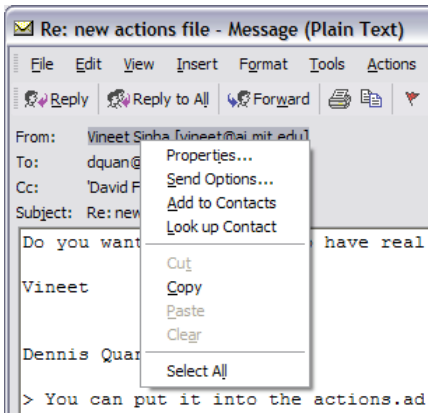

Figure 4. Actions for a contact name in the email compose window (Microsoft Outlook XP)

Context menus have been adopted by most operating systems as the mechanism to query for actions applicable to the indicated object. However, context menus are rarely used consistently throughout any user interface. Some user interface elements afford context menus while others do not. Furthermore, some context menus do not expose all possible actions that the user associates semantically with their corresponding objects. Their inconsistency and incompleteness make context menus less powerful, and hence, less useful and less widely adopted by users than they should be. We aim to fix such inconsistency and incompleteness by providing context menus for all user interface elements and by systematically constructing the menus from metadata.[1] We believe that this uniformity will make the interface much easier to use.

In order to construct context menus from metadata, we first note that every pixel on the screen corresponds to a particular Ozone part that renders that pixel. If that part is a view part, there is a corresponding semantic entity that it is responsible for displaying. That view part can be contained in other outer view parts. All of these view parts together specify a collection of semantic entities that underlie the pixel. We postulate that one of these semantic entities is the thing with which the user wants to interact. To construct a context menu when that pixel is right-clicked, we simply list all actions applicable to those semantic entities.

Like semantic entities, Ozone parts can also afford actions. For instance, the informative part that displays the text "Vineet Sinha" in Figure 3 allows the user to copy its text.

---

[1] Context menus are not widely adopted also because they are not currently discoverable. Once discovered, they are very memorable. We propose labeling the right mouse button "Show Commands" to make context menus more discoverable.

The layout part that shows the messages in a list view format allows the user to reorder the columns.

Figure 5 gives a sample context menu that will be implemented for an upcoming release. The menu is divided into several sections, each listing the commands for a particular semantic entity or Ozone part. The email author entity named "Vineet Sinha" is given the first section. The email message entity is given the next section. Finally the text label used to display the contact's name is given the third section. Commands for other semantic entities and parts can be accessed through the "More…" item at the bottom of the menu. This is only an example of how context menus can be constructed. The exact order of the sections will be optimized by user study and feedback.

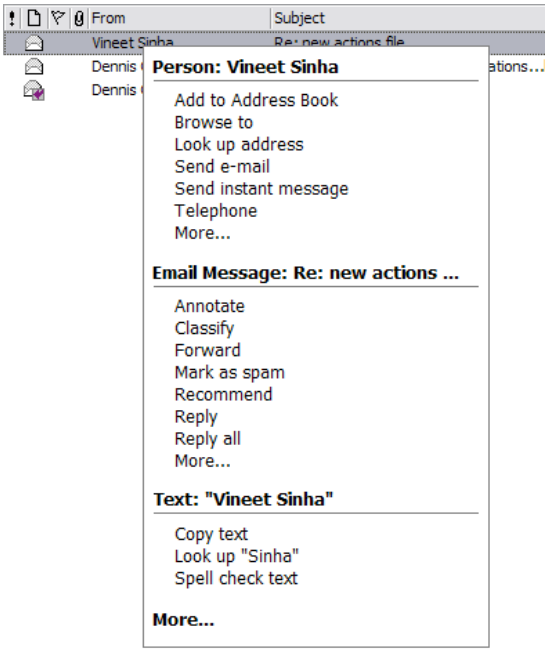

Figure 5. Sample context menu

Of note in Figure 5 are some capabilities not provided in existing email clients. In particular, the "Text" section in the menu offers ways to copy, look up, and spell-check an otherwise dead piece of text. In other email clients, only text inside email bodies can be spell-checked. One can also imagine the usefulness of spell-checking file names [10] and email subjects.

It is arguable that when a user interface element is associated with several semantic entities and parts, its context menu will be overloaded. We believe that with proper modeling of prioritization of commands and in-depth user study, we can heuristically select the most useful commands to list in the menu. Other commands are still accessible through "More…" links. Further, because menu commands are simply members of the collection of all actions applicable to some semantic entity, the user can use the mechanism provided by Ozone for browsing and searching collections to locate particular menu commands. In existing software, menu commands can only be found by visual scans.

Agent Infrastructure

We now turn our attention to agents, which play an important role in not only improving the user experience with regards to keeping information organized, but also in performing tedious tasks or well-defined processes for the user. We also describe some underlying infrastructure needed to make writing and using agents in Haystack efficient and secure.

Agents

In the past, programs that aggregated data from multiple sources, such as mail merge or customer relationship management, had to be capable of speaking numerous protocols with different back-ends to generate their results. With a rich corpus of information such as that present in a user's Haystack, the possibility for automation becomes significant because agents can now be written against a single unified abstraction. Furthermore, agents can be written to help users deal with information overload by extracting key information from e-mail messages and other documents and presenting the user with summaries.

As we alluded to earlier, collections can be maintained automatically by agents. Modern information retrieval algorithms are capable of grouping documents by similarity or other metrics, and previous work has found these automatic classifications to be useful in many situations. Additionally, users can build collections prescriptively by making a query. An agent, armed with a specification of what a user is looking for, can create a collection from the results of a query, and it can watch for new data entering the system that matches the query.

For example, agents can automatically filter a user's e-mail for documents that appear to fit in one or more collections defined by the user, such as "Website Project" or "Letters from Mom". Because membership in collections is not one-to-one, this classification can occur even while the message remains in the user's inbox.

Agents are used in Haystack to automatically retrieve and process information from various sources, such as e-mail, calendars, the World Wide Web, etc. Haystack includes agents that retrieve e-mail from POP3 servers, extract plaintext from HTML pages, generate text summaries, perform text-based classification, download RSS subscriptions on a regular basis, fulfill queries, and interface with the file system and LDAP servers.

The core agents are mostly written in Java, but some are written in Python. We utilize an RDF ontology derived from WSDL [5] for describing the interfaces to agents as

well as for noting which server processes hosts which agents. As a consequence, we are able to support different protocols for communicating between agents, from simply passing in-process Java objects around to using HTTP-based RPC mechanisms such as HTTP POST and SOAP **Error! Reference source not found.**.

Belief

When multiple agents are used to generate the same information, issues arise as to how to deal with conflicts. For instance, if one agent is tasked with determining the due date of a document by using natural language processing and another agent does the same by extracting the first date from a document, which is to be believed when there is a conflict? In instances such as this, it is important that information be tagged with authorship metadata so the user can make an informed choice of which statement to choose.

To accomplish this we discuss a part of the system ontology that is used for describing attributes about actual statements themselves, such as who asserted them and when they were asserted. Under the premise that only three values, namely subject, predicate, and object, are required to describe statements in our model, it is possible to give statements identifiers and to assert an author and creation time to the original statement. In fact, the RDF model prescribes that in order to make statements about statements, the referent statement must be reified into a resource and assigned a URI, and the referring statements can then use the reified resource in the subject or object field.

This use of reification brings up a subtle issue concerning RDF. In a document containing RDF, it is assumed that all statements are asserted to be true by the author. In order to make a statement about another statement that the author does not necessarily believe is true, the target statement must exist only in reified form. In essence, the author is binding a name to a specific statement with a certain subject, predicate, and object, but is not asserting the statement to be true, only instead asserting other properties about that statement using the name.

Keeping track of various levels of trustworthiness is important in a system that contains statements made by numerous independent agents, as well as information from users' colleagues, friends, family, solicitors, and clients. In order to maintain metadata on the statements themselves in an information store, one solution is to have the information store become a "neutral party", recording who said what and when those things were said, but not asserting their truth. This is accomplished by having all statements made by parties other than the information store reified. (An alternative is to have one entity—perhaps the user--be at the same trust level as the data store. However, this results in statements made by the user being handled in one fashion and those made by others (which have been reified) handled in a different fashion. For simplicity of implementation, we keep the data store neutral.)

Once we have a system for recording statement metadata, we can examine issues of retraction, denial, and expiration of assertions, *i.e.*, statements asserted by specific parties. Consider an example where an agent is responsible for generating the title property for web pages. Some web pages, such as those whose contents are updated daily, have titles that change constantly. Often users want to be able to locate pages based on whatever it is they remember about the page. One approach for handling constant mutations in the information store is to allow agents to delete a previous assertion and to replace it with an up-to-date version. However, it would be powerful to allow users to make queries of the form "Show me all web pages that had the title *Tips for Maintaining Your Car* at some point in time." By allowing agents to retract their assertions, queries can still be made to retrieve past or obsolete information because this information is not deleted. Additionally, this system permits users to override an assertion made by an agent by denying the assertion, yet retains the denied assertion for future reference.

In a system such as this where multiple parties and agents provide information, we are often concerned with impersonation and forgery. To solve these problems, we propose supporting digitally signed RDF. The digital signature permits the information store to determine and verify the author of statements with certainty. In an ideal system, users and agents sign all RDF they produce with assigned digital signatures. However, the W3C is still working on the details of supporting signed RDF at the statement level, and the implementation of a digital signature system is beyond the scope of this project. For our current prototype, identifier strings are used in place of true signatures.

Adenine

In a system such as Haystack, a sizeable amount of code is devoted to creation and manipulation of RDF-encoded metadata. We observed early on that the development of a language that facilitated the types of operations we frequently perform with RDF would greatly increase our productivity. As a result, we have created Adenine. An example snippet of Adenine code is given in Figure 6.

The motivation for creating this language is twofold. The first key feature is making the language's syntax support the data model. Introducing the RDF data model into a standard object-oriented language is fairly straightforward; after all, object-oriented languages were designed specifically to be extensible in this fashion. Normally, one creates a class library to support the required objects. However, more advanced manipulation paradigms specific to an object model begin to tax the syntax of the language. In languages such as C++, C#, and Python, operator

overloading allows programmers to reuse built-in operators for manipulating objects, but one is restricted to the existing syntax of the language; one cannot easily construct new syntactic structures. In Java, operator overloading is not supported, and this results in verbose APIs being created for any object oriented system. Arguably, this verbosity can be said to improve the readability of code.

On the other hand, lack of syntactic support for a specific object model can be a hindrance to rapid development. Programs can end up being three times as long as necessary because of the verbose syntactic structures used. This is the reason behind the popularity of domain-specific programming languages, such as those used in Matlab, Macromedia Director, etc. Adenine is such a language. It includes native support for RDF data types and makes it easy to interact with RDF containers and services.

Figure 6. Sample Adenine code

The other key feature of Adenine is its ability to be compiled into RDF. The benefits of this capability can be classified as portability and extensibility. Since 1996, p-code virtual machine execution models have resurged as a result of Java's popularity. Their key benefit has been portability, enabling interpretation of software written for these platforms on vastly different computing environments. In essence, p-code is a set of instructions written to a portable, predetermined, and byte-encoded ontology.

```
# Prefixes for simplifying input of URIs
@prefix : <urn:test-namespace:>

:ImportantMethod rdf:type rdfs:Class

method :expandDerivedClasses ; \
rdf:type :ImportantMethod ; \
rdfs:comment \
"x rdf:type y, y rdfs:subClassOf z => x rdf:type z"
   # Perform query
   # First parameter is the query specification
   # Second is a list of the variables to return,
         # in order
   = data (query {
      ?x rdf:type ?y
      ?y rdfs:subClassOf ?z
   } (List ?x ?z))

   # Assert base class types
   for x in data
      # Here, x[0] refers to ?x
```

Adenine takes the p-code concept one step further by making the ontology explicit and extensible and by replacing byte codes with RDF. Instead of dealing with the syntactic issue of introducing byte codes for new instructions and semantics, Adenine takes advantage of RDF's ability to extend the directed "object code" graph with new predicate types. One recent example of a system that uses metadata-extensible languages is Microsoft's Common Language Runtime (CLR). In a language such as C#, developer-defined attributes can be placed on methods, classes, and fields to declare metadata ranging from thread safety to serializability. Compare this to Java, where serializability was introduced only through the creation of a new keyword called `transient`. The keyword approach requires knowledge of these extensions by the compiler; the attributes approach delegates this knowledge to the runtime and makes the language truly extensible. In Adenine, RDF assertions can be applied to any statement.

These two features make Adenine very similar to Lisp, in that both support open-ended data models and both blur the distinction between data and code. However, there are some significant differences. The most superficial difference is that Adenine's syntax and semantics are especially well-suited to manipulating RDF data. Adenine is mostly statically scoped, but has dynamic variables that address the current RDF containers from which existing statements are queried and to which new statements are written. Adenine's runtime model is also better adapted to being run off of an RDF container. Unlike most modern languages, Adenine supports two types of program state: in-memory, as is with most programming languages, and RDF container-based. Adenine in effect supports two kinds of closures, one being an in-memory closure as is in Lisp, and the other being persistent in an RDF container. This affords the developer more explicit control over the persistence model for Adenine programs and makes it possible for agents written in Adenine to be distributed.

The syntax of Adenine resembles a combination of Python and Lisp, whereas the data types resemble Notation3 [11]. As in Python, tabs denote lexical block structure. Backslashes indicate a continuation of the current line onto the next line. Curly braces ({}) surround sets of RDF statements, and identifiers can use namespace prefixes (e.g. rdf:type) as shorthand for entering full URIs, which are encoded within angle brackets (<>). Literals are enclosed within double quotes.

Adenine is an imperative language, and as such contains standard constructs such as functions, for loops, arrays, and objects. Function calls resemble Lisp syntax in that they are enclosed in parentheses and do not use commas to separate parameters. Arrays are indexed with square brackets as they are in Python or Java. Also, because the Adenine interpreter is written in Java, Adenine code can call methods and access fields of Java objects using the dot operator, as is done in Java or Python. The execution model is quite similar to that of Java and Python in that an in-memory environment is used to store variables; in particular, execution state is *not* represented in RDF. Values in Adenine are represented as Java objects in the underlying system.

Adenine methods are functions that are named by URI and are compiled into RDF. To execute these functions, the Adenine interpreter is passed the URI of the method to be run and the parameters to pass to it. The interpreter then constructs an initial in-memory environment binding standard names to built-in functions and executes the code one instruction at a time. Because methods are simply

resources of type adenine:Method, one can also specify other metadata for methods. In the example given, an rdfs:comment is declared and the method is given an additional type, and these assertions will be entered directly into the RDF container that receives the compiled Adenine code.

The top level of an Adenine file is used for data and method declarations and cannot contain executable code. This is because Adenine is in essence an alternate syntax for RDF. Within method declarations, however, is code that is compiled into RDF; hence, methods are like syntactic sugar for the equivalent Adenine RDF "bytecode".

Development on Adenine is ongoing, and Adenine is being used as a platform for testing new ideas in writing RDF-manipulating agents.

Data Storage

RDF Store

Throughout this paper we have emphasized the notion of storing and describing all metadata in RDF. It is the job of the RDF store to manage this metadata. We provide two implementations of the RDF store in Haystack. The first is one built on top of a conventional relational database utilizing a JDBC interface. We have adopted HSQL, an in-process JDBC-compatible database written in Java. However, early experiments showed that for the small but frequent queries we were performing to render Ozone user interfaces, the RDF store was overloaded by the fixed marshalling and query parsing costs. Switching to a larger scale commercial database appears to result in worse performance because of the socket connection layer that is added in the process.

To solve these problems we developed an in-process RDF database written in C++ (we use JNI to connect it to the rest of our Java code base). By making it specifically suited to RDF, we were able to optimize the most heavily used features of the RDF store while eliminating a lot of the marshalling and parsing costs. However, we acknowledge this to be a temporary solution, and in the long term we would prefer to find a database that is well-suited to the types of small queries that Haystack performs.

Storing Unstructured Content

It is important for us to address how Haystack interacts with unstructured data in the existing world. Today, URLs are used to represent files, documents, images, web pages, newsgroup messages, and other content accessible on a file system or over the World Wide Web. The infrastructure for supporting distributed storage has been highly developed over the past decades. With the advent of technologies such as XML Namespaces and RDF, a larger class of identifiers called URIs subsumed URLs. Initially, RDF provided a means for annotating web content. Web pages, identified by URL, could be referred to in RDF statements in the subject field, and this connected the metadata given in RDF to the content retrievable by the URL. This is a powerful notion because it makes use of the existing storage infrastructure.

However, with more and more content being described in RDF, the question naturally arises: why not store content in RDF? While this is certainly possible by our initial assumption that RDF can describe anything, we argue this is not the best solution for a couple of reasons. First, storing content in RDF would be incompatible with existing infrastructure. Second, leveraging existing infrastructure is more efficient; in particular, using file I/O and web protocols to retrieve files is more efficient than using XML encoding.

Hence, we do not require that existing unstructured content be stored as RDF. On the contrary, we believe it makes sense to store some of the user's unstructured data using existing technology. In our prototype, we provide storage providers based on HTTP 1.1 and standard file I/O. This means that storing the content of a resource in Haystack can be performed with HTTP PUT, and retrieving the content of a resource can be performed with HTTP GET, analogously to how other resources' contents (e.g., web pages) are retrieved. Our ontology uses the Content class and its derivatives, HTTPContent, FilesystemContent, and LiteralContent to abstract the storage of unstructured information.

Putting It Together

At this point, we have described ontologies for personal information management and user interfaces, as well as an agent infrastructure and a data storage layer. In order to gain a fuller understanding of how these components work together, we illustrate an example interaction between the user and Haystack

Figure 7 shows the user's home page, which is displayed when Ozone is first started. Like a portal, the Ozone home page brings together in one screen information important to the user. This information is maintained by agents working in the background. The actual presentation of this information is decoupled from the agents and is the responsibility of Ozone view parts. For instance, the home page displays the user's incoming documents collection, which is managed by the Incoming Agent. When messages arrive, the Incoming Agent may decide to enter them into the incoming documents collection. Similarly, when read messages have been in the incoming documents collection for some period of time, the Incoming Agent may decide to remove them. These mutations to the incoming documents collection are automatically detected by the collection view part sitting on the home page; the view part updates the display accordingly. One can envision the Incoming Agent taking on more intelligent behaviors in the future, such as moving a message deduced to be important but yet unread to the top of the collection

As mentioned earlier, strings of text on the screen corresponding to appointments, news articles, or e-mail messages are not merely dead pixels. Instead, users can manipulate them with context menus and drag and drop them between different areas of the screen. For example, one can imagine dragging an e-mail from the incoming documents view to the calendar in order to set up an appointment. Because the underlying semantic object is connected to the visual representation, the calendar view part can intelligently determine the correct response to the drop operation.

By removing the burden of user interface rendering from the agents, the software designers are encouraged to enrich the agents with more capabilities. One can imagine prolific collaboration between different agents in the Haystack system. For instance, upon retrieving the weather forecast for today, the Weather Agent can notify the Calendar Agent of the grave possibility of a snow storm approaching; the Calendar Agent in turn can attempt to reschedule the user's appointments appropriately. In other systems, especially portals, the focus of a weather agent would be on rendering the weather as HTML, not interacting with other agents to maximize end user benefit.

The news applet displays news downloaded from Resource Site Summary (RSS) feeds of interest to the user. The RSS Agent downloads news on a periodic basis and incorporates the RSS files (which are RDF) into the user's corpus. To take advantage of the collection view part for displaying news, another agent translates the news from the RSS ontology into the Haystack collection ontology. In the future it will be possible to have another agent filter the RSS feeds for the particular articles thought to be most

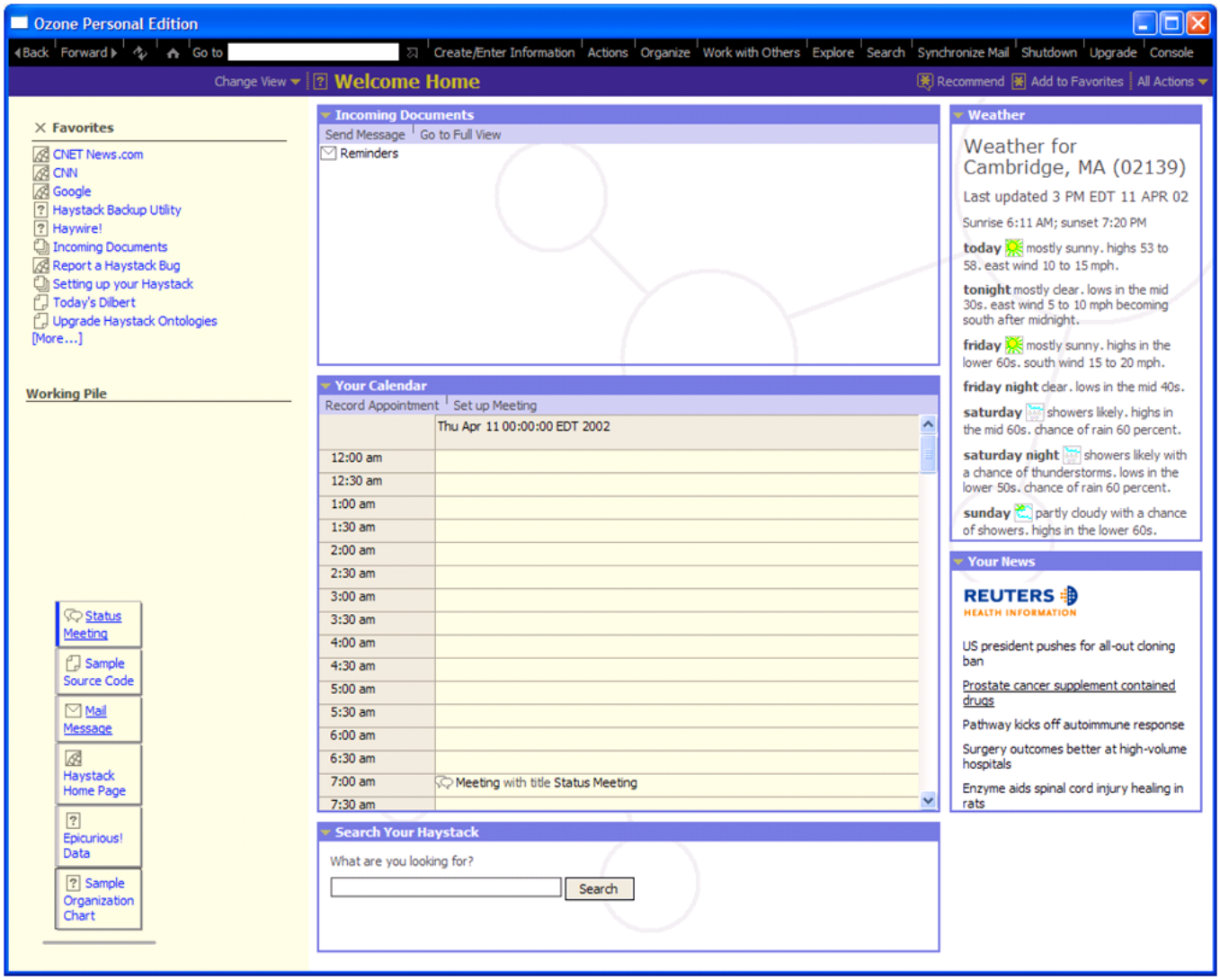

Figure 7. Ozone screenshot

interesting to the user.

Furthermore, the layout of the entire home page and all of its customizations is described in metadata. As with other objects, this layout can be annotated, categorized, and sent to others.

Future Work

Haystack provides a great platform for organizing and manipulating users' information. In this section we touch upon two topics we are currently investigating that build new abstractions on top of the data model discussed above.

Collaboration

Enabling users to work together, exchange information, and communicate has become an absolutely essential feature of modern information management tools. The focus of current off-the-shelf products has been on e-mail and newsgroup-style discussions. However, the addition of rich metadata manipulation facilities creates many possibilities for Haystack in fostering collaboration.

First, Haystack encourages users to have individualized ontologies, so converting between these ontologies when exchanging data will need to be examined. Agents can be instructed in the relationships between different ontologies and can perform conversion automatically. As an alternative one can imagine an ontological search engine that is consulted whenever a user enters data. This way users end up using the same ontologies to describe similarly-structured data.

Second, security issues arise when sharing data. Support for belief networks will need to be expanded to allow users to distinguish their own information from information obtained from others. Access control and privacy will need to be examined to allow users to feel comfortable about storing information in Haystack.

Finally, metadata describing individual users' preferences towards certain topics and documents can be used and exchanged to enable collaborative filtering. Sites such as epinions.com promote user feedback and subjective analysis of merchandise, publications, and web sites. Instead of going to a separate site, users' Haystacks can aggregate this data and, by utilizing the belief network, present users with suggestions.

Organization Schemes

We have started to investigate the many ways in which people organize their personal information in physical form, such as bookcases and piles. We believe that each method of organization has different advantages and disadvantages in various situations. In light of this, we propose to support several virtual organization schemes simultaneously, such that the user can choose the appropriate organization scheme to use in each situation. Different schemes act like different lenses on the same corpus of information. We will provide agents that help the user create and maintain these organization schemes.

References

[1] Box, D., Ehnebuske, D., Kavivaya, G., et al. *SOAP: Simple Object Access Protocol*. http://msdn.microsoft.com/library/en-us/dnsoapsp/html/soapspec.asp.

[2] Goland, Y., Whitehead, E., Faizi, A., Carter, S., and Jensen, D. *HTTP Extensions for Distributed Authoring – WEBDAV*. http://asg.web.cmu.edu/rfc/rfc2518.html.

[3] Dourish, P., Edwards, W.K., et al. "Extending Document Management Systems with User-Specific Active Properties." *ACM Transactions on Information Systems*, vol. 18, no. 2, April 2000, pages 140–170.

[4] Berners-Lee, T., Hendler, J., and Lassila, O. "The Semantic Web." *Scientific American*, May 2001.

[5] Christensen, E., Cubera, F., Meredith, G., and Weerawarana, S. *Web Services Description Language (WSDL) 1.1*. http://www.w3.org/TR/wsdl.

[6] *Resource Description Framework (RDF) Model and Syntax Specification*. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[7] *Resource Description Framework (RDF) Schema Specification*. http://www.w3.org/TR/1998/WD-rdf-schema/.

[8] Adar, E., Karger, D.R., and Stein, L. "Haystack: Per-User Information Environments" in *1999 Conference on Information and Knowledge Management.*

[9] Karger, D and Stein, L. "Haystack: Per-User Information Environments", February 21, 1997.

[10] Raskin, J. "The Humane Interface." Addison-Wesley, 2000.

[11] Berners-Lee, T. *Primer: Getting into RDF & Semantic Web using N3*. http://www.w3.org/2000/10/swap/Primer.html.