Belief Layer For Haystack

by

Marina Zhurakhinskaya

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 28, 2002

Copyright 2002 Marina Zhurakhinskaya. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author……………………………………………………………………
Department of Electrical Engineering and Computer Science
May 28, 2002

Certified by………………………………………………………………
David R. Karger
Associate Professor
Thesis Supervisor

Accepted by……………………………………………………………...
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Belief Layer For Haystack

by

Marina Zhurakhinskaya

Submitted to the

Department of Electrical Engineering and Computer Science

May 28, 2002

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

We have designed and implemented a service for determining the truthfulness of the statements maintained by the Haystack system. These statements can be asserted or denied by various sources interacting with the system. The belief service bases its truthfulness decisions on the sources' trust rankings and on the restrictions imposed by other information about the statements. To achieve this functionality, we have designed data storage structures for keeping and easily retrieving all the relevant information, as well as the algorithms for executing the logistics of the operations on statements. We made available a command line interface for using the belief service features and outlined possible augmentations to the service.

Thesis Supervisor: David R. Karger

Title: Associate Professor

Acknowledgements

First and foremost, I would like to thank David Karger for his guidance, support and for taking time for our discussions of the project. I would also like to thank Dennis Quan for suggesting the idea for this project and for helping me integrate it with the rest of the system. Thanks to other Haystack team members who were helpful at various times – David Huynh, Vineet Sinha, Mark Rosen, Nicholas Matsakis, Kai Shih, Damon Mosk-Aoyama, Ian Lai, Svetlana Shnitser, and Ilya Lisansky – it has been my pleasure working with you and having your advice. The most heartful thanks is to Alex Rakhlin for being so incredible all this time! I am very grateful to all my friends for being there, you all are wonderful! My mother and my father deserve a very special recognition for their love, care, and advice!

Contents

1 Introduction 11

1.1 Overview of Haystack……………………………….. ………………………….11

1.2 RDF Abstraction………………………………...…………………………..14

1.3 Motivation and Goals of the Belief Service………...……………………………15

1.4 Thesis Outline……………………………………………………………….16

2 Specifications for the Belief Service 17

3 Stores Maintained by the Belief Service 21

4 Managing Authors and Their Trust Priorities 27

4.1 Identities and Authors……………………………………………………….27

4.2 Defining Trust Priorities…………………………………………………….27

5 Operations on Statements 29

5.1 Statement Addition………………………………………………………….30

5.2 Statement Assertion…………………………………………………………30

5.3 Statement Denial…………………………………………………………….31

5.4 Statement Retraction………………………………………………………...32

5.5 Statement Deletion…………………………………………………………..33

6 Singlevalueness 35

7 User Interface 39

7.1 Adenine……………………………………………………………………...39

7.2 Statement Denial and Retraction……………………………………………39

7.3 Setting up and Updating the Trust Priorities………………………………...40

7.4 Singlevalueness……………………………………………………………...40

7.5 Sample Interaction in an Adenine Console………………………………….40

8 Future work 43

8.1 GUI Augmentations…...43

8.2 More Property Restrictions…44

8.3 Dates and Their Use…44

9 Conclusion 47

A Tying in Belief Service Into Haystack Platform 49

A.1 Haystack RDF Representation…..49

A.2 Bootstrap File…50

List of Figures

1 Ozone Screenshot……………………………………….……………………………14
2 Expressing Opinion About a Statement……………….………………………….....18
3 Alternatives For Reifying a Statement…………………..……………………………22
4 Alternatives For Recording Authorship Information…...……………………………25
5 Statement Assertion……………………………………………………………..31
6 Statement Denial……………………………………………………………...32
7 Statement Retraction……………………………………………………………33
8 Statement Deletion…………………………………………………………….34
9 Sample of RDF Statements Utilizing Singlevalueness Feature………………………37
10 Sample Interaction in an Adenine Console…………………………………….....42
A1 IRDFStore Interface Code……………………………………………………...50

1 Introduction

Haystack is a personalized information retrieval system that allows users to store, maintain, and query for information. The central part of the system is the semistructured repository of statements manipulated by agents and users. The belief layer was built on top of this repository to establish truthfulness values of the statements. Besides the standard operations of statement addition and deletion, it provides sources (agents and users) with an ability to express opinions about statements, such as to assert or to deny them. The belief service is guided by these opinions, as well as by the trust rankings of their sources, in determining which statements to believe. It also makes it possible to specify the uniqueness of the correct value of some property of an object, and decides which value is truthful in case there are contradicting statements. Thus, the belief service enhances Haystack with more information maintenance abilities and allows the end-user to work exclusively with the set of the believed information.

In this section we first introduce the Haystack system and describe some of its current functionalities. We describe the Resource Definition Framework (RDF) that Haystack utilizes to organize the semistructured data it maintains. We next explain how a belief service can be used to augment this general data storage framework. The desired expansions take into account the source of the information, as well as the restrictions imposed by the data context, to produce a believed subset of the information kept in the system.

1.1 Overview of Haystack

The amount of digital information a usual computer user accumulates and processes nowadays is tremendous. This 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 was started in 1997 to investigate possible solutions to this very problem [1]. 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 as well as various user documents. Haystack has

recently been reincarnated to take advantage of the expressive Resource Definition Framework (RDF) as its primary data model.

Currently, Haystack allows users to easily manage their documents, e-mail messages, appointments, tasks, and other information. The users are able to structure data in the fashion that they consider most suitable. For example, Haystack provides flexibility by letting the user specify various attributes of the documents. The structure of metadata, which is data about data, is not constrained. In this way the user need not be conscientious about schemata and can enter incidental properties specific to some particular document, and not the whole class of objects. Moreover, the user is able to model the customary properties that a certain class of objects could have. For example, it is hard to foresee all the possible types of information that diverse users would want to store in their address book entries. These could include home, work, and cellular phone numbers, e-mail addresses, homepages, home addresses, and birthdays. Making more fields built-in to an address book product is problematic, because it would overload the user who just wants a simple address book, but it would fail to be functional enough for the person who is able to come up with more useful fields. It is best to create a system where a user is able to communicate his own ideas on what attributes to store for particular classes of objects, and this is the approach that Haystack takes.

A big problem with many document management systems, including paper-based ones, is the inability to conveniently file documents in more than one category. To address that, Haystack supports collections of objects, where an object may be a member of more than one collection at a time. Unlike the shortcut and alias features of the modern operating environments (Windows and MacOS respectively), specification of multiple memberships is actively promoted throughout the Haystack user interface and utilized by the automatic categorization agents [2].

Many operations in Haystack are performed by agents that take on various information processing tasks. These tasks could either be well-defined and have reliable results, or be heuristic and have varying degrees of sensible and useful results. An example of a well-

defined task for an agent is retrieving a weather forecast for today from a page on the World Wide Web, while an example of a heuristic task is a text-based classification of a user's e-mail into different collections. At the moment, agents are used in Haystack to automatically retrieve and process information from various sources, such as e-mail, calendars, and the World Wide Web. 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 communicate with the file system and LDAP servers. Some agents are scheduled to run periodically, and some perform their functions only when they are requested to do so or notified of a relevant change in the system. See [2] for an overview of the recent agent architecture in Haystack.

The Haystack user interface, called Ozone, is designed to allow the user to easily manipulate and visualize his or her information. This information is maintained by agents working in the background. Figure 1 shows the user's homepage, which is displayed when Ozone is first started. The homepage has such areas as the user's incoming documents collection, favorites, working pile, calendar, personalized weather report, and news selection based on the user's interests.

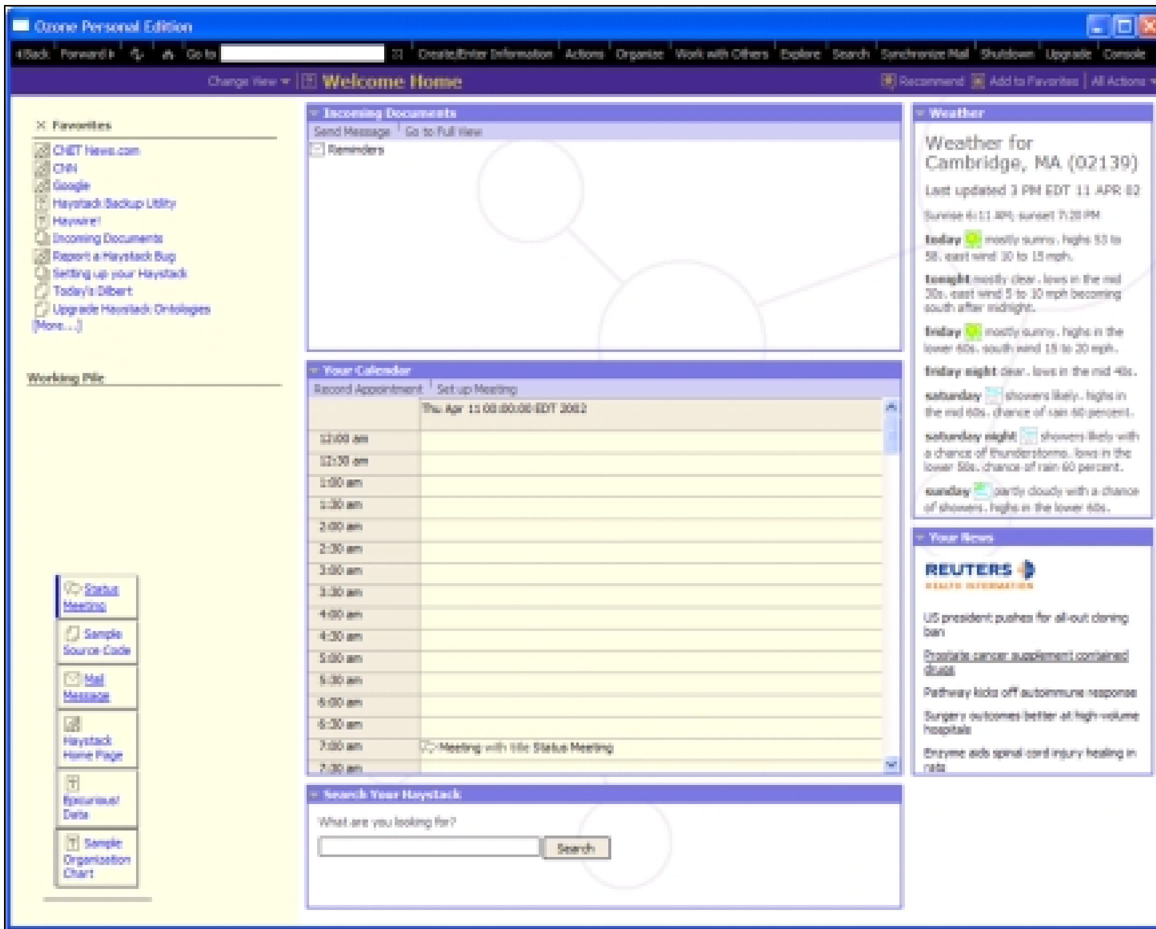


Figure 1: Ozone Screenshot

1.2 RDF Abstraction

To support Haystack metadata and user-document storage, we take advantage of the Resource Definition Framework (RDF), a standard developed by the World Wide Web Consortium (W3C) for storing metadata in a uniform fashion [3]. It was originally created to support agent communication on the Web. RDF describes a directed graph system that contains a set of statements consisting of subjects (nodes), predicates (arrows), and objects (nodes targeted by the arrows). As a simple example, consider a statement "http://web.mit.edu/marinaz/www has creator Marina". The URL is a subject of this statement, creator is a predicate describing a relationship between the subject and the object, and "Marina" is a literal that is an object of this statement. RDF is fully general and can describe all possible kinds of information. It is very suitable for describing the semistructured data maintained by Haystack. In addition, because RDF

provides a standard, platform-neutral way for exchanging metadata, it assists in supporting such inter-platform features as annotation and collaboration.

At the highest level, the RDF storage available in Haystack acts as a repository of statements made by various sources. Tracking who said what is important in a system that contains assertions made by many sources, including users' colleagues, friends, family, solicitors, and clients, as well as assertions made by agents. The next section describes how this authorship information can be used.

1.3 Motivation and Goals of the Belief Service

Imagine there is an agent in the system that is tasked with determining the due date of a document by using natural language processing. Suppose this agent has incorrectly guessed a due date of a document that does not have a due date at all. A user would want to delete this statement about a due date from the system. However, merely deleting it from the RDF store would not be sufficient. This deletion would not prevent the agent, which could be scheduled to perform its task periodically, from adding the incorrect statement to the store again. A user needs to have a way to add a denial of this statement to the system, so that this denial always overwrites the assertion made by the agent. Alternatively, an agent could have incorrectly guessed a due date of a document that has a due date (for example, of a problem set). It would be convenient if a user could come in and add a different statement with a correct due date for that document, and the system would automatically understand that the user's statement overwrites the agent's statement.

The belief layer was built to provide these types of desired functionality. If the RDF store is used alone, it considers truthful all statements that it contains. The belief layer, the objective of this thesis project, does not make this truthfulness assumption. Instead, it allows sources to express their opinion about a statement, this opinion could be either assertion or denial of a statement. By asserting a statement a source declares that it believes this statement to be true, and by denying a statement a source declares that it believes this statement to be false. If there are multiple opinions about some statement,

the belief service chooses the most trusted one. The belief layer bases its decisions on the trust priorities list of different sources that it maintains. It also uses its information about trust priorities to resolve contradictions, such as the ones that arise when there are multiple different assertions about a certain property of a resource, and it knows that there could only be one correct value. Thus, the belief layer acts as a lens over the RDF store, providing the user with an enhanced range of querying and information maintenance abilities. Providing the user with a subset containing only believed and trusted information is certainly essential for a smart personal information management system.

1.4 Thesis Outline

We discuss the requirements for the initial version of the belief service in section 2. In section 3, we describe the three stores that comprise the storage facilities of the belief service: the primary RDF store, the authors store, and the cache of believed statements. Next, in section 4, we explain how the authors and their trust priorities are defined. The logistics of addition, denial, retraction, and deletion of statements are described in section 5. Next comes discussion of the property restrictions implemented and the broader standard designed for the DARPA Agent Markup Language (DAML) in section 6. Ideas for the user interface in section 7 and for future work in section 8 conclude this document.

2 Specifications for the Belief Service

The belief service should be added as a new layer between the main RDF store and the rest of Haystack, and it should provide enhanced functionalities for statement management compared with the plain RDF store. In order to minimize the changes that need to be made in the system upon the introduction of the belief service, the belief service should conform to the same interface as the RDF store. This scheme would allow the belief service to "intercept" the requests to the RDF store made by the rest of the system, and to record all the relevant information, as well as to update its conclusions on the truthfulness of the statements. If the belief service is guaranteed to "intercept" all the requests to the RDF store, it means that it is able to keep the information about the belief values of all statements up-to-date. In this case, the belief service should maintain the set of believed statements that is always current. On the other hand, if there is no such guarantee, the belief service should be able to generate belief values of statements on the fly, at the moment when the statements are queried for by the system.

The belief service should support such operations as addition, assertion, denial, retraction, and deletion of statements. The addition operation is merely for adding a statement to the system, without having its source assert or deny the truthfulness of the statement. In effect, this action "materializes" the existence of any statement by making it present in the repository, but does not create any opinion about the statement, while inviting other sources to express their opinions about it. While Haystack that uses the RDF store alone assumes that each statement in the repository is asserted to be true by the source that has added it, the belief service should allow the source to specify whether it asserts or denies the statement. The reverse operation to assertion or denial is retraction. A source could have no opinion about a statement, which either happens by default or if a source has retracted its assertion or denial of the statement. See Figure 2 for a simple diagram on what opinions a source can have about an existing statement and how it should be able to transfer between these opinions.

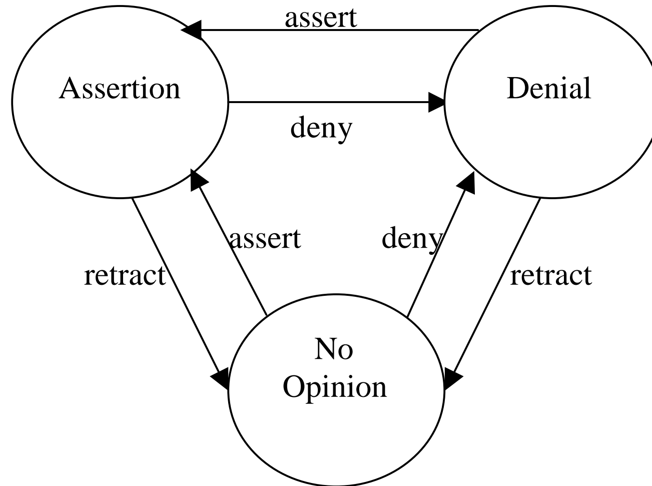


Figure 2: Expressing Opinion About a Statement

Another operation that the belief service should provide is statement deletion. Unlike request for retraction that asks that source's opinion about a statement be removed, request for deletion asks for a statement itself to be removed, and, thereby, for all the opinions of different sources about this statement to be removed too. Statement deletion is not an operation essential for manipulating statements because statements become meaningful to the system only in combination with assertive opinions about them, and the system already provides support for denying or retracting these opinions. However, while Haystack is still at the development stage, we could imagine a situation when some experimental agent erroneously creates an excessive number of useless statements that only clutter the repository and need to be removed. In order to have a reverse operation to plain statement addition, the belief service should support statement deletion. Because deletion might have irreversible effects, the belief service should let the user take charge and should provide only restricted access to this operation.

For the belief service to determine when one of the operations described above is allowed to take affect, it should examine the source that has requested the operation. In order to resolve conflicts that occur when different sources have contradicting opinions, all the sources that act in Haystack should have relative trust rankings. For example, a user could be assigned a top rank, a reliable agent an intermediate rank, and an experimental agent a low rank. It is the prerogative of the user to specify these rankings. The belief

service should be able to assign the default priorities and should provide the user with a way to indicate and update the trust priorities of the sources. Because a user might not always know his preferences for various sources (for example, when he initializes the Haystack system), it should be possible for a user to give a number of sources the same trust ranking. In case there are conflicting opinions from sources with same trust rankings, the most recent one should be trusted. Likelihood that a later opinion is an improved one and should overwrite an earlier one is the rationale for this approach. As a simple case, consider a source that has expressed contradicting opinions. It would expect a more recent opinion to overwrite an earlier one. This is exactly what would happen when the trust rankings of the sources of the two opinions are the same, as they would be for the opinions from the same source. However, this approach might create a problem of the following type. The two sources with equal trust rankings might engage in an infinite loop of each restating its own opinion in order to make it a more recent one. If this situation is detected, a user should either specify unique rankings or request that the sources with the same rankings are assigned different ones arbitrarily.

To know whether the source can be trusted, the belief service should always be able to identify the source that requests an operation to be performed. If the source is stating an opinion (assertion or denial) about a statement, this opinion should be trusted only if its source has the highest trust ranking among the sources of all the opinions about the statement or if its opinion is the most recent among the sources with the equal high trust rankings. If an assertion of a statement is trusted, the statement is believed, and if a denial of a statement is trusted, the statement is not believed. The ability to create a denial of a statement that has not previously been asserted is useful because it allows a source to prevent some less trusted sources from being believed if they assert this statement in the future. When deletion operation is available to different sources in the system, a source should only be able to delete a statement if this statement has opinions made only by lower or equal priority sources.

Property restrictions is another issue that will be explored by the belief service. While more restrictions could be implemented in the future, the initial version of the belief

service should provide an ability to specify the uniqueness of a certain property of a class of objects. The belief service should enforce this uniqueness by only believing the most trusted of the statements about such property. This ability to create a uniqueness requirement was selected to set a simple example of enforcing property restrictions in Haystack. In spite of simplicity, this restriction can be frequently encountered in the information world, e. g. a person can have at most one birthday specified; a car can have at most one Vehicle Identification Number.

Finally, the belief service should have a sensible user interface that would allow future developers of Haystack, as well as potential end-users, to experiment with the functionalities of asserting and denying statements, retracting opinions, specifying property restrictions, and assigning source trust rankings.

3 Stores Maintained by the Belief Service

Since the belief service functions need to be divided between the two layers, the RDF store and the belief service on top of that RDF store, let us first discuss what kind of storage the pure RDF store implementation should provide. It should certainly contain the essential methods for operating on RDF statements, such as the ones for adding, removing, and querying statements. In addition, there is a possibility that the basic RDF store could maintain authors of the statements; however, the question of whether this is an appropriate function for the RDF store has not yet been resolved. It is unclear if the basic RDF store should be responsible for "understanding" the content of the statements it contains, but maintaining the authorship information would require such "understanding." There are several concerns about keeping statements describing the metadata, such as authors, of each statement in the common RDF store.

The first concern is about the implementation of this functionality. The RDF model prescribes that in order to make statements about statements, the referent statement must be reified into a resource and assigned a Unique Resource Identifier (URI). The referring statements can then use the reified resource in the subject or object field. To reify a statement, four extra statements need to be added to an RDF store. They would be there to describe the type of the created URI (type = RDF statement), the subject, the predicate, and the object attributes of this RDF statement. See Figure 3 for an example of how a statement (1) and the information necessary for its reification would be stored in a simple RDF store. Having to reify each statement added to the store would increase by five times the number of statements. Plus adding the metadata such as the author and the date would further increase the size of the general store. In this scenario, the store could quickly grow out of proportion, negatively affecting Haystack's performance.

Storing a reified statement in a three column RDF store:

Subject	Predicate	Object	
<urn:haystack:favorites>	**hs:name**	**“Anna’s Favorites”**	(1)
<urn:statement:md5:84f3…>	rdf:type	rdf:Statement	
<urn:statement:md5:84f3…>	rdf:subject	<urn:haystack:favorites>	
<urn:statement:md5:84f3…>	rdf:predicate	hs:name	
<urn:statement:md5:84f3…>	rdf:object	“Anna’s Favorites”	

Simulating statement reification by introducing a fourth column:

Statement ID	Subject	Predicate	Object
<urn:statement:md5:84f3…>	<urn:haystack:favorites>	hs:name	“Anna’s Favorites”

Figure 3: Alternatives For Reifying a Statement

However, it is possible to avoid reification in practice by maintaining unique statement IDs in the RDF store and using them to reference the statements. Statement IDs are stored in an extra column next to the three columns that contain standard parts of the RDF statement: subject, predicate, and object (see Figure 3 for an example). These IDs are created by using a special algorithm that generates MD5 identifiers to uniquely represent the statements [4]. It is possible to use an ID as a resource representing a reified statement without actually creating four extra statements. The disadvantage of doing this is having to always check which resources refer to these IDs and not to the actual resources described by the RDF statements. If RDF store processes a resource that refers to a statement ID contained in the fourth column, it needs to simulate having the four RDF statements that describe this ID.

The second concern is about the logical distinction that the system would need to make between different statements. Even if the reification question is settled, it still raises a subtle problem. A need arises to differentiate between the original statements and the statements containing the metadata about these statements (such as the statements created during reification and the ones describing authors, etc). Otherwise, we would be stuck in the infinite chain of authorship specification. For example, we would have to specify the author of the statement that says “Statement_URI author Author_ID,” and so on. This problem could be remedied by differentiating between the described types of statements. It is possible to have some kind of mechanism in the system that understands that the

statements containing metadata deduced by the RDF store do not require metadata about them to be generated.

Another question that is unobvious is what should happen if an author denies some statement. How should the denied statement be represented in the basic RDF store? We can declare an RDF store to be a "neutral party," that is simply recording who said what and not making any judgments about the truthfulness of the statements it contains. To enable recording denial on the RDF store level of the system, we can replace statements of the type "Statement_URI author Author_ID" with the statements of the type "Statement_URI assertedBy AuthorID" or "Statement_URI deniedBy AuthorID." Alternatively, we could oblige the basic RDF store to be able to provide information on what statements are believed. For example, we could have the RDF store remove the denied statements and only keep them in the reified form. In this case an RDF store becomes a storage of the true statements only. However, for that it would also need to examine priorities of different authors, and remove the statements only if the authors who deny these statements have the highest priority.

In order not to strain a single RDF store with the metadata that is deduced by the system about each statement, three RDF stores, which provide simple functionalities of statement addition, deletion, and retrieval, are used by the belief service. The first one of them is the primary RDF store, and it corresponds directly to the main RDF store that was used when the system was operating without the belief service. In fact, it contains exactly the same statements that the main RDF store would have contained before the introduction of the belief service. Whenever the statement is requested to be added, if it is not a duplicate of a statement that already belongs to the store, the statement is added directly to the primary RDF store.

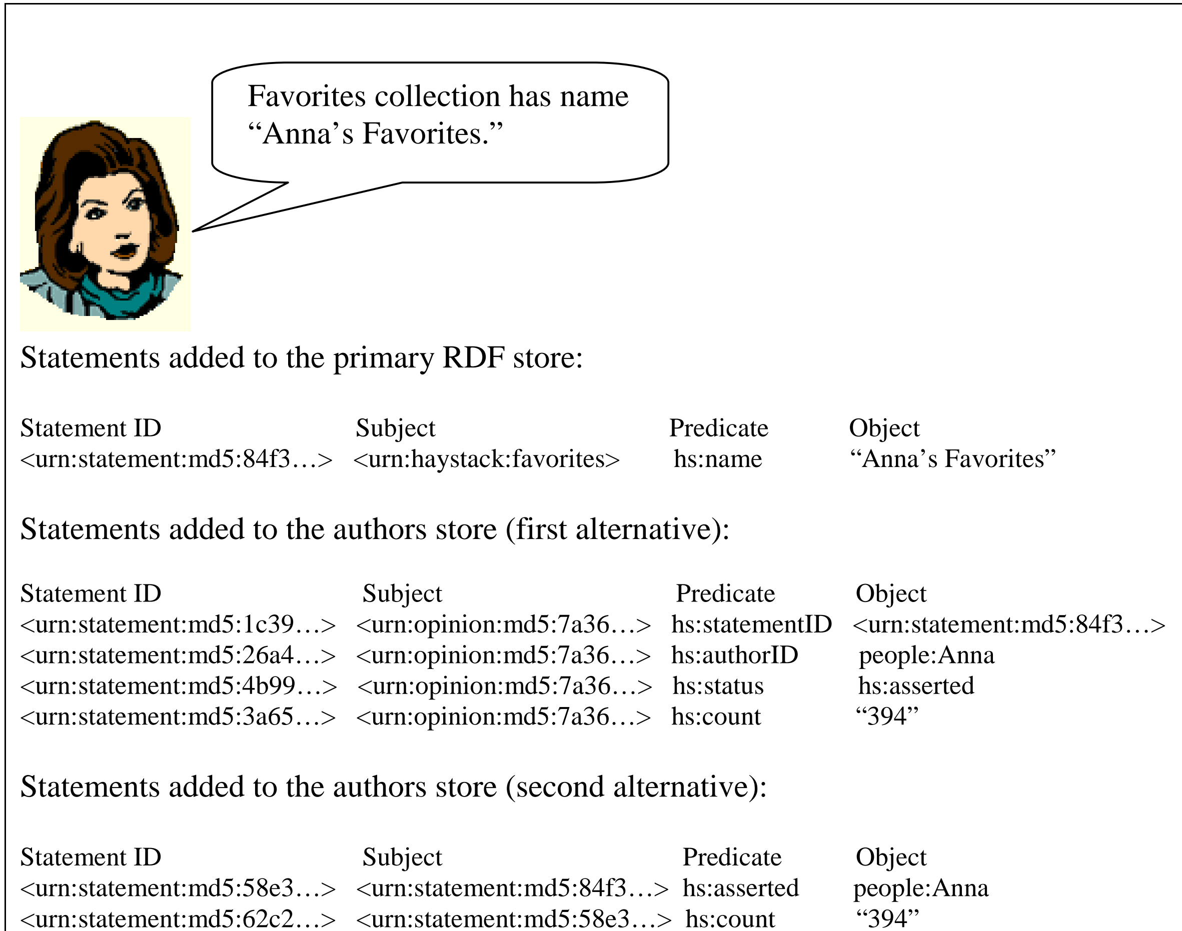
The second store used is the authorship information store. Because, besides author, we need to store such attributes of the statements (described below) as counter and status, we do not utilize the feature of the current implementation of the RDF store that lets one store authors along with statement IDs. Counter represents the sequential order number

that is assigned to a source's opinion about a statement upon its addition to the storage. It is currently used to determine which opinion is more recent. We could have used a date information instead of a counter, but refrained from doing so because computers of different users might have clocks that are not synchronized, and that would introduce unnecessary mix-ups. The status of the authorship information represents the opinion that the source has expressed about a statement, which could be either assertion or denial.

Thus, we want to represent four pieces of information: statement ID, author, counter, and status in a triple statement format of an RDF. The proper RDF model usage would suggest the creation of four separate statements, each one for describing a different attribute with respect to a new opinion ID. See Figure 4 for an example of the statements that would need to be created. However, this implementation would require too many join queries, meaning that for every row containing one sought piece of data, there would be a need to run a subquery for other sought pieces associated with this one. For example, we often want to get information about the authors of a given statement, and what their opinions were. The query would first retrieve subjects of the statements whose object is a certain statement ID, and then would look for the author and status attributes of these subjects.

To avoid constant join queries it is necessary to maintain the most important pieces of connected information in the same row of the table. We designed the authors RDF table to contain statements of two types. The subject, predicate, and object of the statement of the first type are statement ID, status, and author ID respectively. Subsequently, an opinion ID is formed by creating an MD5 identifier of the statement of the first type. The subject, predicate, and object of the statement of the second type are opinion ID, resource "counter," and the counter itself respectively. This design is acceptable because we always know the structure and the nature of the statements stored in the authors RDF store and because this store is only used internally by the belief service. Here we deliberately do not take advantage of the generic representation capability of the RDF data model in order to improve the runtime of the operations.

See Figure 4 for the examples of information representation according to the two alternatives described above. Suppose Anna is the user of Haystack, who wants to customize the name of her favorite documents collection. For that, she makes an assertion which happens to be the 349th opinion expressed by the sources in the system.



Statements added to the primary RDF store:

Statement ID	Subject	Predicate	Object
<urn:statement:md5:84f3…>	<urn:haystack:favorites>	hs:name	“Anna’s Favorites”

Statements added to the authors store (first alternative):

Statement ID	Subject	Predicate	Object
<urn:statement:md5:1c39…>	<urn:opinion:md5:7a36…>	hs:statementID	<urn:statement:md5:84f3…>
<urn:statement:md5:26a4…>	<urn:opinion:md5:7a36…>	hs:authorID	people:Anna
<urn:statement:md5:4b99…>	<urn:opinion:md5:7a36…>	hs:status	hs:asserted
<urn:statement:md5:3a65…>	<urn:opinion:md5:7a36…>	hs:count	“394”

Statements added to the authors store (second alternative):

Statement ID	Subject	Predicate	Object
<urn:statement:md5:58e3…>	<urn:statement:md5:84f3…>	hs:asserted	people:Anna
<urn:statement:md5:62c2…>	<urn:statement:md5:58e3…>	hs:count	“394”

Figure 4: Alternatives For Recording Authorship Information

The third store used by the belief service is the cache, where the set of the believed statements is maintained. This is very convenient because it means that there is no need to check if the statement is believed each time it is queried for. Thus, queries are run against this cache. Normally, cache is constantly maintained up-to-date. However, if there is a need, it is possible to rebuild the cache from the primary RDF store and the authorship information.

4 Managing Authors and Their Trust Priorities

It is necessary for the belief service to be able to identify the sources of statements and maintain information about the relative priorities of these sources. This information enables the belief service to make the decisions about the truthfulness of statements. Here is an overview of how information about authors is kept in the system.

4.1 Identities and Authors

In order to be able to communicate with the belief service, agents and users are currently requested to login into the belief service and obtain a ticket that identifies their session with the service. They pass in this ticket as an argument whenever they call some function of the belief service. Consequently, the belief service gets the information about the author from the session associated with the ticket. Unfortunately, right now, any agent or user is free to name itself as it wishes in order to establish a session with the belief service, so the trusting environment is a requirement. An implementation of unique system-wide identities for various sources that will be represented by a public and a private key of the source is underway. The tickets will soon be replaced by these identities.

4.2 Defining Trust Priorities

All sources of statements in Haystack need to have trust rankings associated with them. The belief service uses these trust rankings to determine the correct opinion if there are several conflicting ones in the system. The ranking is stored along with the source ID in the primary RDF store in the same way as other information about the source is stored. In the future, it will be stored as a part of information pertaining to an identity. By default, the belief service assigns the highest rank to the user and assigns lower equal ranks to all other sources. It is the user's prerogative to assign priority rankings to the sources based on his evaluation of their accuracy and trustworthiness. The belief service checks that these assignments are made exclusively by the user.

Equal ranks are allowed in the system because it is not always possible to tell which source should be more trusted for every pair of sources. If there are conflicting opinions

from equally ranked sources, the most recent opinion is believed. Since different opinions from the same source are treated in the same way as different opinions from the equally ranked sources, the most recent opinion by the source is believed. However, if equal priorities are allowed there is a chance that two agents (automatic sources) could engage in an infinite loop of each restating its own opinion in order to make it a more recent one. This behavior needs to be brought to a user's attention, and the user should assign distinct priorities to these agents. Alternatively, a user can flip a switch in the system, disallowing equal priorities. In this case, the sources, which would otherwise have equal priorities, are arbitrarily assigned distinct priorities. Currently, these ties are resolved by giving a higher priority to the source that was added to the system earlier.

During the initialization of the belief service, all rankings and their source IDs are transferred to an instance of a class that is able to maintain a ranks table. This class was created to eliminate time taken by frequent look ups of the rankings in the primary RDF store and to provide recurring operations on the rankings. It is efficient to keep ranking information in a separate record because the number of sources in Haystack is expected to be limited and the rankings are not expected to change often.

Rankings are represented by real numbers, which makes it possible to insert a source with a ranking between some existing two rankings without reshuffling them (e.g. a rank of 3.75 between ranks 3.7 and 3.8). Except for the rank value of 0, which means that the source is not ever trusted, the values of the ranks do not matter, only their relative relationships. Thus, when presenting priorities to the user, source IDs can be assigned sequential integers that correspond to the ranks that source IDs are assigned internally.

5 Operations on Statements

It is very important to preserve the consistency among the primary RDF store, authors store, and the cache, when such operations as statement addition, assertion, denial, or deletion happen in the system. These operations could affect all three of those stores or some subset of them. This section describes the logistics of these three operations. First, let us review the requirements for the consistency of the three stores:

1. The highest priority opinion about a certain statement needs to be reflected in the cache. If this opinion is an assertion, then the statement should be in the cache, and if this opinion is a denial, then the statement should not be there. The opinion has a highest priority if its source has the highest trust ranking or if it is the most recent opinion from the sources with equally high trust rankings.
2. All opinions need to be stored in the authors store, even the ones that did not take effect at the point when they were stated. This is necessary, for example, in the case when the highest priority source retracts its opinion and it is necessary to determine what opinion should then take effect.
3. The source with a lower priority should not be able to take any action against the opinion of the source with a higher priority, because a more trusted source has more authority. For example, the operation of deletion can affect only opinions that were made by lower or equal priority sources; and only in the case when all the opinions were by lower or equal priority sources is the statement itself deleted from the primary RDF store.

This section presents the algorithms that are designed to satisfy these requirements. Each operation is called with a ticket that identifies the source (Source) who is requesting this operation to be performed. The algorithms presented handle an operation with respect to a single given statement (Statement). However, some method interfaces for these operations allow requests that the operation be performed on multiple statements. For example, it is possible to request to have multiple statements added by passing them in in a single container. The delete method supports requests for deletion of multiple statements, and allows wildcards in place of subjects, predicates, and/or objects. The retraction and denial methods accept a single statement. It will be very easy to write other

appropriate method interfaces in the future, if necessary. Generally, no matter what set of statements is being passed in, the first step is to break it down into individual statements, and then perform the requested operation on each statement as the algorithms below prescribe. Except for a straight forward operation of statement addition, all operations below are accompanied by the pseudocode describing their logistics.

5.1 Statement Addition

The addition operation simply adds the statement to the primary RDF store, and does not make any changes to the other two stores. Because we do not even know the author of the added statement, and the statement is not asserted by anyone, it is not believed and should not be placed in cache. The addition operation is useful in the case someone who has no opinion about a certain statement wants to start a discussion and have other sources express opinions about it.

5.2 Statement Assertion

The assertion operation ensures that the asserted statement is in the primary store, and that the information about the assertion is in the authors store. Also, if the belief service concludes that this assertion is the most trusted opinion about this statement, it adds this statement to the cache. Notice that it is possible for the statement to be in the cache prior to the assertion, in which case no changes need to be made to the cache, and only the fact that this opinion was stated by the current source needs to be added to the authors table.

```
Assertion
   if Statement is in the primary store already then
      if it is asserted by the same author as Source
         record the fact that the assertion is being added again in the authors table
      else
         add to the authors table the fact that Source asserts this Statement
      endif
   else
      add Statement to the primary store and the assertion to the authors table
   endif

   perform the subroutine AddToCache

Subroutines:
AddToCache
   if the statement is not in the cache then
      if subroutine CanAddToCache returns true then
```

```
        add this statement to cache
        check if any statement needs to be removed from cache, because this
        one has a higher priority
      else
        report back that the assertion is not trusted
      endif
  endif


CanAddToCache
  get all the denials for this Statement
  while there are more denials to process do
    if the source of a denial has a greater priority than Source then
      return false
    else
      if  the source of a denial has an equal priority to Source then
        if this denial is more recent than the assertion of this Statement by Source then
          return false
        endif
      endif
    endif
  endwhile
  if there is a contradicting statement made by someone with a higher priority then
    return false
  else
    return true
  endif
```

Figure 5: Statement Assertion

5.3 Statement Denial

The denial operation ensures that the denied statement is in the primary store, and that the information about the denial is in the authors store. Also, if it concludes that this denial is the most trusted opinion about this statement, it makes sure that this statement is not in the cache. Notice that it is possible to add a denial of an entirely new statement that has not previously been asserted. This option might be useful for a source that wants to ensure that less trusted sources are not believed if they assert this statement in the future.

```
Denial
  if primary store contains this Statement then
    if this Statement was already denied by the same Source then
      record the fact that the denial is being added again in the authors table
    else
      add to the authors table the fact that Source denies this Statement
    endif
    perform the subroutine RemoveFromCache
```

```
  else
     add Statement to the primaryRDFStore and the denial to the authors table,
     definitely do not need to remove Statement from cache, because it could not be there
     before without being in the primary store, but we do want to store the denial around
     for future reference
  endif

Subroutines:

RemoveFromCache
  if the denied statement is in cache
     if subroutine CanRemoveFromCache returns true then
        remove this statement from cache
        check if any statement needs to be added to cache, because before it was
        contradicting to the one that was just denied
     else
        report back that the denial is not trusted
     endif
  endif

CanRemoveFromCache
  get all the assertions for this Statement
  while there are more assertions to process do
    if the source of an assertion has a greater priority than Source then
       return false
    else
       if  the source of an assertion has an equal priority to Source then
          if this assertion is more recent than the assertion of this Statement by Source
          then
             return false
          endif
       endif
     endif
   endwhile
   return true
```

Figure 6: Statement Denial

5.4 Statement Retraction

The retraction operation allows a source to retract its assertion or denial of a statement. It is different from deletion because it does not affect opinions about the retracted statement of other, even lower priority, sources. All opinions of the opinion type (assertion or denial) specified by the source are removed from the authors store, and the currently believed opinion about the statement is determined and reflected by the cache. In the case when the retracted opinion was the last one in the authors store about the statement, the statement itself is removed from the system.

There are two alternatives to retraction. The first one is useful when a source wants to change its opinion about a statement from assertion to denial, or vice versa. Instead of first retracting its opinion and then stating a new one, it can just go ahead with stating the new one, which will automatically overwrite the old one. The old opinion will stay in the authors store, but will stop being the most current opinion of the source. The second alternative to retraction is deletion.

```
Retraction
   remove from the authors table opinions of the specified opinion type made
   by Source
   find an opinion with the highest priority about that Statement
   make sure it is reflected by the cache
   if the opinion retracted was the last one about Statement then
      remove Statement itself from cache and primary store
   endif
```

Figure 7: Statement Retraction

5.5 Statement Deletion

Deletion is a radical measure because its aim is to completely remove the statement from the system; it therefore removes not only the opinions about a statement of a source that has requested deletion, but also the opinions about the statement of the lower priority sources. If, after all these removals, there are no opinions about the statement left in the system, the statement itself is removed completely. Even though deletion might seem to be a more thorough-going operation, it actually does not ban the statement from reappearing in the system. If the source’s goal is to state a disagreement with the statement, and make this disagreement persistent, the denial operation is the appropriate choice.

Thus, deletion is most suitable for debugging-like purposes, for example when it is necessary to clean out from the system the statements that one deems useless and unlikely to reappear. Because deletion might have drastic effects, but is not an operation essential for manipulating the statements, a special switch is provided by the belief service. This switch has three modes. In the first mode, none can request statement deletion, this mode is there to make sure that the user does not delete the important data accidentally. In the

second mode, only the user and the agents operating on behalf of the user, which are able to present user's ticket, can request statement deletion. In the third mode, deletion operation is available to all sources, however, they can just request it against the statements that were asserted or denied only by the sources with lower or same priority.

Another reason why deletion should be handled discreetly is that other statements might be referring to a statement that was requested to be deleted. It is the source's responsibility to make sure that no statement is referring to a statement that no longer exists in the system. In the future, the belief service could provide a recursive deletion operation, which, if trust priorities allow, would not only delete a statement itself, but would first search recursively and delete all statements that refer to the statement that is requested to be deleted.

```
Deletion
   if this Statement is in the primary store then
      initialize a boolean flag canRemoveAll that indicates that were able to remove all
      opinions and set it to true
      get all the opinions (assertions and denials) for this Statement
      while there are more opinions to process do
         if the source of an opinion has a greater or equal priority to Source then
            remove this opinion
         else
            set canRemoveAll to false
         endif
      endwhile
      if canRemoveAll is false then
         report back that the deletion could not be performed completely because of lack
         of trust
      else
          remove this Statement from cache if it is there, and from the primary store
      endif
   endif
```

Figure 8: Statement Deletion

6 Singlevalueness

DAML+OIL (DARPA Agent Markup Language + Ontology Inference Layer) is a semantic markup language for Web resources. While a DAML+OIL knowledge base is a collection of RDF triples, it extends RDF to provide more mechanisms to describe relationships between objects. One of its useful capabilities is a support for specification of various property restrictions for classes of objects [8], [9]. For example, when defining a class Person, it is also meaningful to restrict the value for the property parent to the class Person. In addition, it is instructive to restrict the cardinality of the spouse property to be at most one. DAML+OIL allows us to define nine types of restrictions:

1. If an instance of class X has property *p*, then its value must be an instance of class Y.
2. For a named instance *y*, every instance of X must have at least one property *p* that has value *y*.
3. Every instance of class X must have at least one property *p* whose value is an instance of class Y.
4. Every instance of class X must have exactly N distinct values for the property *p*.
5. Every instance of class X must have at most N distinct values for the property *p*.
6. Every instance of class X must have at least N distinct values for the property *p*.
7. Every instance of class X must have exactly N distinct values for the property *p* that are instances of class Y.
8. Every instance of class X must have at most N distinct values for the property *p* that are instances of class Y.
9. Every instance of class X must have at least N distinct values for the property *p* that are instances of class Y.

If Haystack were to implement these restrictions, most of them would need to be enforced when the information about the instances of specific classes is collected, but restrictions of the types 1, 5, and 8 can clearly be observed by the belief service. To observe the first restriction, the belief service could choose not to believe any statement that has an object of type X, a predicate *p*, and a subject of some type other than Y. To

observe the fifth restriction, the belief service can check if statements with object of type X and predicate *p* have more than N distinct values for subject, and if so, order these statements according to their trustworthiness, and believe only N most trusted ones. To observe the eighth restriction, the belief service can check if statements with object of type X and predicate *p* have more than N distinct values of type Y for subject, and if so, order these statements according to their trustworthiness, and believe only the N most trusted ones.

In the future, it would be worthwhile to define what property restrictions are useful and should be enforced in Haystack. In order to set an example of how a property restriction can be enforced by the belief service, a specific case of the restriction of type 5 was implemented as a part of this project. The case sets N to be equal to 1. Indeed, this property restriction is very useful and often encountered when describing various objects. For example, a person can have at most one spouse and at most one full-time job at a time; a problem set can have at most one due date. Also, this property restriction can be useful if there are multiple conflicting statements in the system when only exactly one of them can be correct. While it is the job of the data gathering layer to ensure that at least one value is obtained, the belief service can decide which one of the many values to believe.

Thus, to communicate to Haystack that a certain class of instances can have at most one correct value for an object in a statement with a predicate *p*, there must be added a statement with a class name resource as subject, singlevalued property resource as a predicate, and a property *p* name resource as an object. A resource is defined to be an instance of a certain class by being a subject in a statement with “rdf:type” resource as a predicate and a class name resource as an object. Both agents and users can assert the above statements, and the belief service applies its usual trust checks to decide which property restrictions to believe. Next, when deciding whether to believe a statement S with a predicate *p*, belief service checks the RDF type (the class) of the subject of this statement. If this RDF type description has a singlevalued property *p*, then belief service checks if there are more statements with the same subject and predicate *p*, and if so, it

believes statement S only if this statement has the highest priority among all the contradicting statements. By default, all properties are not singlevalued. See Figure 5 for an example of the set of the RDF statements that describe a class with a singlevalued property, as well as an instance of that class with different values assigned to that property.

Subject	Predicate	Object	
hs:Person	rdf:type	daml:Class	
hs:Person	rdfs:label	"Person"	
hs:Person	hs:creatable	"true"	
hs:Person	rdfs:comment	"A person."	
hs:Person	ozone:icon	<http://localhost:8100/ozone/icons/types/person.gif>	
hs:Person	**hs:singlevalued**	**hs:FullTimeJob**	
hs:singlevalued	rdf:type	daml:ObjectProperty	
hs:singlevalued	rdf:type	hs:ProprietalProperty	
hs:singlevalued	rdfs:lable	"Singlevalued"	
hs:singlevalued	rdfs:domain	rdfs:Class	
hs:singlevalued	rdfs:range	daml:ObjectProperty	
hs:FullTimeJob	rdf:type	daml:ObjectProperty	
hs:FullTimeJob	rdf:type	hs:RelationalProperty	
hs:FullTimeJob	rdfs:label	"Full Time Job"	
hs:FullTimeJob	rdfs:domain	hs:Person	
people:Anna	rdf:type	hs:Person	
people:Anna	dc:title	"Anna Block"	
people:Anna	**hs:FullTimeJob**	**"Designer"**	(1)
people:Anna	**hs:FullTimeJob**	**"Interior Designer"**	(2)

Figure 9: Sample of RDF Statements Utilizing Singlevalueness Feature

The RDF sample in Figure 8 describes a class Person, instances of which can have a FullTimeJob property with at most one correct value for a given instance. The sample also describes Anna, who is a Person, and contains the two statements naming her FullTimeJob. Suppose statement (1) was added by an automatic agent that processes resume documents and deduces information about people from them. Suppose that later, a user of Haystack notices this deduction, and wants to correct it by specifying that Anna's full-time job is as an "Interior Designer." Because of the singlevalueness feature promoted by the belief service, the user can merely assert the statement (2), and this action will automatically overwrite the agent's statement. The information about the

agent’s statement will continue to be present in the system, but this statement will not be believed. In case the user ever decides to retract his assertion about Anna’s job being “Interior Designer,” the agent’s assertion about Anna’s job being “Designer” will again become the believed one.

7 User Interface

To provide a user with easy access to the belief service features described in the previous sections, the existing user interface of Haystack needs to be augmented. While addition and deletion of statements are currently supported by the user interface, assertion, denial and retraction of statements need to be made available to the user in a similar way. Further, a user should be able to view and modify the trust priorities of the sources in the system. Finally, a user should have a clear way of specifying the singlevalued properties of the classes.

7.1 Adenine

Haystack has a prototype user interface named Ozone, and an Adenine console is part of this interface. Adenine is a language that was created to provide an easy syntax for manipulating the RDF metadata in Haystack [5]. For example, to add a statement the command "add" and the subject, predicate and object of the statement should be specified. It makes possible to query for statements by letting the source of the query specify the wildcards to be returned. Review [5] for the examples of the original Adenine syntax. Adenine has been extended to understand the belief service-specific commands. The Adenine console, which can be run alone or is available as a part of the Ozone graphical interface, lets the user manipulate his data and take advantage of the belief service functionalities. The remainder of section 7 can serve as a tutorial of the interface for these functionalities. Figure 10 shows a snapshot of an Adenine console with a sample interaction that utilizes the new commands.

7.2 Statement Assertion, Denial and Retraction

The assertion function can be called similarly to the statement addition function. A user should specify the "assert" command and all statements to be asserted in the curly braces. A statement consists of a subject, a predicate, and an object separated by the spaces. The denial function accepts a single statement that follows the "deny" command. The retraction function accepts a single statement and an opinion status, which can be "asserted," "denied," or "both." The "retract" command retracts the opinions about the input statement of the specified type.

In order to enable a Haystack programmer to experiment with the belief service functionalities from the Adenine interface, the login function was made available. The “login” command, followed by a resource identifying a source and a password, indicates to the system that the subsequent user interface commands are issued by the source that has just logged in.

7.3 Setting Up and Updating the Trust Priorities

The “setrank” command followed by a resource identifying a source and a string with a positive real number sets the rank of the source. The uniqueness feature of the belief service is conveniently being used to enforce singlevalueness of a trust ranking of a source. Therefore, if a user wishes to change a rank of a source, he can assign a new rank, which would automatically overwrite an old rank. The “listranks” command does not require any arguments and returns a listing of all the sources and their trust rankings in the system.

7.4 Singlevalueness

The “singlevalued” command, followed by a resource naming a property and a resource identifying a class of objects, allows an instance of the specified class to have at most one correct value for that property. It is a requirement for the operation that a resource identifying a class is a subject in a statement with predicate rdf:type pointing to a daml:Class object. To remove singlevalueness restriction, the “multivalued” command followed by a resource naming a property and a resource identifying a class has to be used. By default, all properties of a class are multivalued. The “listsv” command can be used to get a listing of all singlevalueness property restrictions employed by the system. If the “listsv” command is used with one argument identifying a class, all properties that are restricted to be singlevalued for that class are listed.

7.5 Sample Interaction in an Adenine Console

Figure 10 shows a sample interaction that is possible in the current prototype of a user interface. First, a user who presents her Haystack user ID logs in. She asserts a couple of

statements and queries to check that the system has recorded information correctly. Next, she denies one of the statements and the same query does not return the statement that is currently denied. The user imposes a singlevalueness restriction on the full-time job property of the class Person. She also requests a listing of the trust rankings to be displayed. After that, a resume agent logs in and asserts that Anna's full-time job is as a "Designer." The user logs in again, asserts that Anna's full-time job is as an "Interior Designer," and issues a query that confirms that agent's assertion was overwritten. The user is unsatisfied with the resume agent's performance, and, therefore, lowers its trust ranking.

```
Adenine Console
Haystack Adenine Console
Version 1.0
Copyright (c) Massachusetts Institute of Technology, 2001-2002.

% login <urn:6uwlsVrIFAk60mq> "userpass"
Result: dQvc9ofprA9YAIn
% assert { people:Anna people:manages people:Lev}
% assert { people:Anna people:manages people:Natalie}
% printset  ( query {people:Anna people:manages ?x})
<http://haystack.lcs.mit.edu/schemata/people#Lev>
<http://haystack.lcs.mit.edu/schemata/people#Natalie>
% deny people:Anna people:manages people:Natalie
% printset  ( query {people:Anna people:manages ?x})
<http://haystack.lcs.mit.edu/schemata/people#Lev>
% singlevalued hs:Person hs:FullTimeJob
% listranks
<Belief Service> 1.0
<ServiceManager> 1.0
<urn:6uwlsVrIFAk60mq> 1000.0
% login <ResumeAgent> "resumepass"
Result: FEPc5abBye3wfHu
% assert { people:Anna hs:FullTimeJob "Designer" }
% printset ( query {people:Anna hs:FullTimeJob ?x} )
"Designer"
% login <urn:6uwlsVrIFAk60mq> "userpass"
Result: RiNmvLlNPLpJkfm
% assert { people:Anna hs:FullTimeJob "Interior Designer" }
% printset ( query {people:Anna hs:FullTimeJob ?x} )
"Interior Designer"
% listranks
<Belief Service> 1.0
<ServiceManager> 1.0
<urn:6uwlsVrIFAk60mq> 1000.0
<ResumeAgent> 1.0
% setrank <ResumeAgent> "0.5"
% listranks
<Belief Service> 1.0
<ServiceManager> 1.0
<urn:6uwlsVrIFAk60mq> 1000.0
<ResumeAgent> 0.5
%
```

Figure 10: Sample Interaction in an Adenine Console

8 Future work

Introduction of a belief service opens up many horizons for creating a more sophisticated information management system. The following are some ideas for future work. The belief service functionalities could be added to the comprehensive user interface. A profile describing all information relevant to a single statement could be made available. More property restrictions could be supported by the system. The system could store dates of creation of opinions, and these dates could serve to provide a snapshot of the system at any time in the past. An automatic expiration of opinions could also be supported.

8.1 GUI Augmentations

Once various views that Haystack graphical user interface Ozone can provide are established, access to the belief service functionalities can be added to this interface.
The possible use cases need to be studied, and the assert, delete, retract, and deny buttons should be provided in the views where users might want to use these functions. Further, a separate priorities management page should be provided, with a listing of all the sources and their priority rankings. The sources could have links leading to their descriptions and to a sample of their opinions. The page should have an intuitive layout, and let the user know which sources are new and only have automatic rankings. The page could also provide the user with an ability to insert one source between the other two, instead of having to enter a numeric ranking that is between the rankings of the other two sources.

An interface for browsing through the statements could also be created. Besides an ability to view a statement, a user should also be presented with all the existing opinions about the statements and their sources. If some opinions are not believed, the justification for that should be provided. A plausible justification would be a pointer to a contradicting opinion and a ranks comparison explaining why the contradicting opinion is more trusted. The statements with the same object and the same predicate should also be listed, explaining which one of them is believed, if the combination of that object and that predicate allows only one correct value for a subject. For any statement that describes its

object as an RDF class, a user should be able to specify the singlevalued properties that this class can have.

8.2 More Property Restrictions

As is suggested in section 6, more research can be done to define what property restrictions could be useful. The belief service could be augmented not to believe statements with a subject of the wrong class or statements that exceed the quota for the different values of a subject. Right now, the hs:singlevalued resource that serves as a predicate is available for describing the case when quota N equals 1. To extend this to a more general case, an hs:PropertyRestriction RDF type should be introduced. Instances of this type should have values for what property restriction they describe. For example, they could specify maximal cardinality restriction (same as type 5 in section 6) and N associated with this restriction, where N can be any positive integer. The hs:PropertyRestriction instance can serve as a predicate in a statement describing a restriction for some property of some rdf class.

8.3 Dates and Their Use

Currently, a counter is being stored to convey the order in which the opinions were added to the system. If it is possible to ensure clock synchronization among all the sources, a date can be stored instead to support ordering of opinions and some additional functions. For example, it would make possible providing a snapshot of the system at any given moment in time by ignoring the opinions that were made after that given moment and determining which opinions should be believed out of the remaining set of opinions. One important change that would need to be made to enable this feature is retention of all the deleted statements. Instead of completely removing a statement or an opinion about a statement from the system, the fact that it was removed from the system should be stored. Even if these statements and opinions had been removed by the present moment, their retention would allow availability of all statements and opinions about them, thus making it possible to produce a snapshot of an earlier moment.

Another function that would be enabled by storing dates is automatic expiration of opinions. This would be a particularly useful feature for statements that maintain dynamic content. Storage of such changing information as web page titles, reports on availability of certain resources, or the definition of when "today" is can be automated by introducing expiration. Right now, similar results could be achieved by programming automatic agents to regularly update a certain property, and making this property be singlevalued. Such updating would overwrite and, effectively, "expire" the previous value of the property. However, expiration would be especially useful if the user wants to ensure that a certain opinion is no longer valid at some future moment, even though the user might not be around or might forget to overwrite it manually at that particular moment.

9 Conclusion

In the course of this project we have designed and implemented a belief layer for the Haystack system. In an environment where opinions about statements come from various sources, a service that decides which opinions to believe is essential. To sort out the believed opinions, we have designed the data storage facilities that use the Haystack-wide RDF store standard for storing general and belief service-specific information. We have defined and implemented the logistics of addition, assertion, denial, retraction, and deletion of statements. The execution of these logistics is guided by the list of priorities of various sources that is maintained by the belief service. The specification and enforcement of the singlevalueness of certain properties was enabled. Finally, a command line user interface for all these belief service features was implemented and explained.

We have proposed additions to the graphical user interface, the exploration of other property restrictions that could be supported by the belief service, and the use of dates to allow obtaining snapshots of the system at some previous moment and automatic expiration of opinions. We hope that future developers of Haystack will pursue these ideas to further improve the belief service aspect of the system. We also hope that Haystack will become an indispensable tool for taking full advantage of the rich personal information space.

Appendix A: Tying Belief Service into the Haystack Platform

This section familiarizes the reader with the Haystack implementation of the RDF concepts. It then explains how the information flow in the implementation of the system was changed to include the belief service. A new data storage architecture had to be designed in order to support the belief service decision making, and the issues that came up during the design are discussed.

A.1 Haystack RDF Representation

Haystack uses a custom RDF library for manipulating RDF data. Resources, which are all entities described by RDF expressions, and literals, which are simple strings or other primitive datatypes defined by XML, are represented using the haystack.rdf.Resource and haystack.rdf.Literal classes, respectively. Both classes derive from haystack.rdf.RDFNode. Statements are represented using the haystack.rdf.Statement class. Like Java strings, Resource, Literal, and Statement objects are immutable.

Any class in Haystack that implements functions of an RDF statements storage needs to extend the IRDFStore interface. This interface contains the common methods for operating on RDF, such as the ones for adding, removing, and querying statements. See Figure A1 for a full listing of these methods. For example, an add method accepts a ticket identifying the source of the statement and a container with the RDF statements to be added. It attempts to perform the requested operation and throws a ServiceException if the operation fails. A query method accepts a ticket identifying the inquirer, a set of statements, a set of existential variables, and a subset of the existential variables to be returned. Existential variables are the “blanks” that the RDF store needs to fill in to satisfy a query. The query method returns a set of requested matching values or throws a ServiceException if the retrieval operation fails.

```
public interface IRDFStore {
    public String login(Resource userid, String password) throws ServiceException;
    public void logout(String ticket) throws ServiceException;
    public void add(String ticket, IRDFContainer c) throws ServiceException;
    public void remove(String ticket, Statement s, Resource existentials[]) throws ServiceException;
    public Set query(String ticket, Statement[] query, Resource[] variables, Resource[] existential) throws ServiceException;
    public int querySize(String ticket, Statement[] query, Resource[] variables, Resource[] existential) throws ServiceException;
    public Set queryMulti(String ticket, Statement[] query, Resource[] variables, Resource[] existential, RDFNode [][] hints) throws
                                                                                                ServiceException;
    public boolean contains(String ticket, Statement s) throws ServiceException;
    public RDFNode extract(String ticket, Resource subject, Resource predicate, RDFNode object) throws ServiceException;
    public RDFNode[] queryExtract(String ticket, Statement[] query, Resource[] variables, Resource[] existential) throws
                                                                                                ServiceException;
    public Resource[] getAuthors(String ticket, Resource id) throws ServiceException;
    public Statement getStatement(String ticket, Resource id) throws ServiceException;
    public Resource[] getAuthoredStatementIDs(String ticket, Resource author) throws ServiceException;
    public Resource addRDFListener(String ticket, Resource rdfListener, Resource subject, Resource predicate, RDFNode object)
                                                                                        throws ServiceException;
    public void removeRDFListener(String ticket, Resource cookie) throws ServiceException;
    public void replace(String ticket, Resource subject, Resource predicate, RDFNode object, RDFNode newValue) throws
                                                                                                ServiceException;
}
```

Figure A1: IRDFStore Interface Code

Haystack provides two implementations of the RDF store. The first one utilizes a relational database with a JDBC interface. However, introduction of the Ozone user interface, with its small but frequent queries, overwhelmed the RDF store with its fixed marshalling and query parsing costs. To improve the speed of the Haystack application, the in-process RDF database called Cholesterol was written in C++ and JNI was used to connect it to the rest of the Haystack Java code base. It was tailored to RDF, so as to optimize the most heavily used features of the RDF store while eliminating a lot of the marshalling and parsing costs.

A.2 Bootstrap File

When Haystack initially starts up, it gathers information about the essential services it needs to run from the bootstrap file written in Adenine, which is a Turing-universal programming language specifically suited for manipulating RDF metadata with a syntax resembling Notation3 [5]. The original bootstrap file used by Haystack initialized the main RDF store to be the system's connection to the RDF store functions. This main RDF store was realized by the haystack.server.rdfstore.CholesterolRDFStoreService, a class that implements the IRDFStore interface. The belief service was inserted as a new layer between the main RDF store and the rest of Haystack to provide enhanced functionalities for statement management compared to the plain RDF store. In order to introduce the belief service into Haystack, the bootstrap file was rewritten to direct the

system to the haystack.server.rdfstore.BeliefService implementation of the IRDFStore. The belief service was assigned three different RDF stores realized by the haystack.server.rdfstore.CholesterolRDFStoreService. These stores are: the primary RDF store, which corresponds to the main RDF store; the authors store, which maintains information about the authorship of the statements; and the cache store, which maintains a set of believed statements. Like the main RDF store, these stores are persistent across multiple runs of Haystack from the moment the system is first initialized.

When the CholesterolRDFStoreService implementation of IRDFStore is used, the requests for statements to be added or removed come from the rest of the system, so the belief service is able to “intercept” these requests and make appropriate changes to the stores it maintains. This behavior allows the belief service to always contain the most recent information and have decisions about the truthfulness of each statement readily available.

However, this possibility of processing changes as they happen might not be available with all IRDFStore implementations. Research is currently being done to enable the system to connect to the user’s Microsoft Outlook application and upload all of the user’s e-mails into Haystack. The metadata about the e-mails is also represented with RDF; however, it arrives from outside the system. For example, the primary RDF store of the belief service can be realized by haystack.server.cdo.CDORDFStoreService, a class that implements IRDFStore and enables a connection with Outlook through the Collaboration Data Objects (CDO) technology [7]. In this case, belief service needs to be able to process belief values of statements on the fly, at the moment when the statements are queried for by the system.

Bibliography

[1] David Karger and Lynn Stein. Haystack: Per-User Information Environments. February, 1997. http://haystack.lcs.mit.edu/papers/karger-stein-9702.ps.gz

[2] David Huynh, David Karger, and Dennis Quan. Haystack: A Platform for Creating, Organizing, and Visualizing Information Using RDF. http://www.ai.mit.edu/people/dquan/overview.pdf

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

[4] Ronald Rivest. The MD5 Message-Digest Algorithm. http://theory.lcs.mit.edu/~rivest/rfc1321.txt

[5] Dennis Quan. Introduction to Haystack. http://www.ai.mit.edu/people/dquan/haystack_intro.htm

[6] Dennis Quan. An Ontology for Personal Information Stores. http://www.ai.mit.edu/people/dquan/haystack.html

[7] Overview of CDO. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cdo/html/_olemsg_overview_of_cdo.asp?frame=true

[8] Reference description of the DAML+OIL ontology markup language. Property Restriction. http://www.daml.org/2000/12/reference.html#Restriction

[9] Annotated DAML+OIL Ontology Markup. Defining property restrictions. http://www.daml.org/2000/12/daml+oil-walkthru.html#restrictions

[10] Svetlana Shnitser. Integrating Structural Search Capabilities Into Project Haystack. June, 2000. http://haystack.lcs.mit.edu/papers/svetlana.thesis.pdf