Data Manipulation Services in the Haystack IR System

by

Mark Asdoorian

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Computer Science and Engineering

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1998

Author ....
Department of Electrical Engineering and Computer Science
May 22, 1998

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

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

Data Manipulation Services in the

Haystack IR System

by

Mark Asdoorian

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 1998, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Computer Science and Engineering
and
Bachelor of Science in Computer Science and Engineering

Abstract

The Haystack project seeks to design and implement a distributed, intelligent, personalized, information retrieval system. Haystack archives documents with metadata, which is also indexed by the system to improve query results. To support this system, an infrastructure needed to be designed and implemented. This thesis covers the overall design of that infrastructure with a focus on the service model, event model, remote communications model, and necessary services for the addition of our core metadata for documents in the system.

Thesis Supervisor: David R. Karger
Title: Associate Professor

Acknowledgments

I would like to thank Professors David Karger and Lynn Stein for dreaming up the idea of Haystack and for providing their guidance as we designed and developed our model for Haystack.

I would especially like to thank Eytan Adar, the other M.Eng. student who helped design and implement Haystack. I got a lot out of our design discussions and he was always there to answer questions.

I would also like to thank the numerous UROPs who have worked on Haystack over the past 2 years. They are (in alphabetical order) Christina Chu, Dwaine Clarke, Lili Liu, Aidan Low, Eric Prebys, Jing Qian, and Orion Richardson.

Finally, I would like to thank Josh Kramer for actually integrating Haystack into the intelligent room. It was an inspiration to see Haystack being used in such a futuristic setting.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's computing environment, the management of the ever-increasing amount of digital information encountered by a user threatens to overwhelm the user's time and resources. The amount of information available over the world-wide web alone is staggering, but when this information is combined with the mass of information produced by the user and his colleagues, the amount of information a user needs to search increases dramatically. Thus, a tool to quickly and conveniently access this information is required by the user. Information retrieval (IR) systems exist to index text information and provide the ability to query the index, but traditional IR systems provide poor interfaces, do not handle different file-formats, and do not adapt to a specific user.

1.1 The Goal of the Haystack IR System

The Haystack IR system [13] is an effort to construct an intelligent, adaptable, distributed interface to common IR engines. The category of IR engines encompassed by Haystack is that of text-based and database engines. Much of the information that a user will query for is text, or can be transformed into text (in the form of simple format translation or in the form of description), and that is indexed by the text IR engine. One of the key features of Haystack that sets it apart from other IR systems is that along with the object itself, Haystack indexes metadata containing various descriptor fields for the object (added by both Haystack and the user). The index of this metadata can be maintained by a database engine to allow database-style queries. Haystack will use these fields to adapt to the user's information needs.

1.2 The Old Haystack

A first implementation of Haystack was done in Perl. This system was composed of a series of scripts which provided an interface to an underlying IR system. The user interface was provided in a number of facets including command-line, CGI WWW scripts, and Emacs macros. The Perl version of Haystack performed poorly and did not follow an object-oriented model that we now believe the system should. It did, however, both provide the basic building blocks we desired to explore the possibilities of building an intelligent adaptable IR system and help us flush out our design goals.

1.3 The New Haystack

Based on observations made of the Perl version, a new version of Haystack needed to be designed. First, we decided to structure the metadata in a more object-oriented model. It is important that all metadata be first-class and indexable. The metadata representation we chose is digraph structure where the nodes contain the data and the links indicate how the data is related to each other. We believe this is an ideal representation for specifying relations between parts of a document and between documents themselves. Given our new object-oriented approach and the need to be distributed and multi-platform, we chose Java as the development environment.

We also decided to make Haystack function under an event-driven service model. All functionality in Haystack is abstracted into services. For instance, we have a service that allows us to lock resources, and we have a service that fetches the contents of a URL. Some services simply provide some piece of functionality that other services need to use (e.g. the resource locking service mentioned earlier). We call the set of these services that all other services rely on to function our *core* services. Other services are driven by events. Events in our system represent a change to the metadata. Our example of a service that fetches the contents of a URL is an event-driven service that listens for a URL to be added to the metadata. Our data manipulation services, which fill out the metadata for a document, are mostly event-driven since they extract metadata based on existing pieces of metadata.

1.4 The Problem

To build the infrastructure for our new Haystack system, we needed to define and implement the specifics of the data model, to design the service architecture, and to implement a set of core services. The design and implementation of the data model and core services is presented in detail Adar's thesis [1]. We go over them in this thesis so we can understand the fundamentals that the rest of Haystack is built on top of. The problem solved by this thesis is twofold. First, it discusses the design and implementation of the service architecture, including the service model in general, the event model that supports event-driven services, and the remote communications model that allows Haystack to have services distributed across the network. Second, it discusses the design and implementation of our core data manipulation services that perform the steps necessary to archive, extract metadata from, index, and query for documents in our system. We have also made these data manipulation services easy to extend so that users can customize their system by plugging in pieces of code, in both the language Haystack is written in and other languages.

1.5 Thesis Overview

This thesis covers the design and implementation of the service architecture for Haystack and the data manipulation services that use this model. The next chapter goes over some background material for IR in general and then more specifically some issues with the distributed nature of Haystack. Chapter 3 lays out the overall design of the Haystack IR system. In chapter 4 the service model for Haystack is presented. The event model for triggering services is described in chapter 5. Chapter 6 discusses the communication model for remote services. Examples of the design and implementation of data-manipulation services are presented in chapter 7. We conclude with chapter 8 with a review of the path Haystack has taken and where it can lead to in the future.

Chapter 2

Background and Related Work

It is necessary to understand the field of IR before the driving forces for Haystack can be understood. A basic IR system must provide the ability to query a corpus (collection) of documents against keywords and return a list of matching documents. Older IR systems worked on fixed, text-based corpa while newer IR systems such as web search engines continually update their corpa, and some provide facilities to index documents other than text (e.g. multimedia documents such as pictures and audio clips).

2.1 Standard IR Systems

Usually, the IR system will index the corpus, storing keywords in a data structure for fast searching. One example of the way a basic IR system would work is store the terms of a document in a lookup table that maps the terms in the documents to the identifiers for those documents. A query for a search term simply looks up the term in the table to find documents matching the search term. The goal of every IR system beyond this simple system, which returns an unordered list of matches, is to return to the user the best set of matches to a query made against the indexed documents, ordered by how well they match the query.

Various improvements can be made on the indexing or querying side of this simple IR system to attempt to return a better set of matches. At indexing time stemming reduces the words in the input text to their roots to avoid indexing words like *run* and *runs* as separate words. Similarly, a thesaurus can map words to a common meaning. Also, common words such as *a*, *the*, and *and* can be excluded from the index as they really don't carry any semantic meaning, and virtually all documents contain them. On the query end a thesaurus can map query terms to common meanings (e.g. *fast* and *quick*). Also, the number of occurrences of words in documents, if recorded, can be used to rank query results. Relevance feedback is method for refining queries. Based on documents that the user selects as relevant to the query from the initial result set, new query terms are extracted.

All IR systems must deal with the conflict of precision verses recall. Precision is the percentage of documents in the query result set that actually match what the user was really looking for. Recall is the percentage of total documents in the corpus that match what the user was looking for that are included in the query result set. These two numbers are in conflict, since if one improves, the other will generally diminish. For instance, if a user broadens a search topic from macintosh computers to just computers, he will get many more relevant matches, thus increasing recall. However, the user will most likely get a much greater percentage of query matches that are not relevant, decreasing precision.

Another model for storing the terms in the index is the vector space model. In this model, documents are mapped to high-dimension vectors of terms, and a query simply finds the documents whose vectors are at the smallest angle to the query's vector. A popular measure in this model for each dimension of the vector uses term frequency times inverse document frequency (TFIDF). This increases the measure if the word occurs frequently in the document, but decreases the measure if the word occurs often across documents.

Haystack's goal is to build on top of such systems as were discussed in this section.

2.2 Personalization and User Interface Improvements

Both commercial companies and research institutions have been exploring the personalization of IR systems and improving the query interface to adapt to user's specific UI requirements. Personalization includes allowing easy customization of the IR system as well as indexing a user's document collection instead of indexing the entire collection of documents in the world as many of today's WWW search engines try to do. User interface improvements to the query interface explore ways of presenting the results to facilitate faster parsing of them by the user and easier query refinement.

Companies like Fulcrum and Verity are integrating their IR engines with popular Windows software. Both of them offer products for Microsoft Outlook which integrate searching right onto a user's desktop. The user has control over the document collection (generally, the user's email, hard drive, and web pages) and can specify heterogeneous queries that incorporate free text with database-style querying. Other search engine companies are offering personal editions of their products that run on a user's PC, including Digital Equipment Corporation which offers Personal Search 97 [4], a personalized version of their Altavista search engine.

The SenseMaker project from Stanford is another instance of an improvement in the user interface to an IR system [2]. The SenseMaker query results interface is highly customizable. As well, it provides means of searching heterogeneous data sources. The query results view can be customized with regard to both what information is displayed about matches, and how matches are grouped. Through grouping matches and SenseMaker lets users indicate a context for their information need, which SenseMaker can act on by restricting or broadening the search criteria.

A variant of the IR system is an approach from XEROX PARC called scatter-gather [10]. This approach is an improvement on the UI for traditional IR interfaces combined with the use of clustering. Clustering is a method whereby the result set for a query (or any set of documents) is divided into a set number of clusters, or categories. Scatter-gather provides a visual clustering of query results, and allows the user to pick a particular cluster which will cause the interface to refocus on that cluster. The cluster will then be reclustered by itself and the resulting clusters displayed to the user.

2.3 Resource Discovery and Collaborative Filtering

The Harvest system [3] has implemented many of the facilities we desire of Haystack, though they are for an arbitrary group collection, not a user's personal collection. It provides access to multiple information sources; Haystack supports multiple IR systems. Harvest is distributed and has defined a Summary Object Interchange Format (SOIF) for passing around document descriptions between its parts; Haystack is distributed and we need to define a way to pass our metadata to external processes. Harvest contains a subsystem called Essence which performs metadata extraction on documents based on their type. While Harvest uses this to summarize documents to improve the efficiency of indexing, Haystack also wants to do this to allow database-style queries against documents as well as free-text queries.

What's Hot [14] is a distributed collaborative filtering system. It allows users to rate how documents pertained to queries as they are going through the result list. It then uses this information for future queries to reorder the result set. This relies on active user participation, though, and while that provides some improvement, we would like Haystack to perform passive observations to learn about the user. As well, this does not personalize the learning for each user, but instead performs learning globally across all users.

2.4 Distributed Systems

The Content Routing project [20] provides a distributed system for access to a variety of information repositories. It defines a hierarchy of content routers that route query requests and browsing requests down the hierarchy until leaves are reached. The leaves are the actual information repositories themselves. This provides a distributed system for querying related to the way we want Haystacks to be able to query each other.

We now discuss two means of communication that exist for processes in distributed systems to communicate with each other.

2.4.1 CORBA

CORBA (Common Object Request Broker Architecture) defines a model for encapsulating code to communicate with other CORBA-compliant code over a platform- and language-independent Interface Definition Language (IDL). CORBA defines an Object Request Broker (ORB) that acts as the bus between CORBA objects ferrying method requests between objects. Objects can live anywhere on the network. Thus, CORBA provides a language-independent means for objects to communicate with each other using IDL across an ORB.

2.4.2 RMI

Since we are implementing Haystack in Java, we have access to the remote method invocation (RMI) facilities of Java. Java defines an RMI mechanism whereby an object defines some of its methods to be remote. That means that a Java program running in a different process (even a different machine) can call those methods in that object as long as certain registration procedures have been followed. Generally, an object calling a remote method knows it is calling a remote method, that is, the remote method is not meant to be run locally. An example of this would be if a thin client machine wants to offload some processing onto a server, it could make an RMI call to the server. The security in RMI allows the restriction of access to remote objects to authenticated users.

Chapter 3

Haystack Overview

Haystack has several goals all designed with the intent of reinventing the IR system to allow users to more quickly and more effectively fulfill their information needs. To achieve this, Haystack needs the following core functionalities:

- Retrieve documents from various types of locations, potentially keeping a static copy for all time[1]
- Extract and allow addition of metadata that describes documents
- Index the textual representation of documents
- Search against both the text index and the database of metadata

Archiving is optional as it quickly doubles the amount of data stored in Haystack. Indexing is currently done with an off-the-shelf product as is the maintenance of the metadata database. The rest of this chapter first discusses what we think an IR system should entail and second, how we have designed our core infrastructure to support this model of an IR system.

[1]By documents we mean any kind of body of information, be it HTML document, email message, audio clip, etc.

3.1 The Personal, Adaptive, Networked IR System

What does it mean for Haystack to be a *personal, adaptive, networked, IR system*? The key features that we desire Haystack to implement to merit this description are

- **Basic IR functions:** Haystack should provide a way to index the text of documents and then query this index, displaying query results in an easy-to-read format.

- **Easy customization:** Haystack should allow users to customize their IR system to their specific environment. This means allowing them to select the IR engine and database system to use, and allowing users to extend Haystack to handle various document types. It also means allowing the user to configure the UI to their liking (particularly with regard to query results).

- **Easy Annotation:** Haystack should allow the user to annotate documents in the index. These annotations should be indexed so that they can be matched by future queries, so two users with the exact same documents indexed may get different results on the same query if they have different histories of solutions to their information needs (recorded by their annotations).

- **Distributed Services:** Parts of Haystack should be able to be run outside of the main Haystack process. This will allow greater efficiency if the user has access to multiple machines, and it will allow program add-ons that run on different platforms.

- **Learning from Past Queries:** Haystack should observe the user's query behavior and learn from it to provide a better ranked query result set the next time a user performs a related query.

- **Communicating with Other User's Haystacks:** Haystack should be able to query the Haystack's of friends and colleagues of the user when it cannot find enough relevant documents to meet the user's information need.

We have implemented most of the first four features in Java. The UI to Haystack is a primitive command-line interface right now, though a fully-featured web server interface is in development and near completion. We now have infrastructure in place in Java to make implementing the last two features easy. We will first quickly go over the Perl version and its failings and then give an overview of the Java version.

3.2 The Perl Version

The Perl version of Haystack was completed during the summer of 1997. It interfaces with three IR systems (MG [16], I-search [11], and grep) and supports basic archive, index, and query commands. The Perl version has many metadata extractors for a variety of types including many types of email, HTML, postscript, $\LaTeX$, and more. It also carries several UIs including a web server/CGI script interface, Emacs interface, and command-line interface. While it performed poorly, the development of this prototype in Perl helped our group flush out some ideas and come up with a solid design for a personal, adaptive, networked, IR system.

The operation of the Perl version used forking to allow more than one file to be processed at a time. We discovered that managing these forks was impossible, and the code managed to get Perl to produce segmentation faults and other internal errors.[2]. These errors differed depending on the platform and version of Perl. As well, trying to feign an object model for the metadata proved quite cumbersome as did the object-oriented facilities introduced with Perl version 5 [22]. Finally, the code often called shell commands that were often other Perl files, adding even more overhead. While we did try to optimize the code wherever possible, the Perl version still ran quite slowly.

[2]It is not possible with Perl code alone to generate a segmentation fault, as there are no memory-manipulation operations in Perl; these errors were due to bugs in Perl itself

3.3 The Java Version

To remedy the problems of the Perl version, we decided to move to an object-oriented, portable language: Java. Using the threads that are available in Java, we can maintain control over the amount of parallel work we do (unlike forking in Perl). With a real object-oriented language, we are also able to create an object-oriented data model for the metadata that would be well-suited for storage in an object-oriented database. Thus, the new model can be broken up into the following subsystems:

- The object-oriented data model that stores the metadata describing the documents in our IR system.
- The underlying information repositories such as IR engines and databases that store our searchable information.
- The distributed, data-driven service model that provides facilities for creating and maintaining our DM objects and providing an API for UIs to call.
- The UI which can come in many forms: command-line, web server, Java applets, Emacs, etc.

Figure 3-1 shows how these subsystems interact in Haystack. We can see that the system forms a 3-tier architecture. The bottom tier corresponds to the storage and index systems, the middle tier to our data model and services, and the upper tier to the UI. Note that a remote service can be part of the upper or middle tier. Also note that services local to the main Haystack process serve as a nexus for most of the other subsystems. The following sections describe some basic background for the Java version of Haystack necessary to understand the rest of this thesis.

If the reader is unfamiliar with Java, it would probably help to review the glossary of basic Java terminology in appendix section A.1. If the reader is unfamiliar with basic object-oriented terminology (e.g. inheritance, methods, etc.), it would help to read the first chapter of a reference book to a popular object-oriented language such

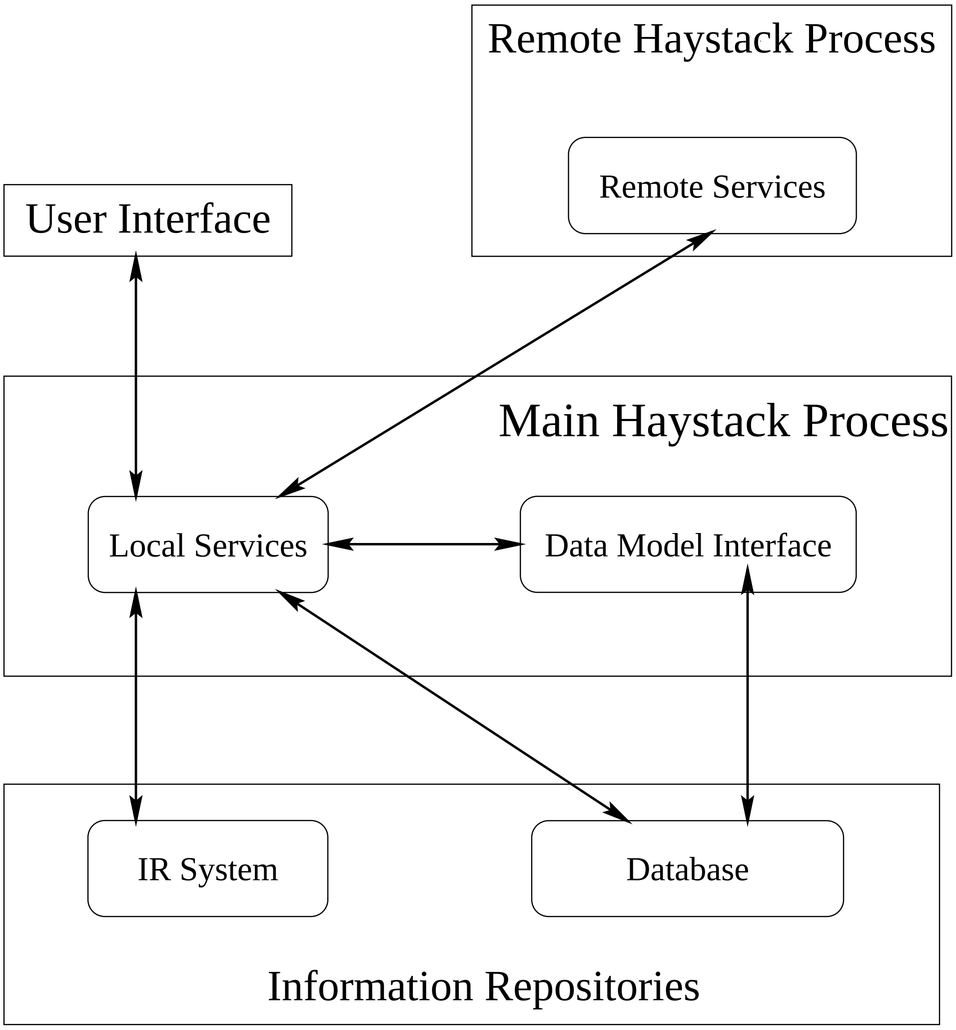


Figure 3-1: The Haystack IR System Architecture

as C++ or Java as most of these books start out with an introduction to object-oriented terminology. As well, while going over the following sections, it might help to refer to the glossary of Haystack terminology in appendix section A.2.

3.3.1 The Data Model

The data model (DM) is based on drawing relationships between parts of a document and between documents. A digraph structure is ideally suited for this. We call a base node in the digraph a *straw*. A straw is a first class object that can be indexed by Haystack.[3] Every straw is assigned a globally unique ID to identify it. These IDs are currently implemented as `BigInteger`s, so there is no limit to how big they can grow. Haystack contains three core types of straws from which all other straws are extended: tie, needle, and bale. We describe these four basic DM elements in the next four sections, and then discuss some implementation issues with how naming works in our DM and how to use the DM objects.

Straws

A straw is a basic node in a digraph. Its links to other nodes are labelled. Labels on a link between two straws indicate the relationship of the straw at the back end of the link to the straw at the front end of the link. Straws are always linked together by ties which are discussed in the next section. Further details of how to use straws will be discussed in *Using the DM* in this section.

Ties

A tie, like any DM object, can have forward and back links. However, since it is used as the link between two straws, it has special forward and back pointers to indicate the two straws it links together. Thus, all links to and from a straw are formed with ties, yet ties themselves can also have links. We believe this provides a

[3] In general, we will use the term *straw* and *DM object* interchangeably to represent an object that is included in the data model.

powerful model that allows us to annotate links between two straws by making the links first-class objects.

Needles

Needles are straws that contain a data element. These are the pieces of information that a user would be searching for. Typical needles that are included for every document are location needles, file type needles, body needles, and text needles. The generation of these needles during the archive process is described in detail in chapter 7.

Bales

Bales are collections of straw. Any straw can point to multiple straws, but a bale also represents an abstract collection such as the emails in an RMAIL file or the files in a directory or the matches to a query. Every document is represented by a bale, `HaystackDocument`, that represents the collection of needles for the document. For every document that represents a collection of documents, a bale for the particular type of collection will be linked to the Haystack document bale for the document.

Naming

Figure 3-2 shows the object hierarchy for the DM objects we have defined in Haystack so far that have subclasses. The straw class and the three core classes each have a corresponding interface. The names in square boxes are interfaces that declare all of the methods necessary for operating on DM objects. The names in the rounded boxes are the classes that implement those interfaces. Anything acting on a core DM object should use the interface to declare its reference to the object. Using the interface names instead of the class names allows us to change the implementation of the DM without having to go into the code and change data type declarations. For a more in-depth discussion of the DM and storing heterogeneous data, refer to Adar's thesis [1]. The following two sections describe the DM implementation in enough detail to understand the rest of this thesis.

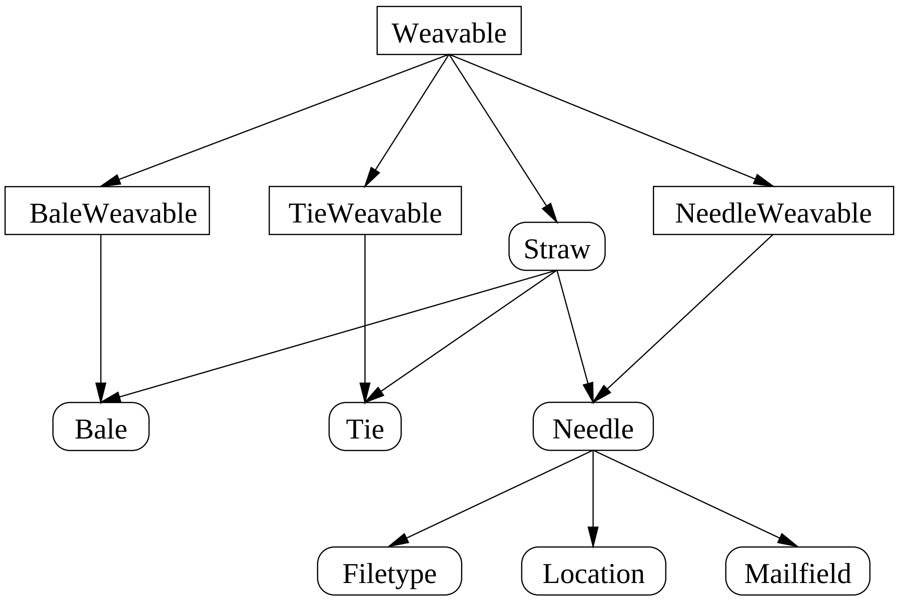


Figure 3-2: Data Model Object Hierarchy

DM Implementation

All objects in the DM reside in the package `haystack.object` or in a sub-package of this package. The class `StrawType` contains static references to the fully-qualified class names of all of the DM objects. When a service wants to refer to the type of a particular DM object, it uses these static references. Following Java conventions, all of our class names are capitalized. The package for every DM object that has subclasses has a sub-package to hold that object's subclasses. The sub-package has the same name as the subclassed object itself except that it begins with a lowercase letter. Thus, the three core DM objects have classes in `haystack.object`. Three sub-packages, corresponding to those class names with a lowercase first letter, reside in `haystack.object` to hold subtypes of those three core straw classes. We refer to the *type* of a DM object as its *straw type*.

We chose to implement a hierarchy of classes for our straw types. We could alternatively have defined our own typing scheme and stuck with only our three core classes of straws. To specify a particular straw type, we would just set a *straw type* field. However, we decided against this approach so we could use Java's built-in typing. This does have the drawback that when user's want to extend Haystack, they must define new straw objects instead of just setting a straw type field in one of our base straws.

If a request is made to automatically tie straw *s1* to straw *s2*, the request first tries to create a tie that matches the straw type of *s2*. If it can't, an attempt will be made to *walk up* the object hierarchy for *s2* until one of the three core DM objects is reached. If so, then a new core object is returned. Otherwise, a new instance of whichever object whose straw type matched first is returned. For example, if we request a tie to be generated for `haystack.object.needle.foo.Bar`, that would be matched first by `haystack.object.tie.foo.Bar` if it can be found, next by `haystack.object.tie.Foo` if it can be found and finally, by `haystack.object.Tie`. We will see how this is useful in the next section when we discuss the ways to traverse the metadata graph.

Using the DM

Anything inside the main Haystack process can directly access the DM. Remote processes must go through local processes running inside the main Haystack process to access the DM. When anything is modifying a DM object, it must lock the object to prevent concurrent changes. In the future, if transaction processing is added to the system, then concurrent changes could be allowed, as could remote changes.

When traversing the metadata graph, it is important to understand how we connect our straws together. As we discussed earlier all links in the digraph are labelled. The label for a link is the straw type of the tie that forms the link. The straw type of most ties corresponds to the straw type of the straw intended to be pointed at by the forward pointer. For example, a location needle's straw type is `haystack.object.needle.Location`, and the tie that points to that needle has a straw type `haystack.object.tie.Location`. We could also have this location tie point to `haystack.object.needle.location.URL` as discussed in the previous section. Figure 3-3 shows an example of a document bale connected to a URL location needle by a location tie. Any changes to the link between two straws must be carried out in both the tie and the straw that it points at to maintain consistency in the DM. We cannot, for instance, change a forward pointer from needle a to needle b without both removing the back link to the tie in needle a and adding a back link to the tie in needle b.

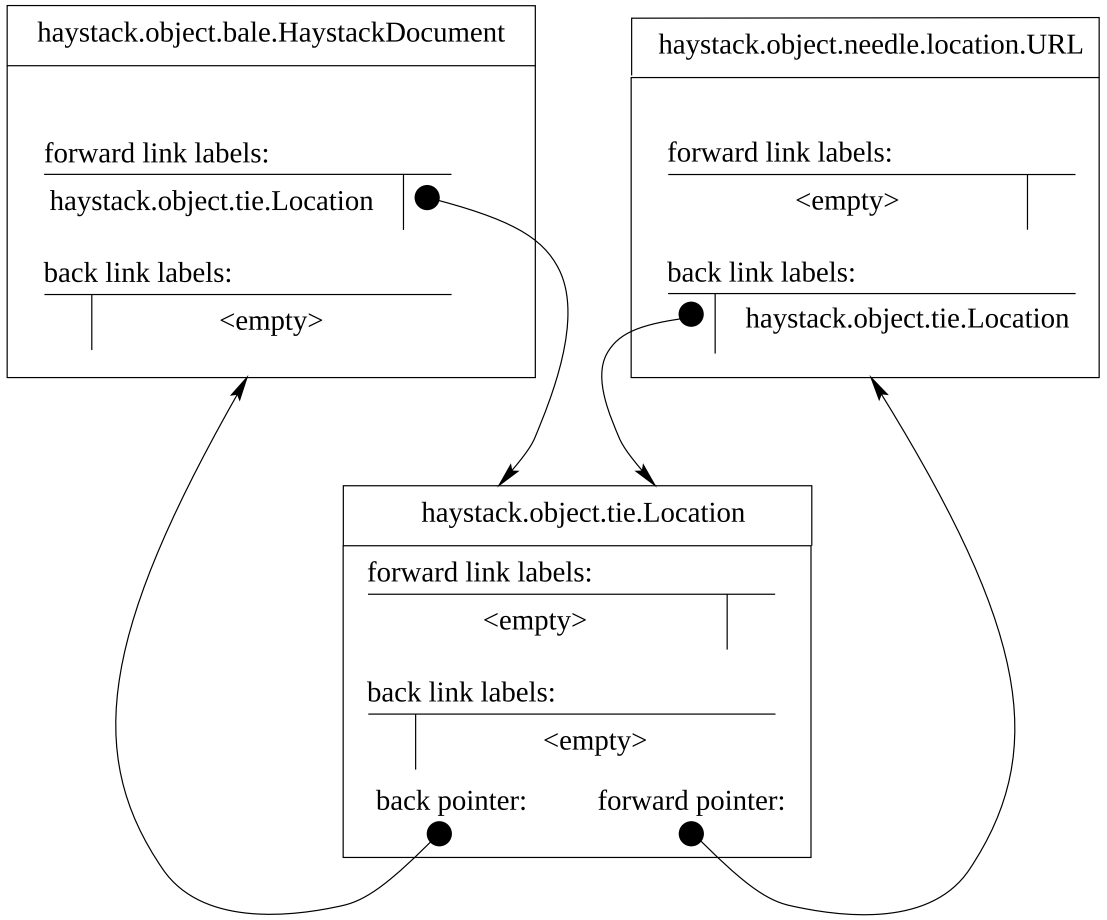


Figure 3-3: Straw Connection Example

Now that we know exactly how two straws are linked together, we can discuss the various means of creating links in and traversing the metadata graph. Straws provide methods for attaching ties to the list of forward and back links. Straws also provide a shortcut to attach themselves to another straw; this method automatically creates the tie needed to connect the two straws together in the manner discussed in the previous section and performs all of the necessary connection operations. Straws provide methods for directly accessing a tie that is linked to a straw through the label for the link as well as a means for accessing the straws that are *tied* to the straw by specifying either the label whose ties it should follow or by specifying the straw type of the object (in that case the straw will search over all labels).

From Figure 3-2 we can see that file type, location, and mail field needles all have subtypes. When we make ties to link specific subtypes of these needles, we only have a tie defined for the base type, so that is the type of tie that is generated. We have also defined a special kind of tie called a causal tie that does not follow this rule. It can connect any kind of straws together. The causal tie indicates that the straw pointed at by the back pointer of the tie was caused by the straw pointed at by its the forward pointer. Thus, if we want to find a pointer to a location needle, we will look for a label *haystack.object.tie.Location*. This is much easier and much more efficient than having to check for each type of tie if we were to have a tie for every different location needle.

3.3.2 Promises

Sometimes, the data for a needle might be quite large. In this case, we want to have a *promise* for the data such that we do not have to store the data but instead, store a means of computing it. For instance, storing the bodies of email messages that are contained in an RMAIL file would double the storage size for an RMAIL file. Instead, we store a method for extracting the message inside a promise. Promise classes reside in the `haystack.object.promise` package and all implement the `haystack.object.Promise` interface.[4] When a promise is set as the data of a needle, the promise is fulfilled each time the data is requested from the needle.

Promise classes all provide a `fulfill()` method that returns a stream (specifically a `Reader`) containing the promised data. Note that any needle whose data can be a promise must return it's data as a Reader. This means that if the data is a string, it must be put into a `StringReader`. A special promise class, `CommandPromise` pro-

[4]`Promise` is not a fourth straw class despite the fact that it is named in a similar fashion. Promises are the data that go into some needles and are thus *objects* that Haystack uses warranting their placement in the `haystack.object` package.

vides static methods that abstract the call to command-line programs that produce the data for a promise. Though not integrated yet, a cache will be used to cache promise fulfillments so that if a particular promise is fulfilled several times in a row, it will not have to perform the computation multiple times. Specific promises are discussed in chapter 7 with the data manipulation services that create and use them.

3.3.3 Services

Services are the functional components of Haystack. Every function of Haystack, from data manipulation to user interface, is encapsulated in a service. We have abstracted our functionality away into services so that we have a general way for services to obtain pointers to other services, whether they are remote or local. This provides for easy integration of remote services into our model. As well, some services are event-driven, activated by changes made to the metadata.

The data manipulation services that this thesis describes in detail are a combination of event services and non-event services. The event services can be considered part of the core of Haystack since they produce the core metadata elements of a document. This includes fetching the body of a document once we have a location, guessing the type of a document once we have a location and a body, extracting descriptive fields such as title and author, and textifying once we have a body and a type. The non-event services act as a middle layer between the core services and the UI. They provide a means to start an archive of a document or to query Haystack.

The next three chapters discuss the design of the service model, the event model used by some services and the remote service communication model used by remote services. This discussion builds up to chapter 7 which explains in detail the core data manipulation services we need to complete our infrastructure.

Chapter 4

The Service Model

The functionality of Java version of Haystack is implemented as a set of distributed, data-driven services. These services primarily run in one Java VM, called the *root server*. However, they can also be run outside the root server in another VM or not in Java at all. All services have globally unique names that are registered with a name service. We chose this model so that it would be easy to plug in new services, and to modify old ones. The section on service names (4.1.1) gives more detail about how we encapsulate versioning information and other metainformation about the service into its name.

The services that make up Haystack's infrastructure can be broken up into the following categories:

- **Core Services:** The bottom layer of services that provide the infrastructure of the service model that other services need to run.

- **Data Model Services:** The services that maintain storage of and access to metadata.

- **Communications Services** The services that allow programs (Java or otherwise) running outside the root server to communicate with the Haystack services running inside the root server.

- **Data Manipulation Services:** The services that act on data; this can involve creation, modification, querying, and reporting; these services are generally event-driven or activated by a UI.

- **UI Services:** The services that provide a user interface to Haystack.

Figure 4-1 shows how these services interact with each other and with the user, and how they can be situated locally or remotely. The arrows represent requests. Most services inside the root server can talk to each other, while services external to the root server talk only to their respective interface services (remote and external communications services). The exceptions to note are that the external communications service must go through core services to call anything else and that the remote communications service is not allowed to talk directly to the data model.

4.1 Service Model Basics

Here we discuss the basic implementation details for services. This includes where services are located and how they are named. As well, we discuss the basic service class that all other services extend and what functionality it provides.

4.1.1 Service Naming

The set of services that come with Haystack are all located in the `haystack-.service` package. Table 4.1 explains the various sub-packages contained in the `haystack.service` package. Each of these packages will be discussed in this chapter and chapters 5 and 7.

All services are assigned a unique name. We determined four attributes that comprise a service name. The first attribute is the name of the class for the service itself. The second attribute is the package that the service resides in. The version and creator of the service round out the name. In the future, a name could contain other attributes that contain metainformation for the service, but only the four original attributes will determine service uniqueness.

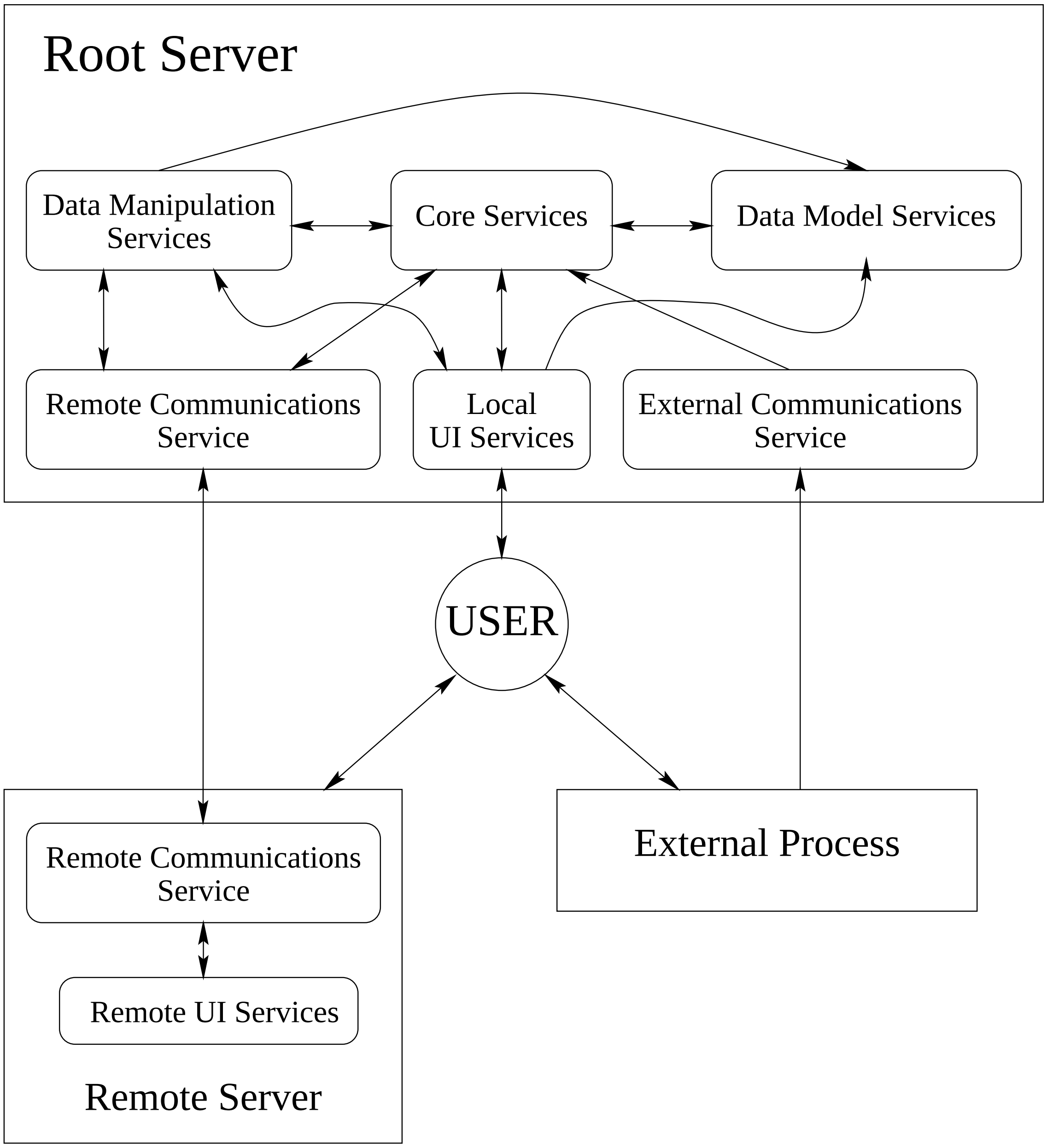


Figure 4-1: Service Model Interactions

Table 4.1: Sub-packages of `haystack.service`

Package	Function
`command`	API command objects
`events`	event model objects
`fetch`	fetch services
`fieldfind`	field finding services
`gui`	GUIs for core services
`ir`	IR engine services
`misc`	miscellaneous data abstractions used by services and miscellaneous abstract service classes
`textifier`	textification services

We have encapsulated the notion of a service name in the `ServiceName` class. Any service that is distributed with Haystack is assigned the creator *system*. We envision that services created by other users would bear that user's login name as the creator. The package for services created by other users can and should be something other than the default package for services that come with Haystack, `haystack.service`. The name and version must be set by the service. All services contain a static data member called *name* of type `ServiceName` that uniquely identifies them. We hope that service names can be extended to include metainformation that describes the service. This information can include measures of the service's trustworthiness or how fast it runs. Haystack can then use this metainformation when it has a choice between using two similar services.

4.1.2 Service Operation

Initialization for services occurs in two steps. The first step is the creation of the service, which corresponds to the call to the no-argument constructor. This generally registers the service with Haystack and assigns a name to the service. The second step in initialization is to call the `init()` method. This step usually loads any necessary information into the service and performs any additional registration so that it is ready to perform its assigned functions. The two-step process allows services to depend on one another in their initialization sequences, but not in their creation sequences. Once all services are created in the first step, any piece of code can reference them.[1]

All services extend `HsService`. `HsService` provides the core functionality for all services, including access to the service's name and the constructor which registers the service's name with Haystack (this registration is described below in section 4.2.2 about the name service). `HsService` also defines two methods called `init()` and `close()`. If a service needs to do something to initialize itself or cleanup after itself, it should override `init()` and `close()`, respectively. Both of these methods provide a means of reporting all error conditions through a single exception: `HsServiceInitException` and `HsServiceCloseException`, respectively. A particular extension of `HsServiceInitException` called `HsServiceCriticalInitException` indicates that there was a critical error during `init()` and that Haystack should abort. Some of these services will be described in this chapter. However, others will be explained in chapters 5 and 7.

4.2 Core Services

Core services provide the internals necessary for services to interact with each other and with the DM. Table 4.2 lists the various core services and their function in Haystack. `HsDispatcher` will be discussed with the event model in chapter 5. All other core services are described in detail in the following subsections.

4.2.1 The Root Server

The root server, which is implemented in the class `HaystackRootServer`, provides a global environment for other Haystack components to use. It maintains several variables that point to services that many other services need to use. Upon creation, it creates each of these services, setting the appropriate public data members. The

[1]Services are initialized in the order they were loaded. This can be important if one initialized service depends on another. In general, this dependence is frowned upon, but it may be necessary in isolated cases.

Table 4.2: Core Services and their Functions

Service	Function
`HaystackRootServer`	Maintains the global environment
`HsNameService`	Maintains pointers to all services
`HsConfigService`	Parses config file(s) and provides config values
`HsResourceControlService`	Maintains locks on named resources or DM objects
`HsCounter`	Maintains synchronized named counters
`HsCacheService`	Maintains a cache for needle contents
`LoggerService`	Maintains a log file
`HsDispatcher`	Dispatches events to services

root server may be created in one of two modes: *root* or *non-root*. In *root* mode, the root server loads all of the core services; whereas in *non-root* mode, the root server only loads those services that can be accessed remotely. Remote access to services is described in chapter 6. In either mode, though, it is the root server's responsibility to maintain the global environment.

4.2.2 The Name Service

The name service is the first service to be loaded. It is implemented in the class `HsNameService`. The name service maintains a mapping from `ServiceName` to `HsService` for every service that is registered. Registration, initiated by `HsService`, is actually done in the name service. It is also the name service's job to initialize and close services that are registered with it (the root server tells it when to do this).

4.2.3 The Config Service

The config service, `HsConfigService`, provides an abstraction for all configuration settings that services need. These settings include things like filenames to store data in, the location of the *HAYLOFT*, and the default archive options. The settings are stored in the preferences file. Currently, the config service stores these settings in a hash table format that has a string key and a vector of string values. The config service itself requires that some settings be present in the preferences file, namely `HAYLOFT` and `LoadRootService`. As you might suspect, `HAYLOFT` indicates the location of the *HAYLOFT*. `LoadRootService` is a list of services that should be loaded when the *root* Haystack root server is started.

Since the config service provides this special functionality, it has a special `init` method that takes the name of the preferences file as an argument. The call to the config service's `init` is special-cased in the root server and is performed before any other services are initialized. That way, any services that the preferences file indicates should be loaded are loaded before any service is initialized. Once this special `init` method is called, the root server tells the name service to initialize all loaded services.

4.2.4 Utility Services

The core services include many utility services that, for the most part, only rely on the core services discussed already, not on each other or any other services. Several other services all rely on many of these core utility services. For instance, `HsResourceControlService` provides a means to lock resources that many services use to synchronize actions. The resource control service can lock a resource named with a string, or it can lock a DM object.

The `LoggerService` abstract class defines a general service that logs status information to either the screen or a log file. Currently, three services extend this class and provide basic logging functionality for all services. These loggers are an error logger (`HsError`) for logging error messages, a message logger (`HsMessage`) for logging general system messages, and a debug logger (`HsDebug`) that will only print the message if the class generating the message has its debug flag set.

Finally, we have two utility services that are not used as widespread as the previous services, but are still essential. HsCounter provides named, synchronized counters for services to generate sequence numbers. This is used, for instance, to generate Haystack IDs for DM objects (see section 3.3.1). As well, we have `HsCache`, which provides caching for files. A promise can use the cache service to cache the fulfillment of promises (see section 3.3.2).

4.3 Data Model Services

To maintain the data model discussed in section 3.3.1, we have data model services. These data model services depend on the core services discussed previously, and provide the rest of Haystack with a means of creating and storing DM objects. Table 4.3 lists the various DM services and their function in Haystack. The DM services are described in detail in the following subsections.

Table 4.3: Data Model Services and their Functions

Service	Function
`HsPersistentObjectService`	Maintains a persistent storage for DM objects
`HsObjectCreatorService`	Provides a means of creating DM objects
`HsLore`	Saves DM objects in a database format for database-style querying

4.3.1 Persistent Storage

All DM objects are maintained persistently on a storage device. This storage is maintained by the service `HsPersistentObjectService`. DM objects register themselves with the service upon creation. Once a DM object is registered, it is the job of the persistent object service to maintain the DM object. All DM objects can be looked up using their Haystack ID. When an object is not found in memory, it is loaded from disk. When Haystack is shut down the persistent object service saves all loaded DM objects to storage. A `deleteObject` method provides a facility for deleting objects permanently from storage.

4.3.2 Object Creator

DM objects should not be instantiated directly (in fact, no DM object has a no-argument constructor). Instead, the service `HsObjectCreatorService` should be used. The `newObject` method of the object creator service takes one argument: a string representing the class name of the DM object you want to create. Generally, this string will be obtained from the class `StrawType`. This class maintains static strings for the straw types of DM objects (see *Naming* in section 3.3.1).

The object creator service will climb the object hierarchy starting at the type passed to it trying to find a valid DM object to instantiate. If Straw is reached in the object hierarchy, then the search stops and a Straw is created. Please refer to *Naming* in section 3.3.1 for a discussion of the naming hierarchy. Additionally, the object creator uses `HsCounter` to assign unique Haystack IDs to every DM object it creates.

4.3.3 External Database

DM objects are also stored in an external database format, if an appropriate database is running on the platform Haystack is being run on. Currently, we support the Lore database, which is an object-oriented database for storing unstructured information. See Adar's thesis [1] for a thorough description of the integration of Lore and the added query benefits gained from database-style queries.

4.4 Communications Services

We have two means for processes outside the root server to communicate with services inside the root server. The first means provides a way for services written for Haystack in Java that do not run in the root server to communicate with the root server. The communication mechanism is similar to Java's RMI and is described in detail in chapter 6. The second means of communication is through the service `HsCommunicator`. This service listens on a network port for commands. The commands must follow the format laid out by the `HsCommandAPI` service. Each API command is encapsulated by a command object located in `haystack.service.command`. To add a new command, a new extension of the command object must be created. The commands to load into the command API are specified in the preferences file.

For more a detailed description of this method of communication, see Adar's thesis [1]. Thus, to extend Haystack the user can choose to write a Haystack service that either runs in a Java VM outside the root VM (more efficient since we can still use the Java object representation) or communicates with Haystack over the network with a defined API (less efficient since the communicator must convert everything to an language-independent format).

4.5 Data Manipulation Services

Data manipulation services comprise the main subject of this thesis. These services are the next layer of services above the core services and data model services. Most data manipulation services are event services that follow the event model described in chapter 5. Others provide an entry-point into Haystack for the UI services to call.

The data manipulation services provide the core functionality that we need in Haystack from archiving and indexing a file to metadata extraction to querying data once it is indexed. The functions discussed in this thesis include

- **Archiving** The creation of a new document bale and retrieval of the body of the document for further processing
- **Metadata Extraction for Annotation** The extraction of metadata from the document describing key characteristics such as file type, title, author, date created, textual representation of the document, etc.
- **Document Collection Handling** The handling of collections of documents such as RMAIL files and directories.
- **Indexing** The actual indexing the IR system does on the text of the document to facilitate searching.

When data manipulation services act on the DM, we must be sure to use the resource control service to lock the DM objects prior to modification. The descriptions of the design and implementation of the data manipulation services can be found in chapter 7.

4.6 Other Services

There are other services that will be written (or are in the process of being written) for Haystack. Notably, user interface (UI) services are being written, and new ones will be written to provide convenient ways of interacting with Haystack. Currently we do have a command-line UI. A Java UI and a web server UI are being worked on. Other UIs could be Java applets, especially for functions such as editing user preferences. We also wish to have a web proxy service that records a user's web browsing behavior (potentially adding web pages to a user's Haystack).

Chapter 5

The Event Model

We have defined a set of events that characterize changes in the metadata stored by Haystack. These events drive many of the functional components of Haystack. The event model is based on the event model for GUI components included with Java [5]. However, we have modified it to suit Haystack's particular needs. The event model contains event sources and event listeners. Event sources are usually the core data services or objects in the DM itself that signal the changes in the metadata that are the causes of events. Event listeners are services that are interested in acting on the changes in the metadata (e.g. fetching a file once we have a location for it) that events encapsulate. We have created an event dispatching service to record which listeners are interested in which events. The event sources notify the dispatcher of a new event, and the dispatcher dispatches the new event to the appropriate services based on which services informed the dispatcher they were interested in the type of the new event.

Our event model differs from the Java event model mostly in that we have a centralized event dispatcher. In the Java event model, listeners are required to register themselves directly with the objects that generate events, so the model is more distributed. We want to retain control over event dispatching centrally, so that we can shut down all events easily and then restart them (see section 5.3.4). Also, the way we specify interests in events is more complicated than possible in the Java event model. This more complicated event interest will provide us with a powerful mechanism for detecting changes in subgraphs of the metadata.

Events in our model can be broken up into two categories. The first category we call Haystack events. These events are triggered by changes in a subgraph of metadata. Specifically, the subgraph a listener for a Haystack event can specify is a star graph. We explain star graphs in detail in section 5.2.1. An example of such a star graph is a document bale tied to both a body and a file type. When this structure is present, we can, for instance, extract type-dependent fields from the body of the document. When a link is formed between two straws, a new star graph is formed in the metadata which generates a Haystack create event. When a needle's data changes, and it is part of a star graph, a Haystack change event is generated for that star graph. From the previous example, if the body changes (i.e. the data of the body needle changes), then we would want to run the field extractor again to see if there are any new fields. Note that no Haystack event can be generated on a straw that is not tied to anything else. We do not believe the existence of a straw has any meaning unless it is connected to another straw (forming a relationship), and thus no services would be interested in acting on it.

Our second category of events was created to handle the exception to the rule just stated. Some services, which do not act on the data changes to modify the DM, but which record, report, or analyze all changes, need to be informed for every change to a straw, regardless of what structure, if any, the change affects. An example of this service is a database service that stores the straws for database-style queries (see section 6.4.2 for an example of this service). Thus, we have created a class of events called object events. There are four types of object events we can generate: the creation or deletion of a straw and the change of a needle's data or a tie's forward or back pointer.[1]

[1] Recall that if you change a tie's forward or back pointer, the corresponding link in the straw pointed at must be changed as well, so capturing changes in ties' pointers is enough to specify all link changes.

5.1 Events and Their Sources

All objects related to events reside in the `haystack.service.events` package. All events in Haystack are extensions of the base event class, `Event`. The `Event` class extends `java.util.EventObject` which is provided by Java as the base class in the Java event model. The event class stores the source of the event (always a straw) as well as the time of the event and an optional string providing extra information about the event. Figure 5-1 displays the hierarchy of classes that extend `Event` (note: abstract event classes are in dashed lines). We have two major types of events in Haystack. The first, `HaystackEvent`, represents a change to a subgraph of metadata. The second, `ObjectEvent`, represents a specific change to a straw, regardless of whether it affects the structure of a metadata subgraph or not. The following two subsections describe these two types of events and their subclasses in detail.

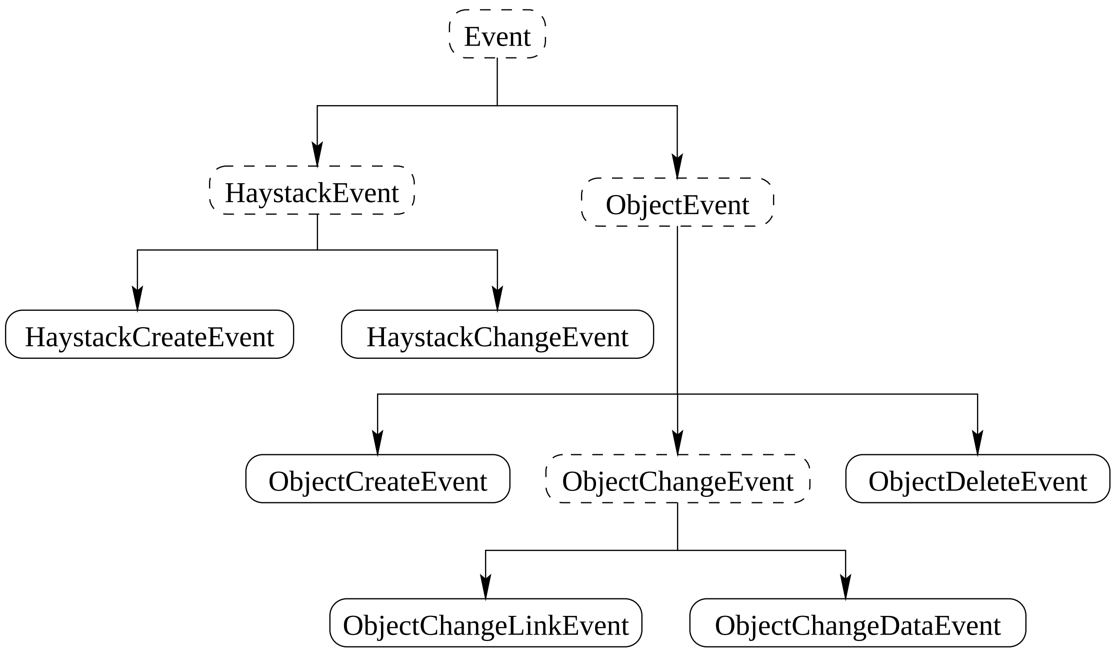


Figure 5-1: Event Object Hierarchy

5.1.1 HaystackEvent

A `HaystackEvent` indicates a change to some subgraph of our metadata. It cannot be generated by an unconnected straw but is generated when at least two straws are connected. There are two types of `HaystackEvents`: a create event (`HaystackCreateEvent`) and a change event (`HaystackChangeEvent`). Both types of events have two pieces of data associated with them: a source (inherited from the base event class) and a cause. The subgraph of metadata that is said to be affected by a Haystack event is the subgraph rooted at the source.

The `Tie` class generates create events when a tie connection is completed (i.e. when the forward and back pointers are set). Any change to the forward or back pointers while it is connected also generates a create event. A create event represents a new subgraph being formed in the metadata. The source for a create event is the straw pointed at by the back pointer of the newly connected tie, and the cause is the straw pointed at by the forward pointer.

A change event is produced when the data for a needle that is connected to at least one other straw changes. Change events cannot happen on needles that have not been either the cause or the source of a create event. Consequently, a change that happens to needle with no back pointers and no forward pointers does not generate a change event. Any change that happens before the create event will not matter until the needle has been *noticed* with a create event. The cause for a change event is always the needle that changed. However, the source can be either the needle that changed or any of the back straws from the needle.[2]

5.1.2 ObjectEvent

An `ObjectEvent` is generated for every change to a straw, including data changes and link changes. Object events have only one piece of data, the source inherited from the base event class. We have three types of `ObjectEvents`: a create

[2]In fact, when change events are generated from a needle, several different `HaystackChange-Events` are generated, but this is discussed later in section 5.3.

event (`ObjectCreateEvent`), a delete event (`ObjectDeleteEvent`), a change event (`ObjectChangeEvent`). Create events and delete events are pretty straightforward. The persistent object service generates a create event when a straw is first registered with it, and it generates a delete event when a straw is unregistered from it. The source for both events is the straw being created or deleted.

Two classes extend `ObjectChangeEvent` which is abstract. The `ObjectChange-DataEvent` class represents a change to the data of a needle. It is generated by `Needle` every time `setData(Object)` is called. The source for this event is the needle whose data changed. The `ObjectChangeLinkEvent` class represents a change in the link structure between two straws. This event is generated by a `Tie` in the same situation as a `HaystackCreateEvent`. The source, however, is the tie itself.

5.2 Event Listeners

Services need to specify their interest in particular events. A service can be interested in either a haystack event or an object event. Following the Java event model, we have defined an interface, `Listener` that extends `java.util.EventListener` to represent an interested service. This interface declares a `getName()` method which returns the name of the interested service. To describe each of the two types of events in our event model, we extended `Listener` with `HaystackListener` and `ObjectListener`. These two listener interfaces declare the methods used to handle an event sent to an interested service (e.g. `handleHaystackCreateEvent(Hay-stackCreateEvent)` for a Haystack create event).

To specify an interest in a Haystack event, a service must specify the straw types for the straws of the subgraph the event was generated for. We have decided to limit the subgraphs that services can be interested in to star graphs, that is, a graph rooted at a straw with ties pointing out one level deep to other straws. We can efficiently check for this kind of subgraph being satisfied, but anything more complicated would not be worth the extra overhead. This type of interest is encapsulated in the `StarGraph` class described below in section 5.2.1. Services interested in object events are assumed to be interested in all object events, so they cannot specify a particular straw they are interested in. It is possible, in the future, that we could come up with listeners that would be interested in object events only on certain straw types. This functionality would be easy to add to the current means of dispatching object events. Interests are registered with the event dispatcher which is described below in section 5.3.[3]

5.2.1 Star Graphs

The class StarGraph represents the type of a subgraph in the metadata that forms a star graph. We use it as the type of event sources that a HaystackListener is interested in. The star graph is constructed by specifying a straw type to be the root. It provides a method for adding rays to the star in the form of tie/straw (type) pairs. Note that when specifying the type of a tie and the type of a straw for this pair, we are implicitly including all supertypes of these types. For this reason, we limit the graph to a star graph. Otherwise, we would have a lot of overhead to check when the type of a subgraph matches a particular subgraph in the metadata, and we cannot afford this overhead since these checks are run almost every time a change is made in the metadata.

Internally, the star graph is specified as the straw type of the root together with patterns representing the straw types of the rays of the star. We use the pattern matcher package from Original Reusable Objects, Inc. [19]. Each pattern for a ray in the star graph is represented as the regular expression /<Tie[^:]*:Straw[^>]*>/i, where *Tie* is the straw type of the tie and *Straw* is the straw type of the straw for a ray. This allows us to convert any ray from a metadata star graph into a string form that we can compare against the patterns of the star graph rays. Since we have included the "*" at the end of the straw types for the tie and the straw in the patterns, the comparison will match any subtypes of those straw types in the

[3]For convenience, we have defined two abstract classes, HaystackEventService and Ob-ject-EventSer-vice, that implement the two listener interfaces and extend HsService. They do not actually define the listener interface, but they do provide easy methods for registering an interest so services do not need to know how to use the event dispatcher.

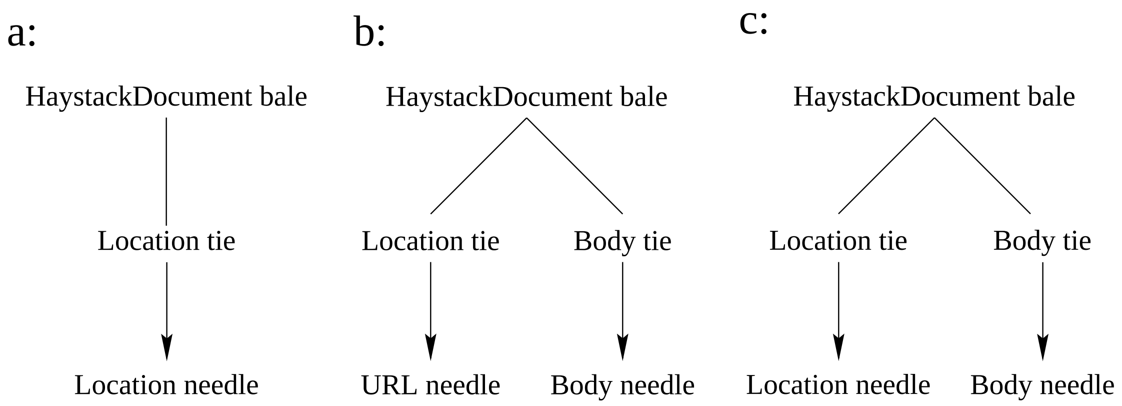
metadata being compared.[4] Figure 5-2 gives some examples of star graphs. These star graphs can be interpreted as `StarGraph` objects or as the straw types of the nodes of a star graph contained in the metadata. The figure shows how `StarGraph` would convert the star graph interpretation into regular expressions, and how it would convert the metadata star graph into a string input to be compared against the patterns of a `StarGraph` object.

A star graph provides two methods for comparing itself to a subgraph of metadata rooted at a certain straw. The first method, `boolean contains(TieWeavable, Weavable)`, checks to see if the ray specified by the tie/straw pair exists in the star graph. The second method, `boolean isContainedIn(Weavable)`, checks to see if the stargraph of metadata rooted at the straw argument matches types with the star graph object. It does this by checking first to see if the type of the straw argument matches the type of the root of the star graph and second, if the straw type of every ray of the star graph matches the straw type of the rays of the star graph rooted at the straw argument. The event dispatcher uses these two methods to decide which services receive a particular Haystack event. In Table 5.1, we can see how the `isContainedIn` method will work on the examples from Figure 5-2. Note that the graphs in Figure 5-2 can be considered to be either star graphs or the straw types from a metadata subgraph; we use them interchangeably in table 5.1.

Table 5.1: `isContainedIn` Examples

Method Call	Result (true or false)
`a.isContainedIn(b)`	true
`a.isContainedIn(c)`	true
`b.isContainedIn(a)`	false
`b.isContainedIn(c)`	false
`c.isContainedIn(a)`	false
`c.isContainedIn(b)`	true

[4] This regular expression syntax comes from Perl 5 [22]; note that it is case-insensitive so that `haystack.object.tie.Location` would match `haystack.object.Tie[^:]*`. Refer to *Naming* in section 3.3.1 to review how we name objects in the DM.



StarGraph patterns for a:
/<haystack.object.tie.Location[^:]*:haystack.object.needle.Location[^>]*>/i

Input to pattern for a:
"<haystack.object.tie.Location:haystack.object.needle.Location>"

StarGraph patterns for b:
/<haystack.object.tie.Location[^:]*:haystack.object.needle.location.URL[^>]*>/i
/<haystack.object.tie.Body[^:]*:haystack.object.needle.Body[^>]*>/i

Input to pattern for b:
"<haystack.object.tie.Location:haystack.object.needle.location.URL>
<haystack.object.tie.Body:haystack.object.needle.Body>"

StarGraph patterns for c:
/<haystack.object.tie.Location[^:]*:haystack.object.needle.Location[^>]*>/i
/<haystack.object.tie.Body[^:]*:haystack.object.needle.Body[^>]*>/i

Input to pattern for c:
"<haystack.object.tie.Location:haystack.object.needle.Location>
<haystack.object.tie.Body:haystack.object.needle.Body>"

Figure 5-2: StarGraph Examples

5.3 The Event Dispatcher

At the heart of the event model is the event dispatcher in the class `HsDispatcher`. This is the one piece of the event model that does not reside in `haystack.service.events`. Instead it resides in `haystack.service` since core elements depend on it (to generate the events).

The dispatcher provides two methods for a listener to register an interest. The first, `registerHaystackEventInterest(HaystackListener, StarGraph)`, registers a listener for any Haystack event whose source contains the star graph. The `registerObjectEventInterest(ObjectListener)` method registers a listener for any object event, regardless of the source. To signal events, the event dispatcher provides a method for each non-abstract type of event we have (see Figure 5-1). The format for the names of these methods is the word *inform* followed by the name of the event (e.g. `informHaystackCreateEvent` for a `HaystackCreateEvent`). These methods are described in the following two subsections.

5.3.1 Generating Haystack Events

To generate a create event, the dispatcher is informed of a tie that has just been fully connected. The inform method for this event checks to see if any services are interested in a star graph rooted at the back pointer of the tie. If so, then it takes the list of potential services and checks their star graph interests entirely using the `isContainedIn` method as described in section 5.2.1. If there are services to inform, the dispatcher packages packages up the forward and back pointers of the tie into a Haystack create event and dispatches to the appropriate services.

To generate a change event, the dispatcher is informed of a needle that has just had its data changed. The dispatcher first checks to see if any services are interested in a star graph rooted at the needle. If there are any, it performs the same step as with the Haystack create event in fully checking each star graph and then dispatching the change event to the appropriate services. In this case the source and the cause of the event are both the needle whose data changed. The dispatcher then checks to see if any services are interested in a star graph rooted at any object pointed at by a back link of the needle. If any are, it checks their star graphs completely and dispatches to the appropriate services, setting the source of the event to be the straw from the back pointer and the cause to be the needle.

5.3.2 Generating Object Events

The inform methods for the four object events are quite simple. They check to see if any services have registered as being interested in object events and dispatch the new event to them. The source for the create and delete events is the straw being created or deleted. The source for the change data event is the needle whose data changed and the source for the change link event is the tie whose back and forward pointers were connected.

5.3.3 A Service Threader

When the dispatcher has determined a list of listeners and a particular event object to send to them, it passes the actual work of calling the handling methods in each listener to the `ServiceThreader` class. The service threader is an inner class of `HsDispatcher`. It implements the Runnable interface so that it can be run inside a Java thread. This allows the call to the inform method to return without having to wait for the event to be dispatched to all interested services. The dispatcher keeps track of the service threaders it has running and the threads in which they are running so that it can control how many concurrent threads are running at the same time. Currently, we do not limit the number of concurrent threads, but it would be easy to impose such a limit, queuing up signals for new events and dequeuing them one at a time as threads finish.

When a service threader is created, it is passed a list of listeners to inform and the event object to inform them about. The service threader operates in a separate thread, looping through each listener in the list of listeners passed to it, and calls the appropriate handler depending on the type of event the service threader is for.

Each time through the loop, it checks to see if a request has been made to stop the service. If it discovers this request, it will stop informing listeners, and remove the listeners it has already informed from its list. This allows us to suspend the operation of Haystack at any point, allowing currently-running services to proceed, but allowing no further informing of events to occur.

5.3.4 Saving State

The event dispatcher is the first service closed when Haystack is shut down. When the event dispatcher is closed, it needs to do several things to stop the processing of events and to save the state of events so they can be restarted the next time Haystack is turned on. First, it sets an internal variable to indicate it is going down. When the variable is set, the dispatcher will not put new service threaders that are created into threads to run. Instead, it just keeps them in the vector of service threaders with the other running service threaders. Next, it joins each of the running threads. As soon as all running threads finish (which will be after the current services they are informing return), the list of service threaders is written to disk.

Upon initialization, the dispatcher service will load up any service threaders that were stored to disk and send them off to the `StartThreader` class. This class waits for the root server to finish initializing to ensure that all services have been initialized and then starts each service threader that was loaded from disk. Figure 5-3 shows two examples of the event model in action with the two basic types of events. At any point during the processing of the listeners 1 through n, the service threader can be stopped and its state saved.

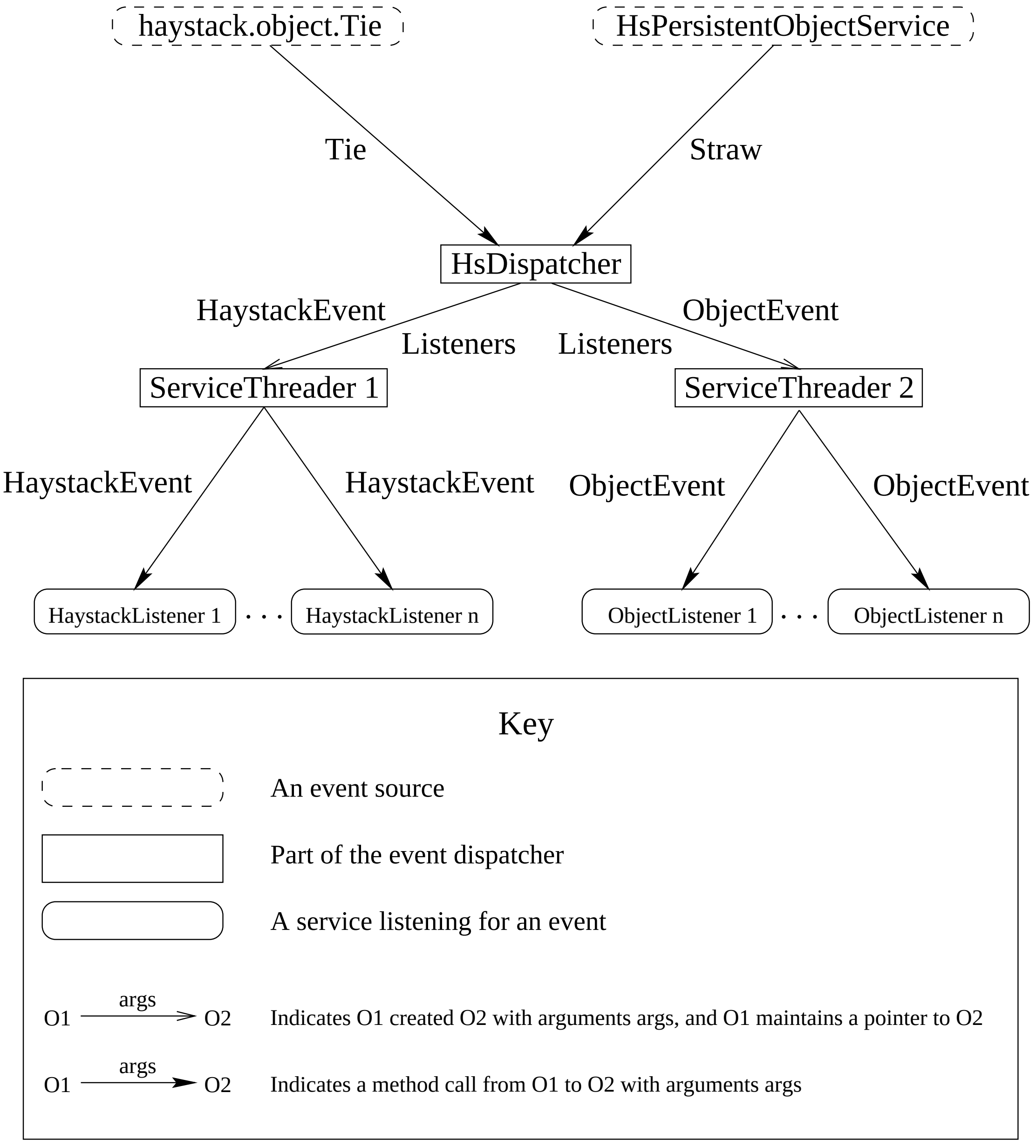


Figure 5-3: Event Model Example

Chapter 6

The Remote Service Communication Model

This chapter describes the design of the communication mechanism between Haystack services in two different VMs. When a method invocation is desired from a service not located in the current VM, the invocation request is passed to the remote communication mechanism to communicate the invocation request to the service in another VM. The benefits of using this approach is that all services communicate with each other as if they were all in the same VM (i.e. through method invocations), even if they are not. Given the distributed nature of Haystack and the ability to add arbitrary remote services, we do not want services to need to worry about handling the remote communications themselves. This mode of communication is similar to the mode implemented by Java RMI (see section 2.4.2).

At the time we were making the decision as to what remote communication model to use, Java's RMI was quite new and we were not comfortable using it. We were told that the current implementation was slow, and that it may no longer be supported. This was not the only reason, however, that we chose not to use Java's RMI . First, Java's RMI provides a facility for remote references of arguments passed to methods. We do not want applications to be able to remotely call methods in a DM object as we do not currently want to deal with the transaction processing that would require. As well, we did not want applications to have to know whether they were communicating over the network or not. Java's RMI requires non-runtime exceptions to be caught on remote methods. In fact, objects that can be made remote in Java are usually not also made local. They are intended to be server-side programs. Our model allows a service to export some of its method to become a remote object, while still being used by local services as well. We use runtime exceptions so an application can decide whether it needs to handle network exceptions or not.

Furthermore, to write a remote object in Java requires extending a particular class which would break our service model given that all services must extend `HsService` and not all services export their methods.[1]. The default naming system that comes with Java RMI was not compatible with the naming scheme we desired. We did not find enough documentation on the implementation of the naming scheme to implement one of our own that would work with Java's RMI. Ultimately, the reason we chose to implement our own remote communication model is so that we could retain control over what is allowed and what is not allowed in our communication model and so that we know our model will be supported in the future (since we define it).

The remote communication mechanism is located in the `haystack.communication` package. The classes `Middleman` and `MiddlemanServer` provide an abstraction for network communication between the *root* Haystack VM and all other VMs, establishing a protocol to communicate across the network.[2] Services can export any of their methods to be called remotely. This export process is described in the section the next section. The rest of this chapter describes the communication infrastructure that allows remote communication between two services.

[1] Java does not have multiple inheritance, otherwise we could have those objects that export methods inherit from both `HsService` and Java RMI's remote service class.

[2] Both the `Middleman` and `MiddlemanServer` implement an interface called `Sendable` which defines the necessary methods for communication; we will use the term *sendable* when referring to something that requires either a middleman or a middleman server.

6.1 Real and Virtual Services

Every service that exports methods will define an interface containing those methods. The service will then have both a real and a virtual implementation that implement this interface. The interface will be used for all references to the service when only the methods declared remote are desired. The real service will implement the functionality described for each method of the service's interface, as well as additional methods required by the implementation. The virtual service, however, is only a stub that packages up the method call into a packet (described below in section 6.2) and passes the packet to the sendable for the VM to send to the corresponding real service in another VM. The return value will be unwrapped from the packet it gets back as a result from the sendable and returned to the method caller.

We chose to use a common interface for the real and virtual services instead of the alternative which would have been to make the virtual service an extension real service. Our choice is better than the alternative. The benefit is that any data members in the real service are not allocated for the virtual service. The interfaces also provide a nice compact way of describing the API to a programmer.

Each service must register with a local, unique (i.e. one per VM) name service. When one service wants to communicate with another service, it requests a pointer to the desired service from the name service. It is the name service's job to ensure that either the service exists in the current VM, or if it does not, to create a virtual service in the VM and return a pointer to that. The name service is described in section 4.2.2.

6.2 Packets

A packet is the unit of information that is exchanged between virtual services and sendables, as well as between a middleman and a middleman server. The class `Packet` is an abstract class that implements `Serializable` but does not contain the actual data for a packet. It does, however, provide a message ID for each packet so that we can have unique identifiers which travel along with unrelated packets and can be used to identify related packets. Thus, related packets will have the same message ID, but unrelated packets will have different message IDs. Since `Packet` is a Java abstract class, it cannot be instantiated. There are three specific packets that we will be sending between sendables: commands, responses, and messages. Each of these three data are encapsulated by the classes `Command`, `Response`, and `Message` which all extend `Packet`.

6.2.1 Commands

A command packet contains a the name of the service to process the command, a method name to call in the service, and an array of arguments (of type `Object`) to pass to the method, as well as a message ID.

6.2.2 Responses

A response packet is created to encapsulate the return value of a command. It contains the message ID that was in the command packet as well as the return value of the command. It also contains a boolean data member called `error` that indicates if the value is actually a return value or an object describing the error. This error object can either be a string that holds a text description of the error or it can be an extension of the `Exception` class, indicating an exception was thrown while processing the command

6.2.3 Messages

A message packet contains a protocol message sent from one VM to another. Generally they contain string messages, but could contain other objects as needed (the `contents` field is defined to be an `Object`). Table 6.1 shows the different types of messages currently supported.

Table 6.1: Message Packets

Message Name	Contents	Purpose
`HAY`	none	greeting from a middleman to a middleman server
`HAYID`	`int`	assigns an integer ID to a middleman
`HAYREG`	`ServiceName` and `int`	registers a service on a specific channel (labelled with the `int` ID) with the middleman server
`HAYBYE`	none	indicates the sender is shutting down

6.3 Network Communication Overview

The `Sendable` interface defines what services can do to communicate over the network. Both `Middleman` and `MiddlemanServer` implement `Sendable`. Only one middleman server runs in the user's root VM. All other VMs for the user run a middleman. All middlemen connect to the middleman server, but not to each other. We foresee most communication to take place inside the Haystack root VM, and the rest to be commands being sent from other VMs to the Haystack root VM.

We do support other modes of communication, including sending a command from the root VM to another VM and sending commands between two non-root VMs. If a command must travel from the root VM to a non-root VM, the real service in the non-root VM must first register itself with the middleman server in the root VM (`Middleman` provides a method `registerService` which does this). If a middleman wants to send a command to another Middleman it must relay it through the middleman server. We do not see a real need for this kind of communication in Haystack, but we are making it available for future use. Each connection between a middleman and the middleman server is assigned a distinct integer so that the middleman server can distinguish between its many connections. Every connection is handled on both sides by a connection handler (described below in section 6.3.1).

We have not implemented security in our remote communication model yet, but it is something that will need to be added before we can risk using it in real situations. Right now, if an adversary knew the port we were listening to and the format for passing commands back and forth, he could cause our Haystack to do malicious things. For example, we could be told to archive a malicious Postscript file that contained commands to erase our hard drive. Adding security would not be difficult. It would involve using public key encryption to establish a shared private session key.

6.3.1 Connection Handler

The class `ConnectionHandler` implements the Runnable interface so that it can be run in a thread. It actually does the work of reading and writing packets between two sendables. Upon creation, it is passed one open connection (a Java `Socket` object) that a sendable has established to another sendable. It can then read and write packets to and from a connection handler for another sendable. The connection handler provides a synchronized method for sending a packet on the connection (we only want one packet in transit at a time so multiple packets do not interfere with each other).

When the thread containing the connection handler is started, the handler sits in a loop reading packets off of the socket connection. When the thread is finally stopped, the connection handler sends a `HAYBYE` message to the sendable it's connected to, and then all local shutdown activities are performed. This is the first way the thread can be terminated (there is one more way described below along with the local shutdown activities in *Handling Incoming Messages* in this section. The job of this handler can thus be split into four subtasks which are described in detail in the sections below.

Sending Commands

The first subtask is to process commands that are passed to it from its parent Sendable (via the `sendCommand` method). This subtask must record the message ID of the command packet and map this id to a `PipedOutputStream`.[3] The thread that

[3]Piped streams are used in Java for inter-thread communication, one thread reads from a piped input stream while another thread writes to a piped output stream that is connected to the piped input stream.

invoked the `sendCommand` method then blocks on the piped input stream which is connected to this piped output stream until it reads the response to the command. The connection handler thread is responsible for reading the response packet off the connection when it comes in and finding which piped output stream to write the response to.

Handling Incoming Commands

Incoming command packets each cause a new thread to be started which will actually invoke the method call encapsulated by the command packet. The `CommandProcessor` class is run in each of those threads. It is given a reference to the connection handler that started it, and the command it is supposed to call. It looks up the service in the local name service, and uses Java's introspection facilities to call the method in the service with the arguments that are in the command packet. It then calls the send routine in the parent connection handler to send the return value back to the originating VM in the form of a response packet.

Handling Incoming Responses

Incoming response packets are handled very easily. The message ID in the response is looked up in the internal table to find which piped output stream to write the response to. The response is then written to this stream.

Handling Incoming Messages

The only incoming message packet we have defined so far for a connection handler is a message packet containing the string `Message.HAYBYE`, which indicates that the thread should halt. Before it halts, it closes each command processor thread that is running and sends an error Response to the list of piped output streams it has. This is the second (and last) way the connection handler thread could be terminated.

6.3.2 Sendable

The `Sendable` interface defines three methods which both the middleman and middleman server must implement. The first is `getNewMsgID()` which returns a String that is a unique message id for the VM. This is used by the Packet class to get unique message IDs (it is defined in sendable so that the unique integer ID of the Sendable may be included to make the message id unique across VMs).

The second method is `sendCommand(Command)`, which takes a command packet and sends it across the network to the appropriate sendable. In both the Middleman and the MiddlemanServer, this method does the following. It starts by creating a `PipedInputStream` and a `PipedOutputStream`, connecting the `PipedOutputStream` to the `PipedInputStream`. It then passes the command together with the `PipedOutputStream` to the appropriate connection handler object which is handling the connection the command needs to go out on. It then blocks, reading from the `PipedInputStream`, until it reads the response packet for the command. It then returns that response to the virtual service. See the description of the connection handler to see how messages are sent and received between a middleman and a middleman server.

Finally, the `removeHandler` method, removes the pointer to the connection handler for the sendable. This is the first task in the local shutdown activities for a connection handler to ensure that no other commands are started up while we are shutting down. Also, in the middleman server, this method has the additional behavior of unregistering all services that had been registered using a `HAYREG` message packet.

6.3.3 Middleman

When the constructor for `Middleman` is called, the new middleman establishes a connection with the middleman server. It then constructs a new `ConnectionHandler` and passes off the connection to it. Each middleman has only one such handler as all communication is directed through the middleman server. Therefore, every call to `sendCommand` is passed on to the handler. This connection is maintained for the entire session until the middleman is closed. We keep the connection open so we do not incur the connect overhead each time we make a remote request. A `registerService` method allows services in a non-root VM to register with the root VM. This allows them to be called from the middleman server in the root VM or another VM (in which case the call is relayed through the middleman server). Finally, a `stop` method allows the middleman to be stopped from sending any more commands; this method will also stop any commands currently being processed.

6.3.4 Middleman Server

When a `MiddlemanServer` is constructed, it opens up a socket either on port 9643 (by default) or on a specified port. It implements `Runnable`, so that it can be placed in a thread and run as a server. The server listens for packets on the socket. The packets must be message packets and can be one of two different messages: `HAY` and `HAYREG`. `HAY` indicates that a middleman wants to establish a new connection, so the middleman server will spawn off a new connection handler and pass it the new socket connection. `HAYREG` indicates the middleman in the remote VM wants to register a remote service with the root VM. The service is added into the name service for the root VM and added to the list of services which the middleman server can forward remote calls to.

Commands are forwarded with the `sendCommand` method just as they are with a middleman. The difference is that the middleman server must decide which connection handler to send the command to. This is determined by an internal mapping of service name to connection handlers that the middleman server maintains. The server also catches the `ThreadDeath` exception, so that when the server is stopped, it sends a `HAYBYE` message out over every connection handler telling all the other middlemen to stop as well (if the root VM is going down, then all child VMs should too).

6.4 Examples

All services in Haystack do not export methods. Certain core services and most event services only run in the root VM and are not intended to be called outside of it. Here we present two examples of services which use the remote communication model. The first service, `HsError`, is intended to be run inside the root VM with the possibility of it being used outside the root VM as well. The second service, `HsLore`, is an unusual service in that it is intended to be run outside the root VM and accessed via a virtual service from inside the root VM. Figure 6-1 shows how 3 VMs could be set up with both examples shown.[4]

6.4.1 HsError

`HsError` is an extension of `LoggerService` which is an abstract service that writes messages to the screen or to a log file. VM 1 in Figure 6-1 contains the root `HaystackRootServer` and the real instance of the error logging service. VM 2 contains some UI service that may want to log an error and the virtual instance of HsError (gotten by the UI service by a request to the local name service). It invokes the method in the virtual `HsError` which wraps up the invocation request into a command packet and passes it off to middleman 1. The middleman passes the command packet along with a newly created piped output stream to connection handler 3 which proceeds to send the packet to connection handler 1 in the root VM which is handling the connection to VM 2. The connection handler spawns command processor 1 to process the command packet. The command processor unpackages the command, and invokes the requested method in the real instance of `HsError`. When it gets the return value back, the command processor sends it back, in the form of a response packet, to VM 2 through the connection handlers. Connection handler,

[4]This figure shows two out of the three possible communication modes: MiddlemanServer to Middleman and Middleman to MiddlemanServer. The third mode, Middleman to Middleman, can be though of as just the composition of Middleman to MiddlemanServer and MiddlemanServer to Middleman. Also, do not try to understand this figure entirely without reading the descriptions of each examples.

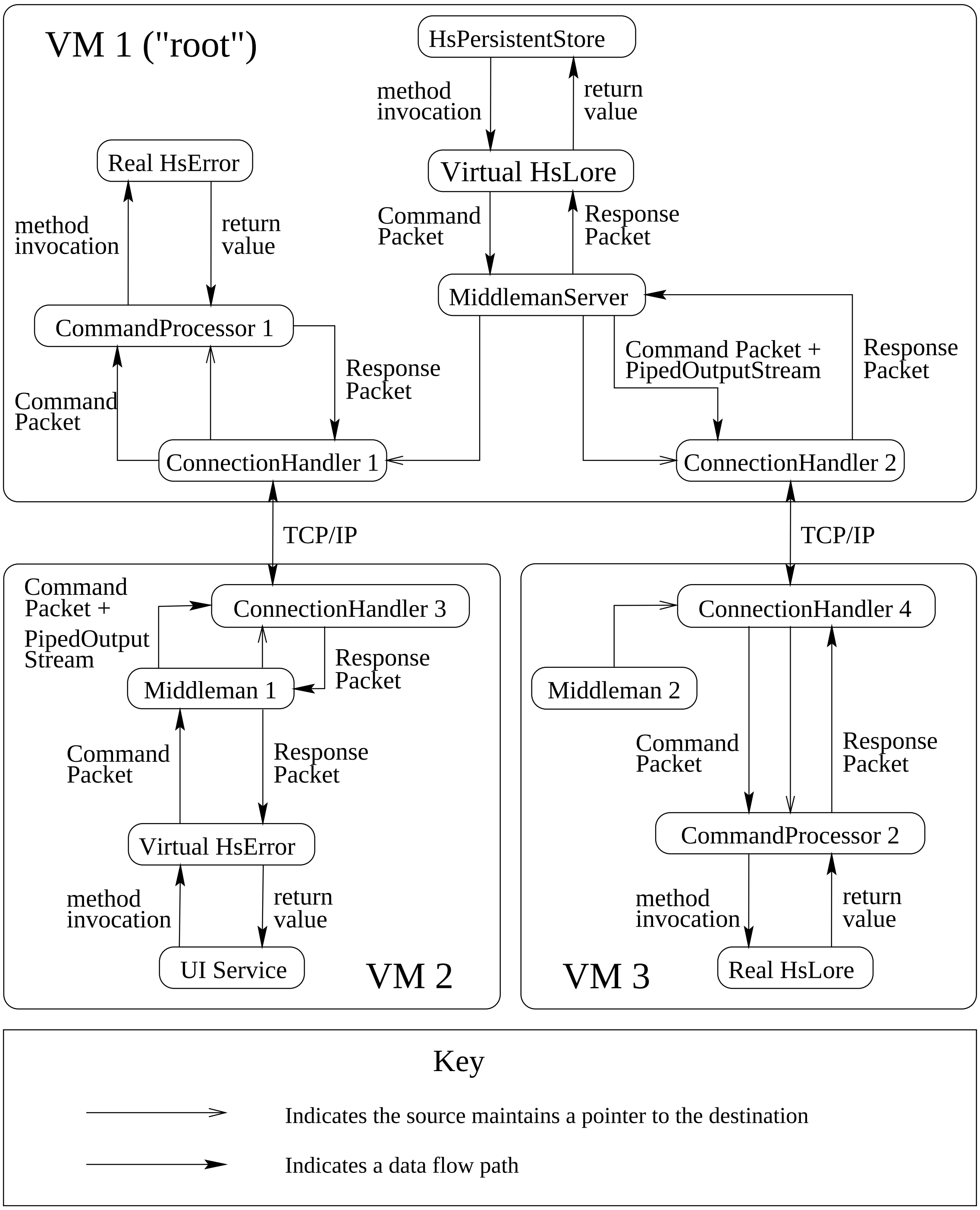


Figure 6-1: External Communication Examples

upon receiving the response and decoding which piped output stream to write it to, writes the response packet to the selected piped output stream that middleman 1 is blocking on. This will cause the middleman to return the result packet back to the UI service in the form of a return value (or exception if one was thrown by the real `HsError` in VM 1).

6.4.2 HsLore

The Lore service, `HsLore` provides a way to store the metadata in an object-oriented database. However, since Lore only runs on Solaris machines, it may need to run as an external process if the user's main machine is not a Solaris machine. This is an unusual case, but one we definitely want to be able to handle. The virtual `HsLore` runs in the root VM, while the real `HsLore` runs in VM 3. Method invocation requests would originate in the root server, say, by the persistent store service in this instance. The virtual `HsLore` packages up the request into a command packet and passes it off the the middleman server. Unlike in the previous example where the middleman only had one connection handler to chose from, the middleman server must decide which connection handler to pass the command packet to. It does this by looking up the service name from the command packet in an internal table that maps service names to connection handlers.[5]. Once the connection handler is chosen, the request is passed along just as in the previous example but with the middleman and middleman server reversing roles.

[5]The real `HsLore` in VM 3 must have registered itself with the root VM using middleman 2 in order for this mapping to have been created

Chapter 7

Data Manipulation Services

In this chapter we discuss the design and implementation of several of the core services that act on metadata in the Haystack IR system. We call these services data manipulation services. Many of the data manipulation services are event services (specifically interested in Haystack events) that form the core of the archive and indexing process, though some are simple services that act as a layer of abstraction between the user interface and the data model. Data manipulation services perform all the steps necessary to fetch, annotate and index a document. They rely on the core DM services and, in the case of event services, can be activated by changes to the metadata made by other services or, in the case of non-event services, can be called by user interface elements (through method invocations). All of these services are run inside the root server, though some export methods to be used outside the root server, as well. The following sections, which describe the various data manipulation services, are laid out in the order a document would pass through the services they describe from archive to index.

7.1 Archiving

The first step Haystack performs on any document added to it is to archive it. The various steps of archiving are described in this section. They include the addition of a location into the metadata, the retrieval of the body of the document from the location and the decision of whether to proceed with the archive based on whether the document has been archived before. The first service, the archiver that adds the location into the metadata, is not a data-driven action like most of our data-manipulation services. In fact, it provides an abstraction barrier between the UI and our data model to allow the user to add new information into Haystack. Appropriately, there is a set of options that is presented to the user through the UI that the archiver must deal with. We present these options first before going over the steps of archiving.

7.1.1 Archive Options

Table 7.1 lists the options made available to the user at the time of archive. Most of them are straightforward. The only complex option is the matching option. Some of the straightforward options might seem unneeded, but they become important if you consider archiving a directory. The options given to the directory are passed on to the files in the directory that are caused to be archived (see section 7.3.1). Thus, while the options may not matter for the document the user selected to archive, they may matter to documents that are added to the system as a result of the original document.

Table 7.1: Archive Options

Option	Meaning
Verbose	Indicates whether or not the user wishes an interactive session; if so, no default behaviors are used.
Save	Indicates whether or not the user wishes Haystack to save an archive copy of the document.
Empty	Indicates whether or not Haystack should process empty files.
Dotfile	Indicates whether or not Haystack should process files beginning with a dot.
Softlink	Indicates whether or not Haystack should follow
Match	Indicates what action to take if either the location or the contents of a document or both matches a document already in Haystack.

The matching option is specified by several parameters that determine how Haystack handles duplicate documents. We use two characteristics of a document to check for duplicates: location match and checksum match. This leads to four possibilities of matching status. As well, since we get the location match status before the checksum match status (we cannot get the checksum without the location), we allow the user to specify whether or not to proceed when we get a location match, so that makes a total of five parameters to specify the match option. If both the location and checksum do not match, then we know we have a new document. If both the location and checksum match, then we know we have an exact duplicate. The options made available to the user for this case are aborting the archive or creating a brand new document, separate from the old document. Aborting would make the most sense in this case since there should be no new information generated from an exact copy.

The difficult cases arise when the location and checksum disagree on whether a document has already been archived. In the case where the location does not match and the checksum does, we could be dealing with a copy of the document, or the document could have moved. The options available to the user are to create a new document, merge the new location into the document whose checksum matched, or supercede the old location with a new location. In this case, the option that makes the most sense depends on whether the file has been copied or moved. If the file has been copied, merging would make the most sense, but if it has moved, then superceding makes the most sense. A UI to Haystack might allow the user to specify when files have been copied or moved, so when that UI calls the archiver, it will know what to set this parameter to. The final case is when the location matches but the checksum does not. In this case, the content of the document has changed. The options available to the user for this case are to create a new document or to supercede the body of the document whose location matched with the new body. In this case, the option that makes the most sense is to supercede the body.

We have encapsulated the archive options into a class, `ArchiveOptions` located in `haystack.object`. `ArchiveOptions` gets default values from the config service, but those defaults can be overridden. One of the constructors for `ArchiveOptions` takes a hash table mapping option name to the value. Any options not specified in this hash table are loaded from the config service. The `ArchiveOptions` class implements `Serializable`, so it can be written to disk, and saved for each document we archive. We do not store the archive options with the metadata for a document since they do not describe the contents of the document but instead provide control information.

7.1.2 Archiver

The first step in archiving is to tell Haystack what location to examine. The service `HsArchive` provides this functionality as an archiver. It provides several overloaded instances of the `archive` method to allow different means of specifying this location (and any additional information that might be known prior to archiving). Recall that in *Needles* in section 3.3.1 we introduced location needles. We will now discuss the types of location needles we currently have defined in Haystack. A URL location specifies just a URL to retrieve a document from. We have an RMAIL location that specifies a particular RMAIL message inside an URL location that is an RMAIL file (see section 7.3.2 for a further discussion of how this location is created and used). Finally, the directory location needle provides a way of specifying the location of a directory.[1] In the future we will define a location needle for a query to specify the location of a *query document*, though this location will be added to Haystack by the query service, not the archiver (see section7.3.3).

In all cases, the archiver always creates the `HaystackDocument` bale for the document being added to the system. Once a location needle has been created (if it wasn't provided directly), the archiver checks it against the database of previously archived locations. If a match is found, a check is made with the archive options object to see if we can proceed. If we cannot proceed, the archiver aborts with an exception. Otherwise, it uses the location needle that matched.[2] The most simple

[1] A directory location needle actually holds the same kind of information as a URL location needle; however, they are conceptually different as one points to a single document and the other points to a collection of documents, so we give them each their own location needles.

[2] We only allow one location needle per unique location to exist in Haystack. If two documents

instance of the `archive` method in `HsArchive` just takes a string, which is assumed to be a URL, and converts it to proper URL syntax; it then creates a location needle for the URL and proceeds with the archive. Other forms of the archive method take archive options or an already created location needle, as well as a vector of other needles to attach to the document bale once it is created. The `archive` method returns the ID of the newly created document bale (or throws an `ArchiveException` if something goes wrong).

7.1.3 Fetcher

Once we have a location attached to a document's bale, we will need to fetch the body of a document. This is the job of a fetcher service. Each type of location has its own fetcher defined. The fetcher service is an event service that is triggered by the star graph consisting of a document's bale tied to a location needle.

We have defined an abstract fetch service in the class `haystack.service.-fetch.FetchService`. This class provides the base features of a fetch service but defines an abstract method for generating the data of the body needle as well as a protected vector (`docTypes`) for specifying the straw types of the location needles an implementation of this abstract fetch service would be interested in fetching. The `init` method of the fetch service registers interests with the dispatcher for each straw type specified in `docTypes`. The star graphs it registers are rooted at a `HaystackDocument` bale and contain a single ray consisting of a location tie pointing to one of the location needle types specified in `docTypes`.

The handler method for a `HaystackCreateEvent` checks to see if a body needle was specified at archive time. If not, it creates a new body needle and attaches a causal tie from the location needle to the body needle. The fetcher then calls the protected abstract method, `setBodyData`, which should be defined by a subclass of the fetch service to set the data of the body needle. Once the body needle has

have the same location, then we want to make the connection between them by having them point to the same location needle (this can happen if the document changes and the user wants to keep both versions in Haystack).

data, the fetch service requests the checksum of the body from the body needle. The checksum, as with the document location in the archiver, is checked against a database of previously archived checksums. If a match is found, then the fetcher uses the needle of the matching checksum (as well as the body needle that caused the matching checksum's needle); otherwise, it creates a new checksum needle, attaching a causal tie from the body needle to the checksum needle. The last action of the fetcher service is to add the checksum needle to the `HaystackDocument` (the body needle is not attached at this point; the reason is given in the next section).

Often the data for a body needle is a promise (see section 3.3.2). In fact, a sub-package of the promise package is dedicated to fetch promises. The class `haystack.object.promise.fetch.FetchPromise` is the abstract superclass of all fetch promises. A fetch promise provides a protected method (`httpGet`) to its subclasses for retrieving a document specified by a URL. The `fulfill` method for the fetch promise first finds the location needle for the body whose data is the promise and then passes it to a protected `fulfill(Location)` method. This method returns the body of the document which is then returned by the fetch promise's `fulfill` method.

The URL fetch promise is the simplest implementation of a fetch promise as it's `fulfill(Location)` method just calls the `httpGet` method on the URL in the location needle. Similarly, the URL fetch service (`HsURL`), is our simplest fetch service. However, from Figure 7-1, we can see how little code and how little knowledge of how the DM works it takes to create a new fetch service. The abstract fetch service does most of the work. Similarly, the abstract fetch promise does most of the work for fetch promises. We have defined similar abstract services (and promises, where relevant) for other data-manipulation functions (described below) that make it easy for the user to extend Haystack.

7.1.4 File Checker

Once the archiver and fetcher have run, the metadata structure will be a `Haystack-Document` tied to location and checksum needles. This metadata subgraph is the

```java
package haystack.service.fetch;
import haystack.service.ServiceName;
import haystack.object.StrawType;
import haystack.object.needle.Location;
import haystack.object.needle.Body;
import haystack.object.promise.fetch.URL;
import haystack.exceptions.*;

/**
  * This is an event service that listens for the creation of a
  * document Bale with a URL needle.  It provides the setBodyData
  * method that FetchService needs to properly handle a fetch on
  * a URL Location.
  */
public class HsURL extends FetchService {
  /**
    * the name of this service
    */
  public static final ServiceName name =
    new ServiceName("HsURL", "v1.0", "haystack.service.fetch");

  /**
    * the basic constructor
    * @exception DuplicateNameException if this service has already
    *                                   been loaded
    */
  public HsURL() throws DuplicateNameException {
    super(HsURL.name);
    this.docTypes.addElement(StrawType.URLNeedle);
  }

  /**
    * sets the data in bodyNeedle to be a fetch promise for a URL
    * @param locNeedle The Location Needle
    * @param bodyNeedle The Needle for the body of the doc
    */
  protected void setBodyData(Location locNeedle, Body bodyNeedle) {
    bodyNeedle.setData(new URL(bodyNeedle.getID()));
  }
}
```

Figure 7-1: FetchService Example – HsURL.java star graph that triggers the default file checking service, `HsFileChecker`. Aside from the responsibilities of handling that event, the file checker is also responsible for maintaining the database of previously archived locations and checksums and the `ArchiveOptions` for documents. Also, it maintains state information about documents that are currently being processed. The state information is then used when the event handler is called.

We did not mention earlier what exactly is done with the location and checksum match status that the archiver and fetcher obtain. In fact, the archiver and fetcher report those statuses to the file checker. As well, the fetcher does not tie the body needle to the document, but instead reports it to the file checker. We do this because at the time the fetcher runs, it does not know if the document will be archived for sure, so we don't want to trigger any services expecting a body by tying the body to the document's bale. When the file checker is triggered, it looks up the matching information for the document that triggered it and then queries the archive options object to tell it what to do. If the archive options object says to abort, then any new straws that were created for this document are removed from persistent store and processing stops. If we are to merge, then the location needle that triggered the file checker service is tied to the document's bale with a tie called `AdditionalLocation`.

Superceding is a much more complex situation. If we are to supercede, then we must check to see whether it was the location or the checksum that matched. If the location matched, we want to supercede the body (and checksum). That requires untying the body tie and checksum tie for the matching document and tying the new body and checksum needles to the document. As well, the file checker must tie the new body tie and checksum tie to the old body tie and checksum tie with a superceded tie. This effectively cuts off the old body and checksum from the document bale, allowing access only through the new checksum and body. Any other new straws that were created for the new document will be removed from the persistent store. On the other hand, if the checksum matched, we want to supercede the location. To do this, the file checker reties the location needle of the matching document to the document's bale with an `AdditionalLocation` tie.

The new location needle should then be tied to the matching document with the old location tie, and all other new straws that were created for the new document should be removed from the persistent store. All processing should stop at this point.[3]

The file checker exports some of its methods for use outside the root server's VM. Specifically, the lookup methods for previously archived locations and checksums and the archive options for archived documents are all exported. This allows an external program to query whether a location or the contents of a file have already been archived and indexed by Haystack. The archive options are made available for programs that the user might write to walk through a user's Haystack and, perhaps, decide whether or not to re-archive a document.

7.2 Metadata Extraction

Once we have a body for a document, we can extract various pieces of metadata. This metadata can be used in database-style queries either alone or in conjunction with a free-text query. The following are examples of the metadata that is extracted: the file type of the document, the author of the document, the creation date of the document, a summary of the document, and the text of the document.

7.2.1 Type Guesser

The type guesser, `HsTypeGuesser`, is a event service triggered off of the structure of a `HaystackDocument` tied to location and body needles. The job of the type guesser is to examine the location and body and guess the type of the document. Currently, we have implemented the type guesser for file types. For many file formats, examining the name is sufficient for guessing the type. However, we do need to examine the body sometimes, as we learned with the Perl version. We have not yet implemented any of the checks needed on the body of the documents. Implementing this addition

[3]The fetch service and file checker service will both be triggered with a create event when the new location needle is tied to the document, but they will discover, through causal ties, that the metadata they normally set is already there, so they just exit.

will not be difficult as it involves mostly converting the Perl 5 regular expressions that we have in the Perl type guesser to use the ORO, Inc. pattern matcher for Java [19].

7.2.2 Field Finding

Many times, we will want to restrict our queries based on a database-style condition. For instance, we may only be interested in documents written by a certain author or created after a certain date. We will have field finder services to extract these fields from documents. We do not yet have any field finder services implemented, but they will not be difficult to create. As with other data manipulation services, we will have an abstract field finder service that provides an abstract means of adding fields. HTML, LaTeX, and email field finders are in development, and more can be added by users. For more information about the integration into Haystack of database-style queries based on these types of fields see Adar's thesis [1].

7.2.3 Textifying

Since we use a text-based IR engine to index our metadata, we must convert a document's body to a textual representation. This conversion is done by a textifier service. Many document formats, such as email, are already plain text. Others, like HTML and LaTeX, have markup tags that should be removed. Postscript and DVI documents contain commands that create the text of a document; for these documents, we don't really want to index the commands, but instead index the text they produce. Other binary formats such as multimedia files could have textifiers that describe the contents in textual form. We currently have a textifier written for postscript format. Plain-text formats are assigned a default textifier that just passes the text through, and all other formats have a textifier that returns nothing. We will soon have textifiers for some markup language documents, and we are always looking to add outside textifiers for other formats.

The results of a textifier are placed in a text needle. This needle represents the text of the body of the document. The text needle, however, does not represent the entire text representation of the document we wish to index. As discussed in section 7.4.2 about IR system services, other straws in the metadata for a document can be included in the full text representation of a document.

An Abstract Textify Service

We have defined an abstract textify service, `TextifierService` in `haystack.object.textifier` to allow users to easily add textifiers for document types not handled by Haystack already. The textifier service defines a vector, `docTypes`, that subclasses should set. The vector contains the straw types of the file type needles for documents handled by the textifier. During initialization, the textifier registers interests for `HaystackDocument`s tied to each of the straw types in `docTypes`.

The event handler finds the body of the `HaystackDocument` that triggered it by following the causal tie from the file type needle.[4] It then checks causal ties pointing into the body needle to see if a text needle has already been caused by the body. If so, it returns. If not, the handler proceeds to make a new text needle. It uses an abstract method to set the data for the needle. This abstract method, `getTextData`, takes as arguments the document's bale, the body needle, and the text needle and returns the data for the text needle. As is discussed in the next section, most data for text needles will be promises.

Textify Promises

Since the size of the text of most documents is on the order of the document itself, we really don't want to store the actual text in our metadata as disk space could become a problem. In an ideal world with limitless disk space, storing the actual bits of the text would be quicker, but given that we do have space limitations, we settle for storing instructions for generating the bits. We do this by using textify promises.

[4]recall from the type guesser, section 7.2.1, that all file type needles have a causal tie to the body the type is for.

Specifically, we have an abstract promise `TextifyPromise` in the package `haystack.object.promise.textify`, just as we had an abstract fetch promise. We believe most textify promises will be fulfilled by calling some external command that acts on the body of a document to do the textification. Many of these utilities have been written and are freely available over the Internet or purchased through third-party software vendors. The abstract textify promise provides four ways to to fulfill a textify promise by sending the body caused by the text needle for which the promise is the data to an external command. The four ways correspond to using either a file or a pipe for the input and/or the output to the command. In all cases, the method takes the command to be executed as the first argument, plus any other necessary information. Table 7.2 lists these methods.

Table 7.2: `fulfill` methods for `TextifyPromise`

Input	Output	Method
STDIN	STDOUT	`fulfill(String)`
STDIN	file	`fulfillInputFile(String, String)`
file	STDOUT	`fulfillOutputFile(String, String)`
file	file	`fulfill(String, String, String)`

Example: Postscript documents

We will now go over an implementation of the abstract textification service and abstract textify promise. The textifier service, `HsPostscript` declares its `docTypes` to be only the postscript file type needle. When triggered, it sets the data for the text needle of the document to the postscript textify promise, `haystack.object.promise.textify.Postscript`. When the data is requested from the text needle, the textify promise is fulfilled. The promise pipes the body of the document to the external command `pstotext` using the `fulfill(String)` method of the abstract textify promise. We chose the `pstotext` because it is freely available and is fairly lightweight and easy to compile (very simple c files). Again, the addition of handling the textification of postscript documents was very easy given our abstract classes that have already been defined.

7.3 Handling Document Collections

Various kinds collections of documents exist in the information space. These can range from multi-part documents such as an RMAIL file or a MIME-encoded message to file-system structures like a directory. These collections, regardless of their origins are treated as documents by themselves in Haystack. The collection of parts is represented by a bale. The parts themselves are also documents on their own and are indexed as such. Every location that is not a URL location in Haystack now is a collection; in the future we may have locations other than URL that are not collections (e.g. if someone comes up with the next World Wide Web that uses a new protocol for specifying locations). There are also collections for some of the file types of a URL location. Note that collections from a URL location will always have a body themselves besides the contents of their parts. However, collections represented by locations do not necessarily have a body associated with them (e.g. a directory). The three document collections we want Haystack to support initially are directories, mail files, and queries. We present the design and expected implementation of these handlers here, as they are not yet implemented.

7.3.1 Directory

A directory collection is generated from a directory location. It corresponds to the physical collection of documents on a filesystem. The directory service, `HsDirectory`, will be responsible for creating a directory collection. It listens for a document bale connected to a directory location. The event handler will first tie a directory file type needle to the document bale for the directory. It will then create a directory bale and proceed to recursively archive each file in the directory, using the same archive options as those given to archive the directory itself, and placing a pointer to the `HaystackDocument` bale for each newly archive document in the directory's directory bale.

7.3.2 Mail files

Mail files are collections of email messages. Several mail file formats will be handled by Haystack including RMAIL and From formats. Each format is a specific file type for a document with a URL location. The messages themselves have a special kind of location that depends on the format of the mail file. For RMAIL and From formats, the location for a message would be tied to the URL location for the mail file and the message ID of the message. This uniquely specifies the message, and allows a fetcher for RMAIL or From format messages to retrieve the message.

The event handler in the event service that handles the mail collection will be triggered when a mail file is assigned its file type needle. The handler for a mail file needs to parse the file and break out each message. It should create a type of mail location for each message and send that location and the body it extracted for each message to the archiver. While metadata extractors might be limited for the mail file itself, mail messages are a format which lends easily to extraction of several fields including the to, from, and subject headers.

7.3.3 Queries

Query collections will be created by the query service. We currently have a simple query service that, much as the archiver does for archiving, acts as a middle layer between the UI and the DM for adding new documents to the Haystack. This query service will be extended to create these query result collection documents. In this case, though, the collections represented by queries are not physical document collections. The are conceptual document collections. A whole file system, called Semantic File System [7], has been constructed on the idea of specifying locations of files or groups of files by queries. We believe that if Haystack keeps these collections, indexes them, and changes them over time as it observes the user, Haystack can use the query collections as part of new query results. Since the query collections will have been adapted over time to match the user's information needs, they should provide information that Haystack can use to reorder result sets of new queries to put matches more relevant to the particular user at the top.

7.4 Indexing

Once the changes to the metadata from the previous data manipulation services are complete, it is the job of the indexer to pass each new document that was generated to the IR system. This is the final step of our core set of data manipulation services. We currently use document bales to define our *indexable clusters.* This is an arbitrary decision which could be changed to include any part of the metadata. An indexable cluster is a grouping of straws that should be indexed as a unit. There are two steps to indexing, the first is deciding when an indexable cluster has settled into a stable state, that is when no more services are being activated to modify it. Once we have decided that it is in a steady state, we want to queue up the file to be indexed by the particular IR system we are using (we could actually use more than one IR system if desired, though right now we limit it to one IR system).

7.4.1 Indexer

The indexer is an event service. However, unlike other event services that have predefined interests, the user can specify the interests of the indexer. This allows arbitrary structures in the metadata to be indexed. The most popular structure to be indexed is a document bale connected to a text needle. Every time a text needle is attached to a document bale or the data of a text needle that is connected to a document bale changes, the indexer would get called. Other examples of indexable structures are if a comment needle is attached to a document bale and if an author needle is attached to a document bale. When the indexer is triggered, it should wait for the indexable cluster at the source of the event to settle down to a steady state. Once no more services are acting on the indexable cluster, the cluster can be sent off to an IR system to be indexed.

The indexer service is implemented in the class `HsIndex`. We have limited the star graphs that trigger the indexer service to be single-ray stars. We really do not need anything more complex as the indexer does wait for a cluster to settle down before indexing it. However, if the config service is improved to include values more complex than vectors, we can specify arbitrary star graphs to trigger the indexer. The star graphs that the indexer registers interest in are specified by two vectors in the config service: `AutoIndex` and `Indexable`. The elements in these two vectors match up. The element in `AutoIndex` specifies the root of the star graph, and the element in `Indexable` should be the straw at the end of the ray. The tie connecting the two is generated based on the straw type of the ray's tip. For example, if a pair was specified as `HaystackDocument` and the straw type for a text needle, then the star graph interest for that pair would be a `HaystackDocument` tied to a text needle with a text tie. These vectors are read in when the service is initialized. As well, the delay to use in checking whether it is time to index the indexable cluster is read in from the config service at initialization time.

The indexer service maintains two queues called the wait queue and the index queue. When the service is first triggered, the source of the event (that is the root of the indexable cluster) is added to the wait queue. If a timer thread has not already been started for the service, one is (see appendix section B.1 for a description of a timer thread). The delay on the timer thread is set to be the delay we read from the config service upon initialization. Every time the timer thread generates a timer event, all straws in the index queue are checked with the resource control service to see if they are locked (recall that anything modifying a piece of the DM must lock that piece first). If they are, that means services are still acting on them, so we should wait. If not, then they are sent to the IR system service (described below in section 7.4.2). All straws in the wait queue are transferred to the index queue every time the timer thread generates a timer event. We have the wait queue to insure that any straw added between generation of timer events gets at least the delay amount of time before it is checked. This check is not perfect as we could perform the check when a straw is in between services and, thus, not locked, but that just means index will be called again and the indexable cluster will be reindexed.

7.4.2 The IR System Service

The IR system service is responsible for efficiently making calls to the underlying IR system for a user's Haystack. It provides a layer of abstraction between Haystack services and the IR system. The IR system service supports three basic features for every IR system. First, it provides a facility for clearing the index. Second, it provides a facility for adding to the index, and finally, it provides a facility to query the index. More complex features can be supported on a per-IR system basis, but these are the three that all IR systems must support. To help in the efficiency of calling the IR system's index facilities, The IR system service includes an index queue that regulates how many files get sent to the IR system at once. We do not want one file at a time fed to the IR system if we are doing a batch job of several files, as that would be inefficient, but we also do not want to wait too long after we archive a file to be able to query on its contents. The queue tries to balance these two concerns.

We have defined an abstract class, `IRSystemService`, in the package `haystack.service.ir`, that implements the index queuing and defines the methods needed to provide a full interface to an IR system. The assumption made is that the IR system has some way to store the bits to index before calling the IR system on those bits. If not, it is up to the implementor of the subclass of `IRSystemService` to implement this. When a straw is first sent to be indexed, the `generateIndexable` method of the straw is called with a size parameter. This method returns a mixed vector of strings and streams that make up the textual representation of the indexable cluster rooted at the straw. The total size of the text in this vector is limited by the size parameter passed to `generateIndexable`. The vector of text along with the ID of the straw is sent to an abstract method, `writeToIX`. This method should be defined in subclasses of `IRSystemService` to store the bits to be indexed. The ID of the straw is then placed on the queue to be indexed.

The queue is maintained inside a helper thread. The helper thread is defined in an inner class to `IRSystemService` called `IRHelper`. This helper thread is running whenever we have straws to index. The thread has two parameters which help it decide when to actually call the IR system. The first parameter is delay and it represents the amount of time the thread waits (measured in milliseconds) between checking the queue. The second parameter is called threshold and represents the number of documents that can accumulate in the queue before we flush them out of the queue and send them to the IR system. This sounds quite a bit like a timer thread (see appendix section B.1). However, we hope to make the queue parameters dynamic in the future so that during times when we are adding a lot of documents, we could potentially raise both the delay and the threshold, and conversely, in times when we are not doing heavy indexing, we could lower either or both of the values.

Example: Isearch

We have an implementation of `IRSystemService` for the Isearch IR system in the class `HsIsearch`. `HsIsearch` defines a `writeToIX` that writes each straw's indexable text to a separate file. The IX file name is the ID of the straw appended with a *.ix* extension. It also defines an `indexDocs(Vector)` method that is used by the IR queue in `IRSystemService` to send a vector of IDs to the IR system's index function. This command actually runs the command necessary for Isearch to index the bits in the IX files. Finally, `HsISearch` defines a `query` method that calls Isearch's query function on a query string and parses the results into a `QueryResultSet` (see appendix section B.2 for a description of the `QueryResultSet` object).

7.5 Extensions

With a specific IR system service defined, we have now given a complete description of the core data manipulation services needed for Haystack to function. Many additions can be made to the core set of services. As well, new services that act on our metadata based on user actions can be written. For example, a proxy server is in development that will watch a user's web activity and potentially archive new web pages or modify the metadata for pages already archived. We have tried in designing our services to make extensions as easy as possible, abstracting general functionality away in abstract services wherever possible. We have especially made it easy to add services that handle new document types to the system.

Chapter 8

Conclusions

We have now specified the complete design and implementation of the infrastructure of Haystack. With this infrastructure in place, we are ready to begin using the system on a trial basis and to begin exploring the issues of adaptation to a user's interests and inter-haystack communication.

8.1 Future Goals

We desire to have an archivist for Haystack to periodically update the archive so that the user is not forced to manually update every file. However, issues of keeping an archive of all changes to files verses keeping an up-to-date index come into play. Ideally, the user should be able to choose, perhaps on a per-file basis, how to deal with keeping the archived file up-to-date. We saw in section 7.1.4 how complicated such decisions can be.

The web server that will provide our main user interface is being completed now. Until then, the command-line interface is really not sufficient to effectively use the system. In the future, we would like to add other interfaces through Emacs and through Java applets. With a proxy server in place, we can begin recording user behaviors including query behavior and browsing behavior so that we can 1) archive new documents and 2) examine the logs to see what can be learned.

There are two main extensions to our core system that are planned as well. First, we would like Haystack to learn from previous queries. This could involve using the query bales that show up in a result set for a query in some way to improve the ordering of the result set. Second, we want to have inter-Haystack communication so that a Haystack that cannot fulfill a user's information need can query other users' Haystacks.

8.2 The Dream

The dream of Haystack is to become the gateway to a user's library of information. That covers all electronic documents on a user's PC, network file systems, and even the paper documents. Haystack should maintain all of these information sources and answer user's information needs with the best possible matches. Over time, Haystack should learn to adapt to the particular interests of the user and update them as they change over time. Finally, haystack should be able to query other user's libraries when an information need cannot be met with the user's library alone.

8.2.1 The Past

The Perl version was used in a functioning system. Joshua Kramer integrated Haystack into the intelligent room in the AI lab at MIT as part of his thesis [15]. The intelligent room is a computer-controlled room that supports speech recognition and has multimedia presentation facilities. The information that the room kept track of was diverse, from web pages that describe the AI lab to VCR stops on a VHS tape. With Haystack integrated, the user could verbally query for information and a result set was displayed on a wall of the room. The user could point to a result with a laser pen and the document would be displayed in whatever format was appropriate, be it a web page in a browser window or a video clip on a TV.

8.2.2 The Present

Currently, Haystack is still in internal development. Most of the core functionality has been implemented with the completion of this thesis and Adar's thesis [1]. Again, we are looking to integrate the new version of Haystack with the new version of the intelligent room.

8.2.3 The Future

There are many paths Haystack could take in the future. Certainly, it will become a research tool for exploring how an IR system can adapt to a user's interests over time. We hope that a study of users' query behavior can be done using Haystack so that we can better understand what sorts of learning we can do based on query behavior.

Appendix A

Glossary of Java and Haystack Terms

A.1 Java Terms

VM A *Virtual Machine* which has its own address space and can interpret multiple Java threads. More than one VM can be running on a machine at a time.

Package A directory that contains related files (you can also have packages within other packages).

Interface A class declaration which does not have any implementation (only public methods are allowed, and they are just declarations, no definitions).[1] A variable's type can be an interface; that variable can then only be assigned a value whose class implements the interface. Typically interfaces are given adjectives or adverbs for names unlike classes which should be nouns.

Implementation A class that implements a particular interface.

Abstract class A class which is halfway between an interface and an implementation: some methods are defined, some are only declared. An abstract class may not be instantiated.

[1] Public static final data members are also allowed.

Inner class A class which is defined inside another class. Typically, these are helper classes which are used only inside the containing class.[2]

Extends The Java term for inheritance. If class c1 extends class c2, then c1 inherits from c2. Java does not support standard multiple inheritance, so a class can only extend one other class. However, it can implement an arbitrary number of interfaces.

Serializable A marker interface that Java recognizes to indicate that a class can be converted into a stream of bytes and sent either to disk or over the network. This usually involves converting each of the data members of the class to bytes, though classes can define their own custom serialization methods.

Introspection The ability of Java to examine the class of an object, looking at the return type and argument types of specific methods.

A.2 Haystack Terms

Archive Verb: The act of saving a copy of a document in a central repository (potentially for good); Noun: said repository.

Index Verb: The act of examining the text of a document and recording, in an efficient manner, the text for future querying in a central repository; Noun: said repository.

Textify The act of setting the text component of a document or actually producing the textual form of the document.

HAYLOFT The place where Haystack keeps all of its files. This includes the archive, the index, the metadata, and any configuration information and supplementary databases.

[2]Inner classes have access to all members (public and private) of their containing class. If an inner class is not declared static, it also has access to the *this* variable of the instance of its containing class.

IX file A file which contains text in a format suitable for a particular IR system.

`HsService` The root of all Haystack services. Note, all instantiable Haystack services start with *Hs*.

preferences The file that contains configuration settings for Haystack.

DM Object An object from the Data Model (extending `Straw`).

Haystack ID A unique identifier assigned to every DM object.

Appendix B

Utility Classes

B.1 TimerThread

A timer thread is a thread that triggers an event every x milliseconds, where x is specified when the thread is created. The listener for the event must be specified at the time the thread is created, and only one is allowed. All timer thread-related classes are in the `haystack.utils` package. The timer thread is implemented in `TimerThread` which extends Java's `Thread` class. The event class for the event generated by the timer thread is `TimerEvent` which extends Java's `EventObject`. The listener for the event is defined in an interface, `TimerListener` which defines one method, `handleTimerEvent(TimerEvent)`. The timer thread provides a method, `stopTimer()` which ceases the generation of timer events, but does not stop the thread (useful if you want the current timer event to finish being handled when you shut the timer down.

B.2 QueryResultSet

A query result set is an enumerable list of pairs representing the result of a query. The elements of the pairs are the id of the straw whose indexable cluster matches the query and the rank the IR system returned for that particular match (for IR systems that do not support ranking, 1 is assigned as the rank for all matches). The ranking ranges from 0 to 1. This object provides a method for iterating through the list much like Java's Enumeration interface [5]. It defines a `nextID` method that returns the next matching id. The `score` method returns the score for the current id. Finally, a method `reset` provides a way to reset the iterator to the beginning of the list. The order of the matching ids is dependent on the order they were added. This list represents a set and, thus, will not allow duplicate ids in the list.

Bibliography

[1] Eytan Adar. Hybrid-search and Storage of Semi-structured Information. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1998.

[2] Michelle Q Wang Baldonado and Terry Winograd. SenseMaker: An Information-Exploration Interface Supporting the Contextual Evolution of a User's Interests. Technical Report SIDL-WP-1996-0048, Stanford University, Stanford Digital Library Project, September 1996. http://www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1996-0048.

[3] C. Mic Bowman, Udi Manber, Peter B. Danzig, Michael F. Schwartz, Darren R. Hardy, and Duane P. Wessels. Harvest: A Scalable, Customizable Discovery and Access System. Technical Report CU-CS-732-94, University of Colorado, Department of Computer Science, March 1995. ftp://ftp.cs.colorado.edu/pub/cs/techreports/schwarts/Harvest.Jour.ps.Z.

[4] Digital Equipment Corporation. *AltaVista Personal Search 97.* http://www.altavista.digital.com/av/content/searchpx.htm.

[5] David Flanagan. *Java in a Nutshell.* O'Reilly & Associates, Inc., Sebastopol, CA, second edition, May 1997.

[6] William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* Prentice Hall, EnglewoodCliffs, NJ, 1992.

[7] David K. Gifford, Pierre Jonvelot, Mark A. Sheldon, and Jr. James W. O'toole. Semantic File Systems. http://www-psrg.lcs.mit.edu/Projects/SFS/newsfs.ps.

[8] Elliotte Rusty Harold. *Java Network Programming.* O'Reilly & Associates, Inc., Sebastopol, CA, first edition, February 1997.

[9] Marti A. Hearst. Interfaces for Searching the Web. *Scientific American*, March 1997. http://www.sciam.com/0397issue/0397hearst.html.

[10] Marti A. Hearst and Jan O. Pedersen. Reexamining the Cluster Hypothesis: Scatter/gather on Retrieval Results. In *Proceedings of the Nineteenth Annual International ACM SIGIR Conference*, Zurich, Germany, June 1996. http://www.parc.xerox.com/istl/projects/ia/papers/sg-sigir96/sigir96.html.

[11] I-search. http://www.isearch.com/.

[12] Java Remote Method Invocation - Distributed Computing for Java, March 1998. http://java.sun.com/marketing/collateral/javarmi.html.

[13] David R. Karger and Lynn Andrea Stein. Haystack: Per-User Information Environments. http://www.ai.mit.edu/people/las/papers/karger-stein-9702.ps.

[14] Arkadi Kosmynin. An Information Broker for Adaptive Distributed Resource Discovery Service. In *Proceedings of the First International Conference of Web Society*, San Francisco, CA, 1996. http://aace.virginia.edu/aace/conf/webnet/html/356.htm.

[15] Joshua David Kramer. Agent Based Personalized Information Retrieval. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1997.

[16] Mg. http://www.kbs.citri.edu.au/mg/.

[17] Scott Oaks and Henry Wong. *Java Threads.* O'Reilly & Associates, Inc., Sebastopol, CA, first edition, January 1997.

[18] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA*. Wiley Computer Publishing, New York, CA, first edition, 1997.

[19] *OROMatcher Users's Guide.* http://oroinc.com/developers/docs/OROMatcher/index.html.

[20] Mark A. Sheldon. *Content Routing: A Scalable Architecture for Network-Based Information Discovery.* PhD thesis, Massachusetts Institute of Technology, Department of EECS, December 1995. http://www-psrg.lcs.mit.edu/ftpdir/papers/sheldon-phdthesis.ps.

[21] Mark A. Sheldon, Ron Weiss, Bienvenido Vlez, and David K. Gifford. Services and Metadata Representation for Distributed Information Discovery. http://paris.lcs.mit.edu:80/ sheldon/dist-indexing-workshop-position.html.

[22] Larry Wall. *Programming Perl.* O'Reilly & Associates, Inc., Sebastopol, CA, second edition, October 1996.