Hybrid-Search and Storage of Semi-structured Information

by

Eytan Adar

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1998

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

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

Certified by ....
Lynn Andrea Stein
Associate Professor
Thesis Supervisor

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

Hybrid-Search and Storage of Semi-structured Information

by

Eytan Adar

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

Abstract

Given today's tangle of digital information, one of the hardest tasks for information systems users is finding anything in the mess. For a number of well documented reasons including the amazing growth in the Internet's popularity and the drop in the cost of storage, the amount of information on the net, as well as on a user's local computer has increased dramatically in recent years. Although this readily available information should be extremely beneficial for computer users, paradoxically it is now much harder to find anything.

Many different solutions have been proposed to the general information seeking task of users, but few if any have addressed the needs of individuals or have leveraged the benefit of single-user interaction. The Haystack project is an attempt to answer the needs of the individual user. Creating such a system requires solving two problems. Half the problem addresses the manipulation of the data into a queryable format. Once the user's information is represented in Haystack, the other half of the problem centers around our desire to answer the highly varied questions a user may ask about this information. In this thesis we will propose a means of representing information in a robust model within Haystack and we will describe a corresponding mechanism by which the diverse questions of the individual can be answered. This novel method functions by using a combination of existing information systems. We will call this combined system a *hybrid-search* system.

Thesis Supervisor: David R. Karger
Title: Associate Professor

Thesis Supervisor: Lynn Andrea Stein
Title: Associate Professor

Acknowledgments

I would like to especially thank David Karger and Lynn Stein for making this thesis possible in the first place. Without their encouragement and support I would not have been writing my Master's thesis today. Additionally, a great deal of the design of the Haystack system is a result of my interactions with David and Lynn. I would also like to thank Mark Asdoorian without whom Haystack would be much less functional than it is.

To Jerry Saltzer and Mitchell Charity who got me started in research: thank you. To the nice people at Stanford (Jason McHugh and Roy Goldman) who implemented Lore and then compiled the code, fixed the bugs, and explained the system to me: you guys are awesome.

Of course none of this would have been possible without my family and my friends, so thank you Mom and Dad, Udi, and Sara. Although you made me that much fatter, thanks Sasha for making sure I had enough sugar in my bloodstream and keeping me company at the LCS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Given today's tangle of digital information, one of the hardest tasks for information systems users is finding anything in the mess. For a number of well documented reasons including the amazing growth in the Internet's popularity and the drop in the cost of storage, the amount of information on the net, as well as on a user's local computer has increased dramatically in recent years. Although this readily available information should be extremely beneficial for computer users, paradoxically it is now much harder to find anything.

Many different solutions have been proposed to the general information seeking task of users, but few if any have addressed the needs of individuals or have leveraged the benefit of single-user interaction. The Haystack project is an attempt to answer the needs of the individual user. Creating such a system requires solving two problems. Half the problem addresses the manipulation of the data into a queryable format. Once the user's information is represented in Haystack, the other half of the problem centers around our desire to answer the highly varied questions a user may ask about this information. In this thesis we will propose a means of representing information in a robust model within Haystack and we will describe a corresponding mechanism by which the diverse questions of the individual can be answered. This novel method functions by using a combination of existing information systems. We will call this combined system a *hybrid-search* system.

1.1 The Problem

There are two different but related search tasks in the digital domain. The first is exemplified by a search in a large digital library. The user is only aware of a few items and does not (and probably *will* not) grasp the full nature of the corpus[1] (just as a library patron doesn't know the full set of books the library has). The second type of search task is more constrained, and answers the user's need to find things in their own information space (i.e. his/her own bookshelf, rather than the library). The bookshelf/personal model has two unique properties that differentiate it from the library: a) the user has direct control of what goes on the bookshelf, and b) the user has direct control of the bookshelf's organization.

Until recently this second type of search task has not been a singificant problem. Users were restricted to limited disk space and limited bandwidth, and could search their corpus manually. Both factors contributed to limitations in the amount of information a user would ever be exposed to. Most systems were therefore designed to deal with the large corpora present only in library type systems. This class of library information systems can be further categorized by the intended audience. One class deals with the very general class of users (i.e. the Internet user base). The other deals with a very small number of specialized users (i.e. librarians or researchers). Unfortunately, neither approach completely satisfies, nor takes advantage of, information specific to a single user.

Today, we are rapidly approaching the end of the limits to personal disk space and bandwidth. Information is making its way quickly to — and residing permanently on — the user's desktop. While in the past personal information retrieval has been only an abstract problem, it has become a real issue today. In this thesis we will concentrate on finding information on the personal level. For our purposes, personal information corresponds to things that a user possesses in some form. The information can exist as physical paper, as digital bits, in the user's memory, or any combination of these forms.

[1] Corpus is an Information Retrieval (IR) term generally referring to a collection of documents.

What kind of questions might we ask about this type of information? Some examples are:

- *Where is the file that David sent me last week?*
- *Which e-mails did I get last month from Lynn that I didn't respond to?*
- *What web pages did I see about "semantic association networks?"*
- *Who wrote this document and when did I get it?*

These questions are a small sample of what a user may ask when trying to make sense of, organize, or find something in, their personal information space. Can we apply previous methods of approaching the general search problem to the personal situation? The short answer is: not efficiently or effectively. A more detailed answer requires an understanding of the tools currently available to us. Section 1.1.1 will describe a model for querying an information system for object retrieval in a collection of data. In section 1.1.2 we develop categories for the different components in this model. We then extend the model in section 1.2 for hybrid-search. Finally, in section 1.3 we will summarize the goals of this thesis.

1.1.1 A Model for Searches

Figure 1-1 illustrates our view of the query process. This feedback system abstractly models aspects of current search mechanisms, and illustrates where this work hopes to improve on them.

- Initial query function: $Q_1(i, c, \Delta) \rightarrow q^i$, where i is the information need, c is a set of hints, and q^i some combined description of the information needs and hints. We will return to the Δ shortly.
- Secondary query function: $Q_2(q^i, G) \rightarrow q^f$, where G is some grammar associated with the search tool, and q^f a formal query expressed in the grammar.
- Information system core function: $I(q^f, D) \rightarrow r_1$, where D is the corpus (data) we are querying against, and r_1 some intermediate set of matches.

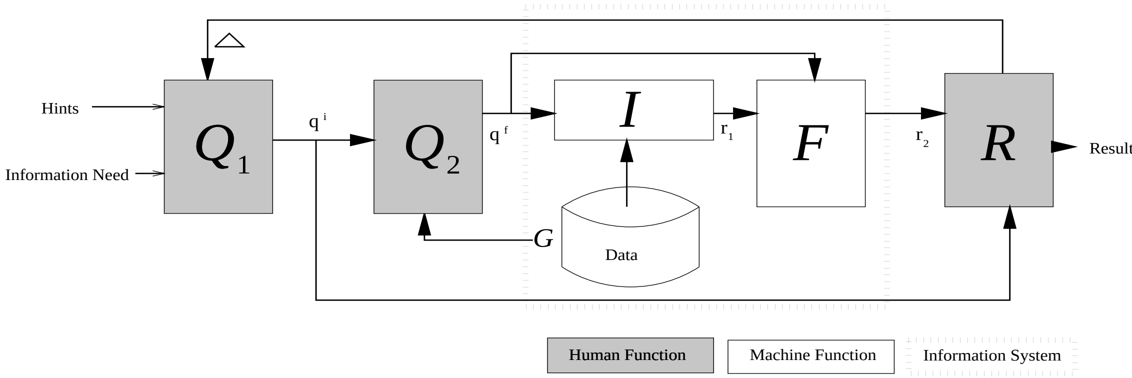

Figure 1-1: The query model

- Information system "post-processing" function: $F(r_1, q^f) \to r_2$, where F performs some post-processing on r_1 to generate some r_2.
- Human evaluation function: $R(r_2, q^i) \to \begin{cases} \text{generate } \triangle \text{ if result unsatisfactory} \\ \text{output } r_2 \text{ otherwise} \end{cases}$

Let's look at this system qualitatively for a minute. We start off with a data corpus (the *data* in the figure) which contains some information the user would like to find. The first task a user performs is to combine their information need (i.e. what they are searching for), with any hints they may have. For example, if the user is searching for some document (their information need), they may also remember that Bob wrote it last year and that it was about "apples" (the hints). The user combines the information need with the hints to form (possibly just in their minds) some informal query in Q_1: "I want the document that Bob wrote last year about apples." Hints are not necessarily directly related to the information need of the user. For example, the book a user is looking for on probability happens to be red. "The book is red," is a hint the user can provide to the help the system to find the book. However, the fact that the book is red has nothing to do with the information need of the user which can be, "information on Markov chains."

Ideally, we would like a user to be able to informally phrase the combination query of information need and hints to the system. Unfortunately, systems today are incapable of handling arbitrary natural language queries, so it is necessary for the user to generate some formal query based on the language specification of the information system. This corresponds to the function Q_2. The previous query might then become (if our information system were a SQL database), "SELECT * FROM docbase WHERE author = 'Bob' AND date > 01/01/97 AND date < 01/01/98 AND topic = 'apples.' "

The user then passes this formal query to the information system, I. The information system in turn generates a set of responses that satisfy the user's formal query. If it is capable of doing so, the system may also rank the responses as to their likelihood of satisfying the query. We will get back to this. This primary result set is then passed to a post-processing function, F. This function acts as an additional level of computation on the initial result set as specified by the formal query. The general idea is that sometimes a user wants a single top matching response, or alternatively sometimes they want a summarization of all the responses, or possibly some other representation (sorted by date, for example). In effect, we generate a new document that reports on the content (or location) of other documents. Portions of the formal query that contain post-processing instructions are passed to F for evaluation.

Finally, once we have this secondary result set it is necessary to turn again to the user. The user must evaluate what was returned by the system based on their informal query. If the information satisfies their need we are done. Otherwise, the user must form what we will call Δ (delta, the mathematical symbol for change). This Δ corresponds to some additional restrictions or modifications the user will apply to his original informal query function, Q_1. Let's go back to the initial query for an example. If Bob wrote many papers last year with a majority being about "apples" *and* "oranges" the system will return many matches. If the user is only interested in the documents that are exclusively about apples, they will have to revise their query to reflect this need. The user can now say that they are interested in, "all documents Bob wrote last year about 'apples' but not 'oranges.' " With a new informal query in hand, the user will now repeat the previous walk through the system. What we would like to do is minimize the number of iterations so that a user gets what they want much more quickly. However, before discussing our particular solution it would be valuable to understand how current solutions fit in this model, where they do the right thing, and where they fail.

1.1.2 Query Model System Components

In discussing the system components we can follow the same route as a human would, and consider the query formulation process first (functions Q_1 and Q_2). Abstractly, there are many different kinds of searches users perform for different information needs. These include searches for data for analytical needs, searches for documents that the user created, and searches for documents related to a current research thread. Different information systems are designed to handle different types of queries. We will call the computable query to the information system the *input*.

Functionally, we can distinguish between two extremes of input into the system[2]. The first of these we will call the *exploratory* or browsing search tools. This type of search involves the use of a "probe" into the information space followed by an iterative refinement of the search until satisfying information is found. An example of this type of search is, "show me douments that have information on the crime rates in northern California cities." The second type of search we will call the *explicit* search. Users will perform an explicit search when they are aware of the existence of some document or object and wish to locate it. An example of this is, "find the web page on the MIT site dealing with patent licensing." Between these two extremes of exploratory and explicit lies the information search spectrum [11]. What distinguishes one end from the other is the set of hints the user applies to the query. Users of information systems begin to formulate better hints either by becoming adept users of the systems or using the feedback loop from the result sets.

But what exactly distinguishes exploratory from explicit? To understand this we introduce two new terms, *confidence* and *specificity*. Imagine an omnitient user. In making a query, the user will be able to cause the system to return a perfect match to the information need (assuming the system provides adequate retrieval mechanisms). We say that this query has high specificity (we get exactly what we want). We can

[2]Ordinarily, the need of a user will fall somewhere between these extremes.

also assume that the user has high confidence that his query will return the right thing. When the user has low confidence and low specificity we can say that they are performing an exploratory search. As the user begines to understand the information space, they are able to pose more specific queries with greater confidence, in essense pin-pointing the document(s) that matches their information need. Unfortunately, queries of higher confidence and specificity tend to be more complex and consist of various hints that a given information system may not be able to understand or utilize. This is important for the system we are attempting to develop here. Naturally, the user of a personal information system will be able to pose queries with higher specificity and confidence than a user of a general information system. We therfore desire an information system that is able to take advantage of the more precise and useful hints a user generates.

Just as we categorized the types of queries, we can do the same for different I functions. The I function is the *implementation* of the information system, and we devise three categories for it. The first form can be classified under the heading of *associative* search tools. For the most part, these correspond to hypertext systems, and allow for explicit, sometimes non-trivial[3], connections to be made between documents. The second category is the *unstructured* or fuzzy search tools. The unstructured search refers to the questions a user has about the (generally) implicit meaning of a document. These tools generally correspond to information retrieval (IR) systems. IR systems were designed to work well with the operations of indexing and searching among text based collections. The last category consists of *structured* or deterministic search tools. Traditionally, these have been database systems. Structured search does not necessarily have to do with the structure of the document, but is usually about querying some data in a predefined schema. An example of this type of query is, "return all documents that are of type *e-mail*." or, "return all documents that are dated 9/9/97"

The final component that requires some discussion is the post-processing function,

[3]By non-trivial we mean that current computer technology could not have created the links automatically.

F. Imagine a query, "find the average grade for student x for this year," applied to database of student records. This can be broken down into two tasks: "find all the grades for student x for this year," and, "average the result set." The first task we have already solved through function I. The second task, which involves some post-processing, must now be applied to the result set. This is where the filter function F comes in. There are a variety of filters we may wish to see implemented. We can design F to allow all documents to pass through. Borrowing from linear algebra, we will call this the *identity* function. Alternatively, we can build some functional summary of the data. We call this function the *aggregate*. Various database technologies provide this facility as a feature of their system and for certain applications this serves a very practical purpose.

Finally, we categorize the complete system $(Q + I + F)$ as the *class* of the system. What will become apparent is that different classes of systems will be better suited for different types of problems. The intuition is that different systems are designed and optimized for different problems. Systems with an Information Retrieval core are generally intended for full-text fuzzy matching. Databases are (usually) intended for highly constrained, exact searches. Hypertext systems are (usually) intended for building complex inter-document associations. Because these different systems have different properties, each is well suited towards different tasks. We believe that at different times, users have different information needs and constraints, and will require different types of systems to satisfy those needs. So while one unified solution would be optimal, this may not be possible.

In chapter 2 we will review why current systems that fit our search model all have useful and negative characteristics. An ideal solution would augment the variety of information systems with the features of other information systems. A system composed of complementary sub-systems, is in our opinion, a much more ideal solution that satisfies the full spectrum of user needs. We call this system, the *Hybrid-Search*.

1.2 Better living through Hybrid-Search

Let's back up for a moment and look at our problem from one level higher. What we would like to accomplish in general terms is a system that assists users remember and organize. We are *not* interested in having the user think in the constrained grammar rules we set for our system. Rather we would like our system to act within the rules and models a human would apply to finding information.

The tool we would like to provide to users is one that is flexible and robust enough to handle the user's internal methods of recall. This means that we would like to provide users with a set of tools that allows a diverse set of queries in which the user may express the same set of hints that occur in their own minds. We call this tool *hybrid-search*.

This thesis was proposed within the context of the Haystack project[23], as a means to extend the functionality of this adaptive personal information repository system. Details of Haystack's core implementation, which have made this thesis possible, will be discussed in great details in chapters 4 and A.

The Haystack architecture provides the means to rapidly integrate a variety of information systems into a user's functional Haystack. From the query perspective, it possible through Haystack to build a "multiplexer" that selects information systems that directly match the users needs, and a method of combining the results. From the indexing side, we can build tools to centrally maintain a data repository. This functionality will allow for the hybrid-search mechanism we are aiming for. Figure 1-2 reflects the goal of breaking up a query into different components for processing by the optimal information system.

This revised model splits the information system into three parts: $I_1 + F_1$ (the IR system), $I_2 + F_2$ (the database), and $I_3 + F_3$ (the hypertext system). Additionally, the model introduces two new functions:

- A Multiplexing function, $M(q^f) \rightarrow q_1^f, q_2^f, q_3^f$ which routes different parts of the original query to the information system(s) best suited to deal with it.

- A combination function, $\Sigma(r_{2_1}, r_{2_2}, r_{2_1}) \rightarrow r_2$ which merges the results of the

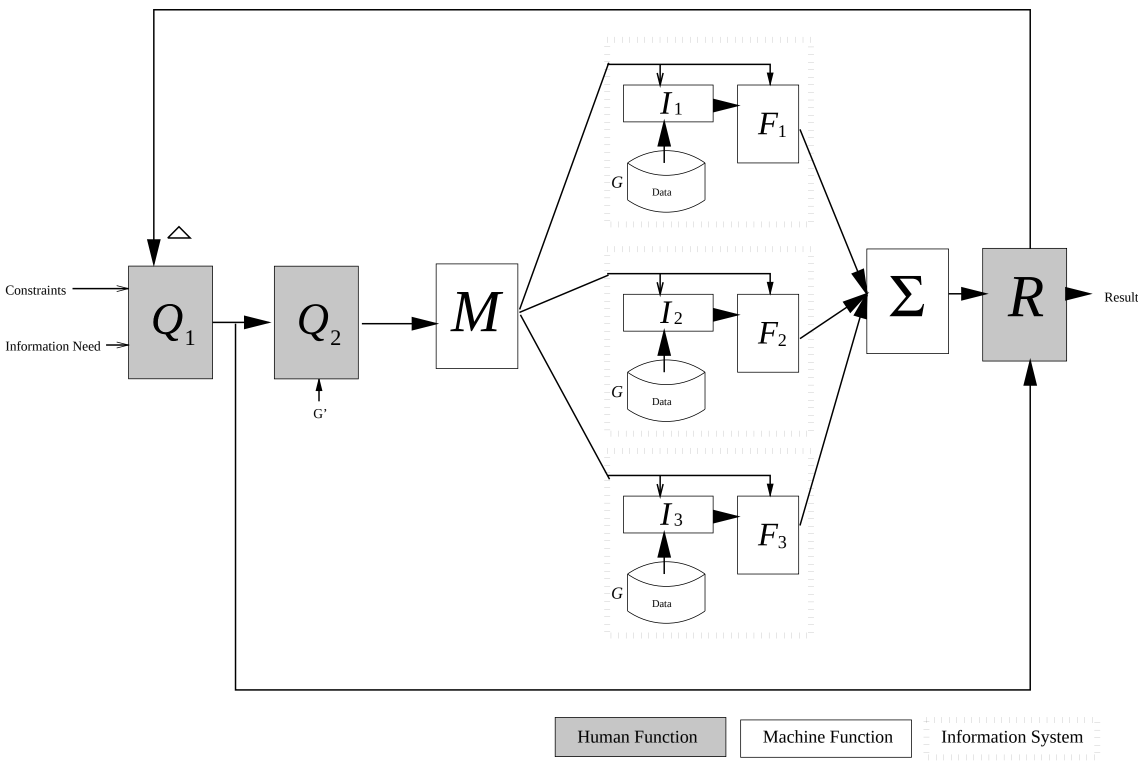

Figure 1-2: The Hybrid-Search Query Model

different filter functions into one result set.

If it is not entirely clear yet what type of query a user could issue to such a system it may be helpful, in conclusion, to provide some examples of hybrid-search type queries (which independent information systems would not be able to answer):

- *Which documents do I have about four legged animals that I wrote last year?* The first part of the query, "about four legged animals," is an unstructured query (we are assuming that we didn't pre-categorize the documents in the collection). The second part, which limits the set of documents to a certain time period, is a structuerd (database) query.

- *Which email did I send to David in regard to his request for clarification on Lore?* This is very similar to the first query. The first part limits us to emails that David sent, again a structured query. The second part, which may require some semantic understanding on the part of the information system, is answerable by an unstructured (information retrieval) system.

- *Which web pages did I visit after looking at the MIT homepage last week?* The first part of the query, "web pages ... after ... MIT homepage," is an associative search[4]. The second part of the query constrains the results to "last week," and is a database type query.

1.3 Thesis Overview

This thesis proposes that:

1. Our hybrid-search system should provide more efficient query mechanisms than hybrid-search *emulation* on any one of the other tools independently.

2. Our system should demonstrate an efficient mechanism for storing the semi-structured information used in our system.

[4] This assumes we have marked which document the user went to after visiting any given page.

3. The system should demonstrate an efficient mechanism for extracting this information.

The next chapter (Chapter 2) reviews previous efforts and gives a general overview the technologies used in this thesis. Chapter 3 describes a model for documents and why we believe hybrid-search will work well in this domain. Chapter 4 covers the implementation of the Haystack data structures. The Haystack data structures will provide us with the means of modeling the document structure. The service model provides the tools for converting documents into the Haystack data model structures. Chapter 5 details the Haystack to Lore interface, an essential component of the hybrid-search implementation. Finally, Chapter 6 discusses some conclusions and describes some future directions for Haystack. Appendix A discusses in great detail the design of the revised service model included in the new release of Haystack.

Chapter 2

Background

In this chapter we will review previous efforts in information system research that may prove useful for the type of hybrid-search capabilities we will attempt in Haystack. Specifically, we will approach the previous work from four directions: unstructured search technologies, structured search technology, associative search technology, and current hybrid solutions. Finally, in Section 2.2 we will discuss the two primary building blocks for this thesis, Haystack and Lore.

2.1 Previous Work

2.1.1 Structured Search

The major contrast between unstructured and structured search is the items we are trying to categorize and index. While the unstructured IR system concentrates on the act of "indexing" across one dimension (language features), structured systems tend to index across multiple dimensions. Traditionally, structured search tools fall into the domain of database technology. A very good introduction to database technologies is [12].

What kind of information is structured? The easiest way to think of structured information is something one could put in a table. For example, if someone had a large compact disk (CD) collection, and wanted to be able to find certain CDs easily, they might create a database based on a number of different dimensions: CD title, performer's name, date released, date purchased, number of songs, song titles, etc. In setting up the database, the user would specify the dimensions[1] and the data types of each dimension. Most databases provide a number of different data types for indexing. These can include, strings, characters, real numbers, integers, and esoteric multimedia and BLOB types.

Another way to look at structured search applications is in terms of *metadata*. Metadata, for our purposes, are information about information. That is, if the CD in the previous example is the "information" we are looking for, the dimensions by which we categorize the CD (title, performer, etc.) are the metadata. In the structured search a user generally does not know anything about the data itself, but understands (or at least has access to) the metadata. Because databases understand data types, it is possible to formulate powerful queries on information other than text. An example of this type of search is, "give me all the names of employees who make more than $30,000 a year." Equivalently in SQL, the (S)tructured database (Q)uery (L)anguage, this could be, "SELECT name FROM employees WHERE salary > 30000." Notice that it is now possible for us to ask questions about numerical values. Symbols such as the > sign have a significance when applied to numerical dimensions.

While it was possible to test cetain mathematical predicates, most databases are incapable of judging between the semantic similarity of two items. For example, while they can tell you that "$1 < 2$" is true, they can't decide if "MIT is near Harvard." A notable exception to this are systems such as VAGUE [34] that allow for extensions to the notion of similarity by allowing additional domain and rule knowledge to be added to the database schema. Unfortunately, querying such a system is tedious and requires much interaction as the system attempts to understand what the user meant. If we look for an Italian resturant near our house, the system will try to understand the concept of "near" and how important certain portions of the query are (i.e. is it more important that the restaurant is "near" us, or that it serves Italian food?).

[1] If we look at the database as a table, the dimensions are generally the columns (i.e. title, performer, date, etc. in this case)

To take full advantage of database systems a user has to have a very good understanding of the structure of the search space as well as the query language. On the other hand, if the user does understand both the structure of the data and the structure of the query language, information can be found in fewer iterations of searching.

An additional constraint of most database systems is their inability to extend past their original schema[2] definitions. That is, once a user describes the dimensions of his data this is more or less set in stone.

In summary:

- Database systems are able to generically deal with many data types.
- Database are good at deciding *equallity* between queries,
- But they have no sense of similarity at the semantic level.
- Databases define constrained information spaces (schemes or dimensions).

2.1.2 Unstructured Search

An unstructured search is generally done by users who require some semantic understanding by the system of the information. The user may only have a conceptual understanding (rather than specific knowledge) of the information they are searching against. We can also think of this type of textual search. Alternatively, we can refer to this type of search in terms of the tools intended as a solution, the Information Retrieval (IR) problem.

An example of this type of search is the case where a user may know, or at least think, that the words "Apple pie" (or some variant) exist in the object(s) they are looking for. The user may or may not have ideas on how to find this particular piece of information. Regardless, they will generally approach the task in the same manner. The first step is to start with some textual search engine, and submit some free-text query. The query is an indication of what the user thinks the document

[2]The schema is esentially the model of the data in the database (i.e. the dimensions upon which we categorize the data).

should/will contain or more generally, what (s)he hopes it will contain. Depending on the quality of the query, the experience of the user, and the capabilities of the information retrieval (IR) system, a useful result may be returned to the user.

IR research has a long history in computer science. Starting in 1950 with H.P. Luhn's KWIC indexing system[28], researchers have endevoroued to create computerized retrieval systems for bibliographic and "full text" applications. A very good general overview of IR technology is available in [41]. What is both a boon and a problem for IR systems is the intense efforts made by the research community to address issues of textual indexing.

The general idea behind trivial IR systems is the use of an inverted file index which maps words to their source documents. Much experimentation has been done to provide optimal data structures for this task. These include anything from hashtables, to B-trees, to more esoteric structures. The goal, of course, is to provide an efficient (i.e. $O(1)$ or some other low order) lookup operation. When a user issues a query for $word_1 \cap word_2$, the IR system will look up the set of documents containing the word $word_1$ and then the set of $word_2$, and find the intersection of the two sets. This type of fairly simplistic IR system is called a boolean IR system. Boolean systems generally have a number of operators based on their namesake, Boolean logic (i.e. AND, OR, NOT, etc.).

Other systems, which are of more interest to us, are the "fuzzy" IR systems. A fairly simple example is a boolean system with ranking. Essentially, when the IR system evaluates the boolean query, it will rank the returned documents based on the number of conditionals the document satisfies. A much more powerful ranking system is the vector-based IR system. In very simple terms, documents are projected into n-dimensional[3] vector space. When a query is issued a vector corresponding to the query terms is created and thrown into the document vector space. The IR system then uses a metric (typically the inner product) to find the vectors that are the closest to the query vector. The theory behind this approach is that documents

[3] Where n corresponds to some subset of words in the given language (i.e. "Apple" is one dimension, "Bowl" another, etc.)

with high co-occurence of words cluster together, and queries that share terms with those documents imply that the documents are related to the query. The exact methodology for this approach is not necessarily the driving factor for it's discussion. Rather, the general concept, that IR systems produce fuzzy matches is of greatest importance. IR systems, being oriented towards the indexing language features, have evolved to capture semantic information about documents.

IR systems are judged by a *precision-recall* metric. *Precision* is simply:

$$\frac{\text{The number of correct answers returned}}{\text{The total number of answers returned}}$$

Recall is defined as:

$$\frac{\text{The number of correct answers returned}}{\text{The number of correct answers in the system}}.$$

Correct implies that something, or more precisely someone, has made the judgment that a certain document matches a certain query. A human will inevitably have a different view of "correctness" of a certain document based on a semantic understanding of the material. It is therefore necessary for IR systems to contain the ability to work "fuzzily" in a way that creates some semantic, rather than purely syntactic, understanding.

The semantic information derived by IR systems is by no means "intelligent." For the most part it has some statistical basis or was derived by some rule based heuristic. Statistical mechanisms include the elimination of low entropy words (i.e. words that are not useful in distinguishing between documents). Heuristic methods include the removal of pre-set, common words (these are called stopwords). Additionally, IR systems have been developed that provide relevance feedback and machine learning functionality that allow the system to improve its results over time. These algorithms create the appearance of semantic understanding by observing human operation. In the rule based algorithm tool kit, IR systems hold thesauri (for indexing not only the word, but its synonyms) and stemming algorithms (for breaking a word down into its root and indexing by the root).

The unfortunate side effect of complex IR system behavior is the inability to handle certain queries. For example, what can we do if we wanted to index a document by additional features such as author, or date? We can of course *emulate* this behavior to some extent. We can place the phrases "author: David" and "date: 01/01/99" at the beginning (or end) of the document, and allow the IR system to index the terms. Furthermore if we wanted to have one document imply a relation to another, say a citation to a document name *document1*, we can add the text "cites: document1."

We call the concept of implementing one information system on top of another (in this case database and hypertext on top of IR) an *emulation* system. When we want to find all papers written by David about "oranges" we could make the query: "'author: David' *AND* 'oranges'." The IR system (assuming it is capable of finding phrases and not just words) will quite possibly return the right thing. Such a query works for the reason that an IR system is capable of matching text to text. However, what happens when we want to find all documents about oranges written between 12/15/98 and 1/13/99? We are no longer able postulate such a query. Even though the IR system can provide us with fuzzy answers, these are fuzzy in terms of text and not numerical values. Searching against different data types is not a forte of IR systems.

In summary:

- IR systems are oriented towards language features
- IR systems have a refined sense of similarity between queries and text documents (i.e. they have some sense of language semantics).
- IR algorithms are optimized for text indexing.
- IR system responses are fuzzy. They return *probable* matches based on some heuristic.

2.1.3 Associative "Search"

Interestingly the first attempt to describe a personal information assistant described what would later lead to the large body of hypertext research [9]. In his essay, *As We May Think*, Vannevar Bush asked his readers to

> Consider a future device for individual use, which is a sort of mechanized private file and library. It needs a name, and to coin one at random, "memex" will do. A memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory.

The memex Bush describes is a mechanical device for building associations between "documents." Subsequently a number of efforts were made to produce such as system in the digital domain, most noteably, Douglas Engelbart's system AUGMENT/NLS (oN Line System)[14]. Although both Bush's and Engelbart's visions were for the inidividual human, Hypertext developed in a different direction. Disk space and bandwidth were once a scarce comodities, and the information systems built tended to have an emphasis on instituitonal knowledge representation. To this day, especially with the advent of the WWW, hypertext systems tend to be used to display one specific view that is produced by the hypertext creator. The associations created between objects generally satisfies the author's agenda, and not necessarily anything the reader could conceive.

So what exactly do hypertext systems do for us? Most systems allow for the creation of named "links" between documents. These links imply relationships between the documents that a user may traverse to get from one document to another. The claim is that associations would seem to be more valuable than mere indexing because humans think in terms of associations [9]. Hypertext systems therefore allow humans to explicitly describe relationships between documents that may not have been derived by a machine. In the case where the hypertext system is in fact entirely text (rather than hypermedia) a number of proposals [2] [4] devise methods for generating links on-the-fly.

However, while hypertext systems are good for browsing, on their own, they are usually ill equipped for searching purposes.

In summary:

- Hypertext systems are good for explicitly defining connections between documents.
- Hypertext systems are good for browsing a document corpus from a certain perspective.
- Hypertext systems would seem to replicate the same type of model humans use for remembering.
- Hypertext systems are not necessarily useful for easily finding things (unless the path is known).

2.1.4 Hybrid-Search

Hybrid Emulation

A number of previous efforts have endevoured to create a hybrid-search solution by emulating one type of system on top of another. In [38] and later in [17] a model is described to extend standard relational algebra to allow Non First Normal Form (NF^2) relations. Such a system allows for nested information in a relational databases and the necessary power to emulate some IR behavior. This approach is however limited to the expressiveness of relational databases and requires new implementations and new query languages to function. Some databases allow for extension by the user and in [30] a database is extended using abstract datatype (ADT) to allow for textual indexing. However, this system only allowed boolean queries to be performed against the data.

In [21] a method is described for using standard SQL to emulate a ranking IR system. This is achieved by creating additional database tables holding TFIDF[4] tables. The argument is that such a solution is more efficient and cost effective than either creating a new type of database or utilizing both an IR and database system seperately. Such a system, unfortunately, suffers from the limitation of the database in handling text specific heuristics (stemming, thesaurus) as well as IR system features such as relevance feedback, and vector based indexing/retrieval.

A similar, albeit Oracle[5] specific, experiment is described in [13] . Most databases will endevour to satisfy a set of so called ACID[6] properties. By utilzing an ACID satisfying database (i.e. Oracle), a consistency is ensured between the text of the database and other structured information. Although in this specific case the database emulates an IR system (rather than using IR heuristics and algorithms), such an approach is generally refered to as a tight coupling. Tight coupling implies that the consistency is maintained between the IR and database systems.

Although tight coupling is achievable even when the IR and Database systems are seperate there is a cost in efficiency. The hybrid-search model we propose will be loosely-coupled and we will not enforce atomic transactions on both the database and IR systems. Rather, a lazy updating scheme ensures that the information represented by the two information systems will converge over time for any given document.

Loosely-coupled Hybrid-Search Solutions

Although in some cases it is attactive to emulate an IR system through a database, generally an approach that links two independent systems proves to be the more popular choice.

A number of systems have approached the query and data fusion problem between disparate information systems using centrally managed domain knowledge. Systems such as [5] utilize domain knowledge to dispatch queries to the appropriate destina-

---

[4]TFIDF is $\frac{termfrequency}{documentfrequency}$ and is simply a way to measure the number of times a word appears in a document vs. in all the documents. This measure is then used to judge relevance of a document to a query.

[5]A commercial database manufacturer, see `http://www.oracle.com`.

[6](A)tomicity, (C)onsistency, (I)solation, and (D)urability

tion and merge the results. This assumes a formal specification of the properties of different systems.

Researchers at the University of Massachusetts, Amherst have perviously suggested a model for intergrating the INQUERY IR system with an Object Oriented database (OODB) in [10] and with a relation database [42]. Another attempt was also made by a different group [22] to intergrate INQUERY with the Sybase SQL database. All three approaches relied on a specialized query language that was preprocessed by a mediator and submitted to either the IR system or the database. Unfortunately, all three database systems require a pre-defined schema for the database and are not readily extensible. The combination we propose will allow for dynamic shifts in the structure of the database.

2.2 Technologies

2.2.1 Haystack

Over the past two years our research group has worked to implement the first version of the Haystack personal information repository system [2]. As of August, 1997 the first version, written in Perl, was released for public review. The system implemented a number of our design goals, the most abstract of which was to provide the users of our system with a repository that implements both intelligent storage and retrieval of arbitrary data. Although we are currently limited to textual representations of the data, anything on the user's hard drive or on the Internet should be fair game for becoming an entry in the user's Haystack repository. A user can easily organize and find information in their repository at a later time. To facilitate these operations, Haystack will make an effort to extract a textual representation of the data object, object metadata, and allow users to annotate the object. In the future, we hope that we will not need to be reliant on "textual representations" and be able query images or sounds. However, for the most part, we are dependent on the technologies of today.

At a more detailed level, Haystack consists of a number of core modules. User interface modules (which currently include a personal web server, command line, and Emacs) provide access to the system. Textifiers provide the machinery to extract text from a variety of file formats. Field finders extract metadata information about a given object. Finally, Haystack provides modules that interact bi-directionally with information retrieval (IR) systems. If any textual information is derived from the object, Haystack can index this information in any basic information retrieval system. We want this abstraction because different IR systems will represent their data differently and present different query options to users. This includes anything from basic Boolean operations (such as "all documents that contain 'MIT' and 'Artificial Intelligence' "), to more complicated relevance ranked schemes (i.e. "documents that are most strongly related to MIT and Artificial Intelligence").

As noted above, Haystack extracts both text (through textifiers) and metadata (through field finders) from data objects it encounters. Metadata includes anything from the automatically generated checksum field, to the user generated comment field.

2.2.2 Lore

Although Haystack provides us with the interface to a variety of information retrieval systems, it is necessary to find a suitable database and hypertext system. We were lucky to discover a system that would provide us with a capability for both built-in. The Lightweight Object REpository (Lore) system from Stanford's database group [32] is a DBMS capable of querying and indexing semi-structured information. This ability satisfies our need to store objects that do not have a pre-defined schema. Lore is an extension to the Object Database Management Groups (ODMG) database standard. The language used to interact with Lore is known as Lorel (Lore language) [1] and extends the Object Query Language (OQL) language (which in turn extends the SQL standard).

Lore data structures are based entirely Object Exchange Model (*OEM*) objects. These data structures can take on atomic values such as *integer*, *real*, *string*, etc. OEM objects can also be complex. Complex OEM objects contain named links to other OEM objects. Additionally, all OEM values carry a unique ID (*OID*). The

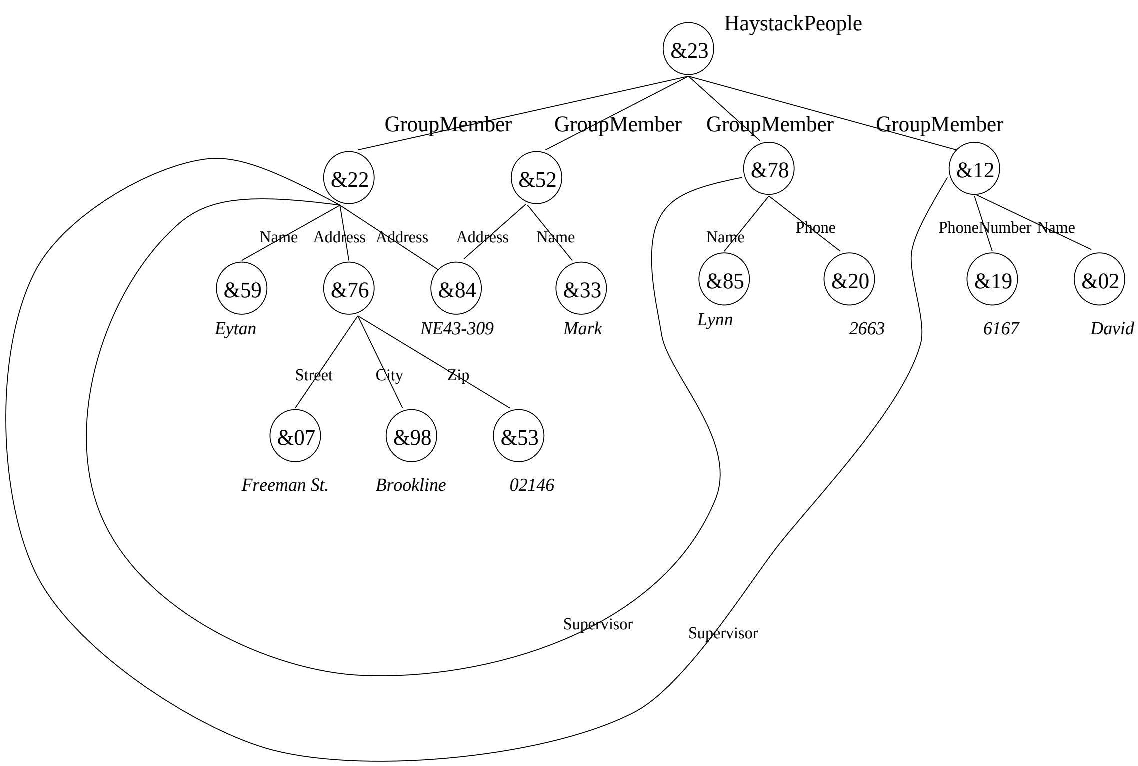

Figure 2-1: A Sample OEM Graph

structure formed by a Lore database can therefore be an arbitrarily complex graph structure.

We will return to the specifics of Lore when discussing its intergration into Haystack (see Chapter 5). However, as some basic knowledge of Lore is required for the next chapter, some examples may help. Figure 2-1 is a sample graph of a Lore database. The text on the connections between OEM nodes indicate the named relationship between the parent (the higher node) to the child (the lower node). The number inside the nodes is the OID of the object. The atomic values for those OEM that have them are shown below the node in *italics*.

There are a number of important things to note about the graph structure we generated. The first, is the ability of OEM objects to hold any number of "children" of a given type. By not pre-defining a schema we can connect any number of named links to an object (*Address* for example). Additionally, it is the link which is named (i.e. is of a specific type) rather than the object it points at. This gives us the ability to connect to a number of objects with the same link name regardless of how the objects are represented. For example, one of the *Address* nodes for the "Eytan" object is a complex OEM object pointing at three other OEM objects: *Street*, *City*, and *Zip*. The other *Address* object is an atomic object with a string component. Finally, we note the ability to explitly name objects, and to set points to those objects.

Figure 2-2 is the corresponding textual OEM model that is used to generated the graph. An OEM "object" consists of a link name followed by some value, between the signs < and >. For example, we create the Name node by specifying: `<Name ‘‘Eytan’’>`. To create a complex object, we place the "children" in braces: `<linkName {oemObject1, oemObject2, ...}>`. Finally, we can name objects by using placing a name followed by a colon in the OEM "constructor." For example: `<AD1: Address ‘‘NE43-309’’>` binds the name *AD1* to the object pointed at by the *Address* link (i.e. *NE43-309*). To point at a named object we use the ampersand followed by the name of the object. For example, in the "Mark" object, we point the *Address* at *AD1* by using: *<Address &AD1>*. It is not necessary to use the same link name in pointing to the same object. We could have, for example, said: `<Office &AD1>`.

```
<HaystackPeople::HaystackPeople {
            <GroupMember {
                        <Name ``Eytan''>
                        <AD1:  Address ``NE43-309''>
                        <Address {
                                    <Street ``Freeman St.''>
                                    <City ``Brookline''>
                                    <Zip 02146>
                                    }>
                        <Supervisor &Lynn>
                        <Supervisor &David>
            }>
            <GroupMember {
                        <Name ``Mark''>
                        <Address &AD1>
            }>
            <Lynn:  GroupMember {
                        <Name ``Lynn''>
                        <Phone 2663>
            }>
            <David:  GroupMember {
                        <Name ``David''>
                        <PhoneNumber ``6167''>
            }>
}>
```

Figure 2-2: A textual representation of the OEM graph

This would have created an *Office* link pointing at the *NE43-309* object.

Now that an OEM representation has been created, it is possible to start asking Lore for information about this structure. The Lorel query syntax looks suprisingly like SQL.

$$\boxed{\text{SELECT } a \text{ FROM } b \text{ WHERE } c_1 \text{ AND } c_2 \text{ AND } \ldots.}$$

The variable a is the view we want to take of the returned data (i.e. how much of the data we want to see). The data we query against is limited to the set b, and we impose additional constraints with the predicates c_1. For example we can issue the query:

```
select X.PhoneNumber from HaystackPeople.GroupMember X
            where X.Name = ‘‘David’’
```

This statement creates a set X of *GroupMembers*, we then sonstrain the set to those group members whose *Name* is David. Out of the result set, we select the object pointed at by the *PhoneNumber* link. In this case, we return “6167.” A slighly more complex example is:

```
select X.#.Phone% from HaystackPeople X
            where X.GroupMember.Name = ‘‘David’’
```

The two characters # and % are wildcard chracters. The # is a path wildcard charcater which will match any link name. In the above examples the # evaluates to: *GroupMember* (only a single path in this example). The *Phone%* means that we are looking for links starting with the string *Phone*. In this case, *Phone* and *PhoneNumber* both satisfy the requirement. However, the where clause limists us to the “David” *GroupMember*, and we again return “6167.” A complete specification of Lorel is available in [1].

The final aspect of Lore that requires some discussion is the concept of DataGuides [19]. Lore has the built in ability to take a snapshot of the current database and produce a new database representing the schema of the original. Such a tool is highly useful for visualizing the domain in which a user can query. A user can, for example, fill in differnt fields within the graph in order to perform a query against the database. This would allow a user to perform queries without necessarily understanding the structure of the database. A DataGuide for the graph before is represented in Figure 2-3.

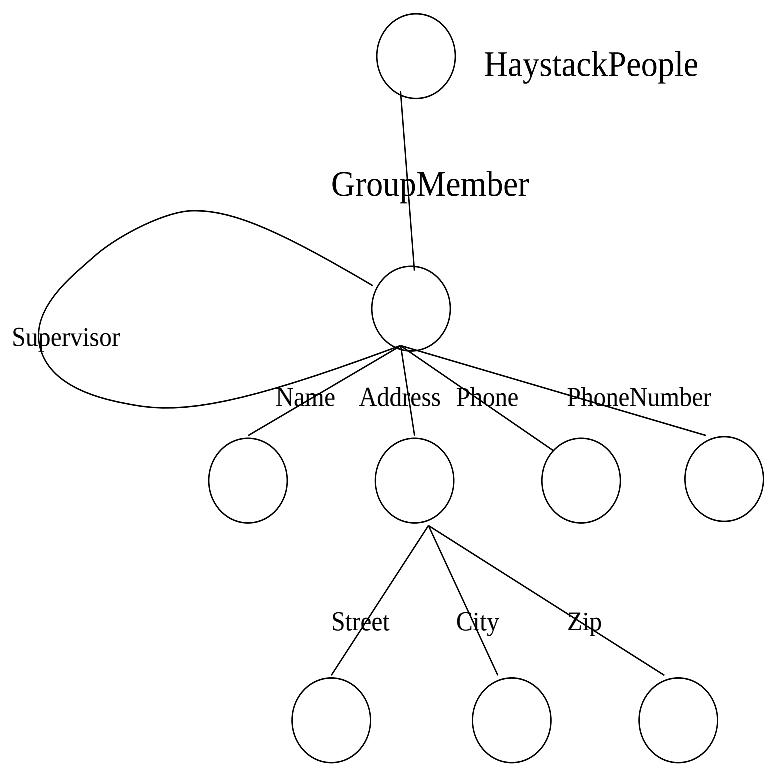

Figure 2-3: A DataGuide Graph

Chapter 3

A Tale of Paper and Straw

Before we can continue with a formal discussion of the implementation of hybrid-search in Haystack, it would be advantageous to understand exactly why hybrid-search is an appropriate method for searching through documents. This chapter will create a model for documents based on state. By analyzing the structure of this model, and the type of queries a user may invoke against it, a data model for Haystack will be developed. We will then show how to index this data structure into a hybrid-search system.

3.1 Documents

To understand the purpose of this thesis, it is important to understand the information we are working with. While we call these objects simply documents, the term is loaded with meaning. This chapter, among other things, will discuss the philosophical (as well as more reality-grounded) aspects of documents [7].

There has been much research of the nature of documents, and an all encompassing study of these definitions is well beyond the scope of this thesis. For our purposes we can condense a number of definitions to give documents following definition: "A source of information in any form which is archivable." This broad definition applies, therefore, to all media formats as long as they maintain some sort of persistence. It is notable, however, that Haystack functions best on documents with some *text* form.

With this definition of documents it is now possible to explore their internal nature. We make the additional argument that documents are composed of *stateful* properties and relations. We can then leverage this state to represent documents within Haystack in a manner that allows for improved searches.

Document State

We are now ready to define a new model for documents, which will be called the *document state model*. A document can be seen as an entity present at some point in time. The document as a unit has some state within that instant of time. This document will also have some properties and relations. The properties of the document and its relationship to other documents can also be seen as units of information with state.

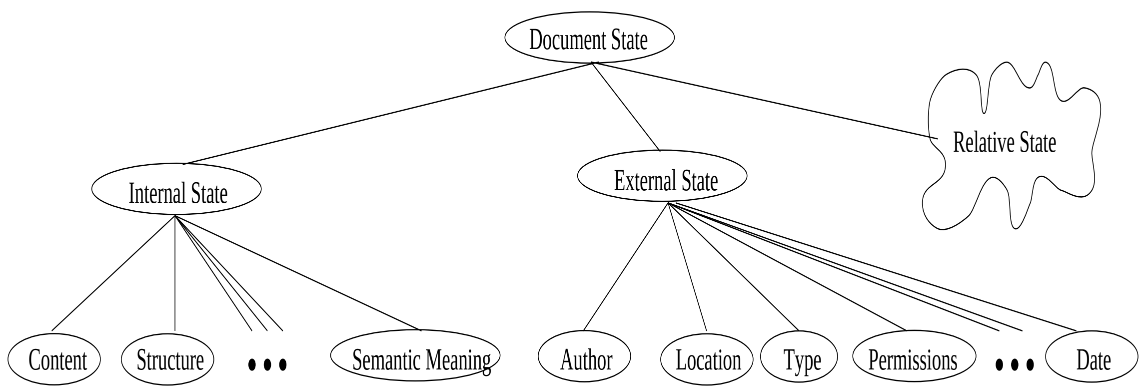

Figure 3-1: A breakdown of document state

Figure 3-1 illustrates this model of document state. Some portions of the document state can be derived automatically, others by semantic interpretation of a user when reading the document. We can separate the document state into three constituent parts (based on properties and relations):

- *Internal State:* Properties that are of this type generally refer to the content or internal representation of the document. For example we can say that a document *contains* the word "apple." Generally, things that are part of the internal state of the document are properties we can define by looking inside the document. These properties may be non-obvious, and may be derived by some semantic understanding of the document[1].

- *External State:* Properties that can be considered meta-data are generally held as part of external state. The author, date of publication, and location are good examples of external state. However, it is important to note that properties are not necessarily exclusive (i.e. a property can be part of more than one state). For example, the author of a document can be "David" by virtue of the fact that we know that David wrote it and emailed it to us. This knowledge is part of some external state. By reading the document we can find that it is "by: David." So we have derived an authorship relation by semantic understanding, and so the author being "David" is also part of the internal state.

- *Relative State:* There are a set of fuzzy properties of a document that encapsulate relationships between one document and another. One document can cite another or link to it. These relationships can sometimes be derived automatically. Sometimes, however, the relationships are derived based on some semantic understanding of a user. This semantic undestanding may be obvious to everyone, to a limited group of users, or to the single user alone. For example, one document can be related to another by virtue of the fact that the reader read both while eating a chocolate ice cream cone. As stated before, we will place non-trivial inter-document properties in this category.

The final important idea to consider is that state of a document may change. We will make the claim that documents present state is also a function of its previous sates. For example, while a document (call it *myThesis*) may have been located on *goose.lcs.mit.edu*, it may have subsequently moved to *haystack.lcs.mit.edu*. A user looking for the document *myThesis* by constraining his search system to the previous location may come up empty handed if the system does not remember that the

[1]We make this claim even though semantic understanding is partially based on comprehending the document in terms of other documents or concepts. If a property is semantically non-trivial (i.e. different readers of a document can conclude different values of the same property) we can add this information to the relative state information.

document used to be on *goose.lcs.mit.edu*. Any piece of state that has previously existed may be of importance to the user who will later look for the document based on this previous state. Additionally, the more expressive an implemented system can be in maintaining all the state information of a given document, the better it will be able to respond to user queries.

3.2 Straw

Now that we have a basic understanding of the document domain, we can turn again to developing a data model for Haystack. Abstractly, our goal is to map what we have learned the document model into usable data structures.

When discussing the query model previously (see Section 1.1.1), we found two components that were part of every query, the information need, and hints. Hints were used to narrow down the results produced by the information system to satisfy the information need. In Haystack we would like to be able to handle three distinct types of hints:

- Hints based on document content
- Hints based on document metadata
- Hints based on relationship between documents

These hints naturally map to the three components of document state described previously. Alternatively, they also map to the three information system types: information retrieval, databases, and hypertext, respectively. Table 3.1 describes the relationships between all the models we have described so far. From left to right, the models become increasingly concrete. The document model represents ideas that will become instantiated in the `Straw` model we will presently describe. Our motivation is to build an efficient and useful implementation that most accurately maps our abstract models into reality.

Now that we have a goal in mind, it is possible to build a data structure representing the document model. We are additionally interested in developing the data

Document Model	Straw Model	Information System	In Haystack
Internal state	Term ties	Information Retrieval	Any IR system
External state	Intra-document ties	Database	Lore
Relationships to other documents	Inter-document ties	Hypertext	Lore

Table 3.1: The relationship between the different models

structure within an object-oriented environment. We define *objects* to be instantiation of a particular class. A *class* is a definition of an object containing both *data members* and *methods*. Data members and methods simply correspond to variables and functions. Classes can be *extended* in a *hierarchical* fashion, where an extension *inherits* certain public data members and methods of its ancestors.

The Haystack data model is essentially a graph structure where the nodes (vertices) and ties[2] are first-class objects. First-class implies that an object can be named, exist on its own, and be passed to, and returned by functions. Both ties and nodes extend a specific class. In adherence to our namesake we call this class `Straw`. `Straw` will be discussed in great detail in Chapter 4, but for now regarding `Straw` objects as nodes will suffice.

The graph structure in Figure 3-3 is a possible (but simplistic) representation of Straws. This model shows *Document* straws connected to a variety of other objects: *publisher*, *title*, *author*, *date*, and other documents. Between straws there are three types of connections: *inter-document ties*, *intra-document ties*, and *term ties*[3]. Although in the implementation of Haystack we don not make these distinctions explictly, this categorization will help in the present discussion.

Intra-document ties represent a mapping to what we have previously called the external state or metadata of an object. For example, we can connect an author node to the document node by means of an intra-document tie. To partition different documents we will also draw a virtual boundry around clusters of straws connected by

[2]We call the connections between nodes ties to maintain consistency with the names we will use in implementing the `Straw` model. Ties are equivalent to edges (in graph terminology) or links.

[3]As stated before, ties are first class objects defined by the class `Tie`, but we'll ignore that for now to keep the model simple.

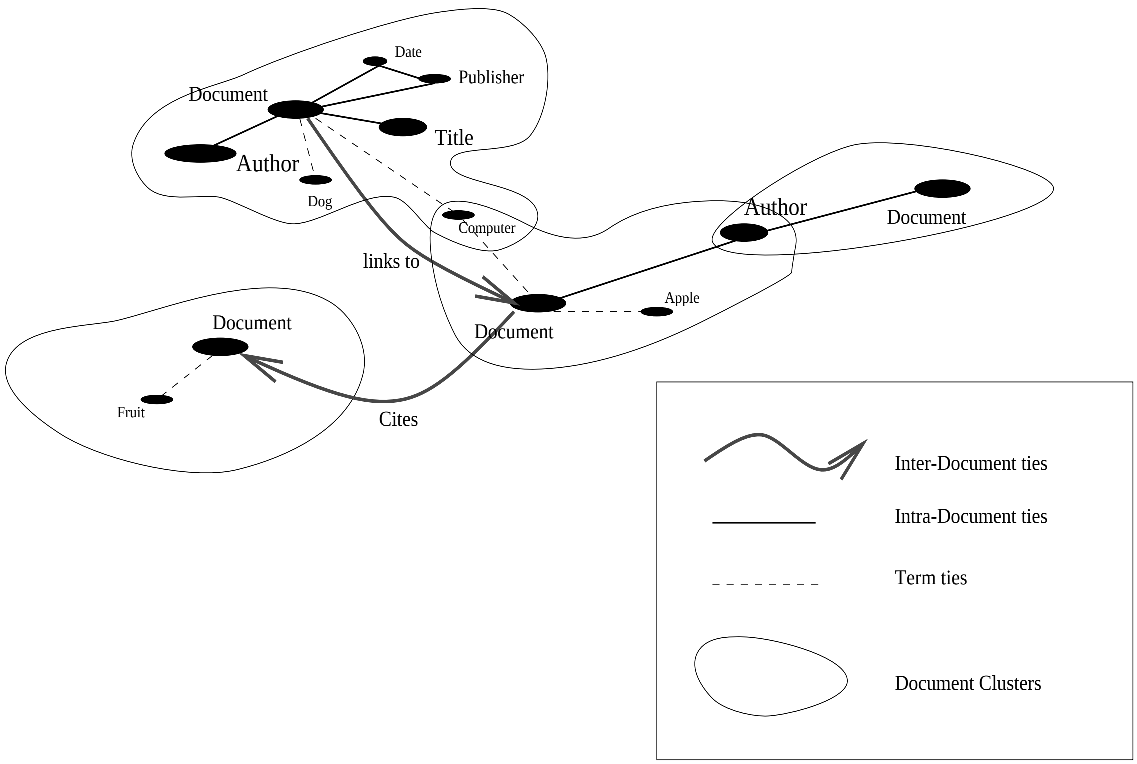

Figure 3-2: The Straw Model in Haystack

intra-document ties and call this a *document cluster*. In the figure we see one central document node for each document cluster. In Haystack we call this document node a `HaystackDocument` object. The nodes that have some associated data with them we will call a `Needle`. For example, we can have an author node/`Needle` which contains the data "David" (with the obvious implication that David is the author). We'll return to both of these objects in Chapter 4 when we describe the implementation of Haystack data model in full.

Inter-document ties represent associations between different document clusters. For example, if one document cites another, we can draw the *cites* connection between the two. Alternatively, if one document is a web page that links to another, we can use the relation, *links to*. This type of linking represents the hypertext aspect of the system.

Finally, term ties allow us to connect terms held within the document to the document cluster. That is, if the word "dog" appears in a document, we connect a node holding the word dog to to the document by means of a term tie. Note that we are not interested in the word "dog" alone, but rather the concept of a "dog." The word in the document may be "dog" but the user may remember "canine" or "pooch." This implies some semantic understanding of the content of a term straw.

With this model, we are now capable of describing documents in a way that appropriately maps the memory and document models. We propose that it is now possible to both describe objects and to query for them in a powerful manner that is consistent with the way information exists in the "real" world.

Hybrid-Search

Although the straw model described above is a powerful one, querying it is an inefficient and difficult process. Queries to this type of network structure are a complex graph-isomorphism problem. Essentially, when a user makes a query they are trying to map a sub-graph they construct mentally to the full graph structure. For example looking for a document by Bob written in January about Apples is the graph consisting of a document node connected to an author node (with the content Bob), a date node (with the content January) and a term node (with the content apples).

However, we need not despair. It is now time to re-introduce the specialized information systems described previously, and begin to build efficiency into the system.

The first task is to combine the term nodes into an efficient data structure. It is here that IR systems come to the rescue. We can now take the term nodes and merge them into the inverted file hash table which full-text IR systems are famous for. This conversion/indexing process is illustrated in Figure 3-3. In this figure we see all the term nodes merged into an inverted file, with the appropriate pointers from the file to the remaining document clusters.

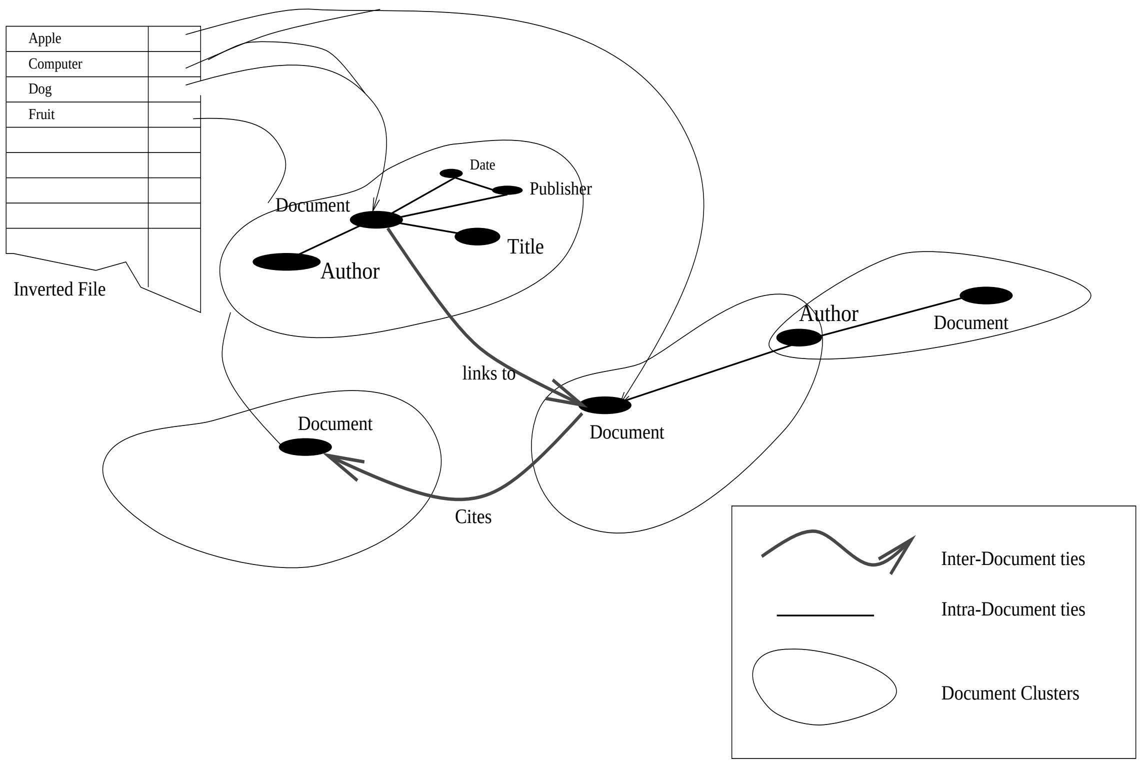

Figure 3-3: The Implemented Straw Model I

The second task involves the optimization of the remaining ties into a queryable structure. The two types of information systems, databases and hypertext systems, would seem to be the best answer. However, we recall that Lore allows us to build both types of systems into one. Figure 3-4 shows this final transformation between the abstract `Straw` structure to the concrete Lore model described previously.

The next chapter discusses in detail the construction of `Straws`, and Chapter 5 details the methods for the straw to database/hypertext converion. The indexing mechanisms built into Haystack are described completely in Appendix A.

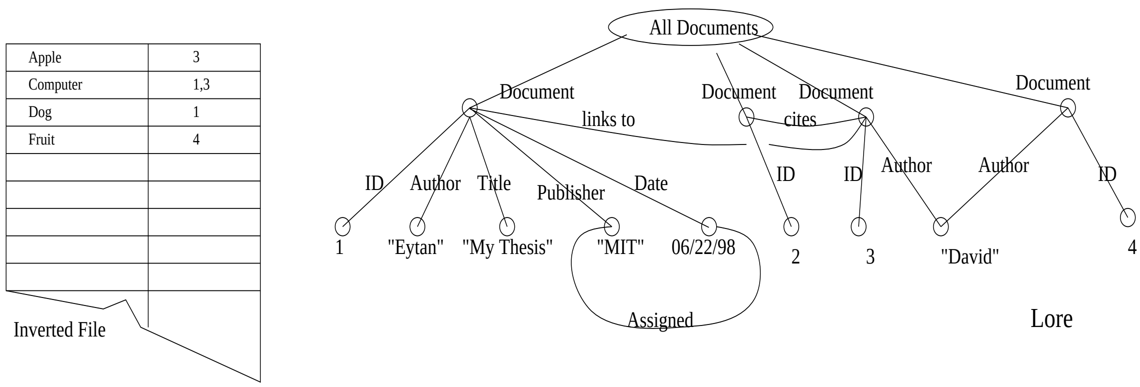

Figure 3-4: The Implemented Straw Model II

Chapter 4

The Haystack Data Model

This chapter describes the general system architecture of the Haystack Data Model. Although we will try to avoid it, this chapter will make some references to Java specific constructs. There are a number of good references that may be of assistance to the reader[1].

4.1 General Overview

The Haystack architecture can be described in two distinct parts, a Haystack Data Model (HDM) and a Haystack Service Model (HSM). The data model is the means by which we represent the user's information space, and the services (for the most part) append to or process the data in some fashion. This chapter discusses the building blocks of the Haystack data model. Specific details on issues such as persistent storage and object creation will be touched upon briefly here, but will be discussed fully in Appendix A.

As described in Chapter 3 the abstract representation of the Haystack data model (HDM) is that of a directed graph with first class edges and vertices. Vertices and edges are typed, and the typing information provides semantic information about the

[1]Specifically, Sun's Java Web site, at `http://java.sun.com`, contains a number of useful pointers and documents.

structure. For example, we can have a `bale.HaystackDocument` [2] node attached to a `needle.Location.URL`[3] node. The `needle.Location.URL` has some content associated with it (let's say http://web.mit.edu/). The implication then is that the URL of this particular `bale.HaystackDocument` is http://goose.lcs.mit.edu/. Nodes in Haystack are called `Straws`. All `Straws` are created with a unique identifier associated with them (see `HaystackID` below). The way to think about this model is in terms of associations. As discussed previously, we are attempting to model the same type of associations that correspond naturally to the document state model. With the HDM it is then possible to represent both the metadata associations between objects (i.e. the URL of a document, the author of a thesis, the date the picture was deleted), as well as the associations between documents (for example, all the documents that I cited in this paper). Figure 4-1 illustrates some of these relationships.

The HDM sample structure illustrates the use of the three main `Straw` components: bales, ties, and needles. Recall the `Straw` model described in the previous chapter depended on three types of ties connecting nodes. In implementing `Straws` in Java these tie types are created implicitly by the objects they connect. For example, to bind `Straws` into one document cluster (i.e. by *intra-document ties*) a `bale.HaystackDocument` object is created, and all `Straws` associated with the same document cluster are attached to that `bale.HaystackDocument`. *Inter-document ties* are created simply by attaching two `bale.HaystackDocuments` together. To create the *term ties* a `Needle` with text is attached to the `Bale`. To review, `Straw` is the fundamental building block for the HDM. `Bales` are collections of `Straws`. `Ties` connect two `Straws` together, and `Needles` have content.

As stated previously, ties (or as implemented: `Ties`) are also first class objects. Specifically, `Ties` are an extension of `Straws` with a number of extra features (discussed in section 4.4). Additionally, because `Ties` are first class, we can apply meta-

[2] `bale.HaystackDocument` is a special node type that is an anchor for any objects a user adds to Haystack (see section 4.6).

[3] The naming convention we will use in referring to specific Straw types is: *strawGeneralClass.StrawSpecificInformation*, where *strawGeneralClass* will be `needle`, `bale`, or `tie`. There is also an implicit `haystack.object` Java package prefix appended to all the objects. For example, the full path for the class `needle.Author` is `haystack.object.needle.Author`.

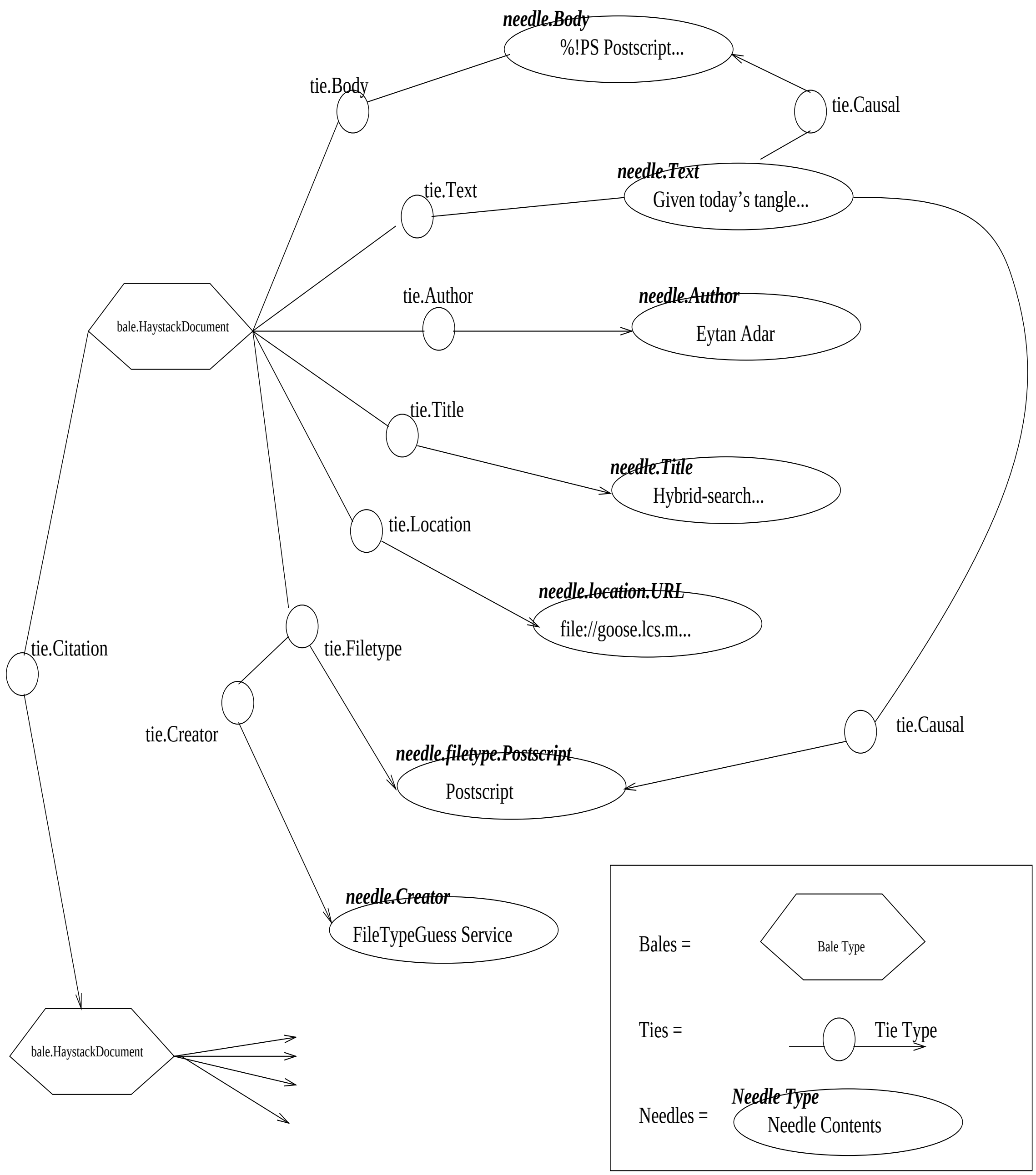

Figure 4-1: A sample `Straw` structure

data and relations to links and not just nodes. The `tie.Filetype` in Figure 4-1 was created by some service, specifically *FileTypeGuessService*. To make this fact obvious a `tie.Creator` is attached to the `tie.Filetype` object with the creator's name attached to that.

It is important to note that all objects are derived from a primary interface. A Java interfaces contains the method signatures of all functions that are required to be defined by any class that implements that interface. The primary HDM interface in Haystack is known as `haystack.object.Weavable`. All Haystack data objects are defined within the Java package `haystack.object`. Other defined interfaces include `TieWeavable`, `NeedleWeavable`, and `BaleWeavable`. This chapter will discuss each of these interfaces in turn as well as their basic implementations. We will also discuss the model by which we add new nodes to the HDM.

4.2 The Mile High View

Before we begin the discussion of specific interfaces and class implementations, it is important to have a high level view of the different objects in the HDM. We also wish to describe the naming convention for HDM objects more fully. Figure 4-2 represents the package hierarchy for HDM objects. There are three important terms we use below: interfaces, classes, and packages. *Interfaces* define (but do not implement) a set of APIs or specifications. For example, we can have an interface called `Drawable` which has the method `drawMe()`. All objects that implement `Drawable` are then expected to implement (or are forced to in Java) to implement the `drawMe()` method. *Classes* are the objects that implement any set of interfaces[4]. Finally *packages* provide a convenient means of clustering related interfaces and packages. For example, in Haystack all classes and interfaces related to the HDM reside in one package (`haystack.object`), and all classes and interfaces related to services in another (`haystack.service`).

[4]Classes implement zero or more interfaces. That is, a class need not implement a specific interface.

The four important HDM items are:

- `haystack.object.Weavable`: The basic interface which all other HDM objects implement or extend. (see section 4.3). Classes that implement this object directly usually reside in the `haystack.object` package.
- `haystack.object.TieWeavable`: This is the interface for HDM link objects. (see section 4.4). Classes that implement this object reside in the `haystack.object.tie` package.
- `haystack.object.NeedleWeavable`: This is the interface for HDM primitive objects (see section 4.5). Classes that implement this object reside in the `haystack.object.needle` package.
- `haystack.object.BaleWeavable`: This is the interface for HDM collection objects (see section 4.6). Classes that implement this object reside in the `haystack.object.bale` package.

It is possible that in the future new core packages will defined to encapsulate other functionality. However, the framework defined for HDM objects makes this a trivial step.

Object Typing

As stated previously, the type of the `Straw` represents semantic information that implies relationships between it and other `Straws`. For example, given a `bale.HaystackDocument` connected to a `needle.Filetype.Postscript`, the "bits" associated with the `bale.HaystackDocument` are assumed to be of type "postscript."

Originally, Haystack was designed with a mechanism for dynamically adding new object types to the HDM. If there was not a class definition (in the Java sense) for an object, a "generic" `Straw` would be created with the appropriate type field set to the new value. This approach, unfortunately, caused a breach of abstraction barriers and the strong typing enforced by Java. Until a Haystack type checking mechanism is implemented, a more modest approach was necessary.

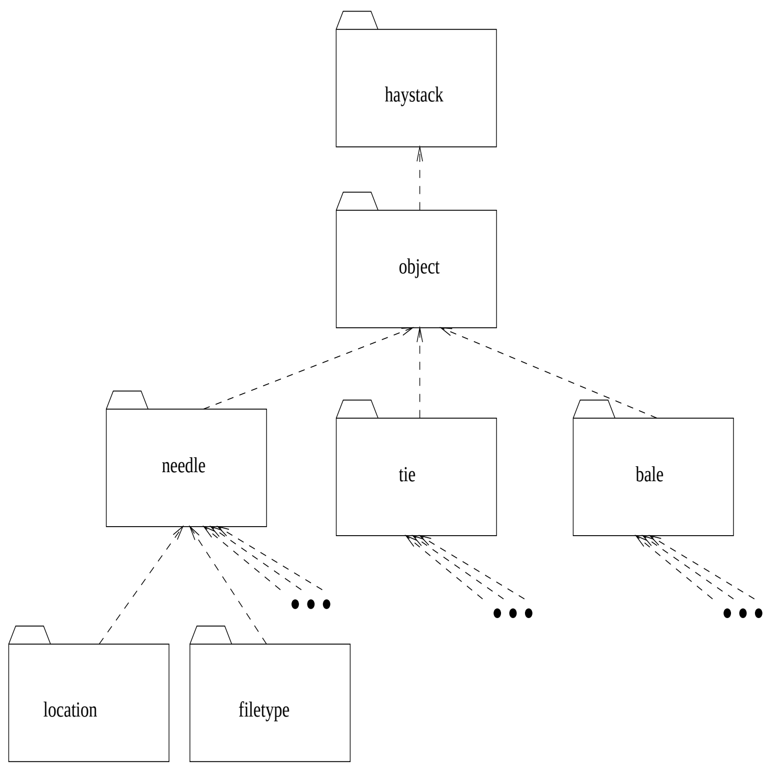

Figure 4-2: A UML diagram of the Haystack object package structure.

All objects defined within the HDM model have an associated Java implementation. The Java implementation can be a simple stub, and does not need to provide any additional functionality. It is not necessary to recompile any portions of Haystack to add new `Straw` sub-types, but it is necessary to compile the stub.

The other fundamental invariant of the HDM model is the package/class hierarchy. To extend one `Straw` class, a package is created with the `Straw` name in all lower case letters. All extensions to the original `Straw` are placed within that package. For example, in Haystack we have a generic `needle.Filetype` object (notice the upper case). Let us also assume that we have some agent (human or computer) that is interested in all Postscript files that enter into a Haystack. If `needle.Filetype` was the "deepest" we went, the agent would, inefficiently, have to check inside every single `needle.Filetype` to decide which are Postscript and which aren't. However, we already know which documents are Postscript and can make this more obvious to interested agents. To do this we create the package, `needle.filetype` (notice the lowercase). We can then create the class, `needle.filetype.Postscript`, which will reside in this package and extend the `needle.Filetype` object. The Postscript specific `Needle` can still be casted (i.e. converted) to a `needle.Filetype`, but it is now possible to tell what the `Needle` will contain by just looking at the type information of the object.

The only other requirement for extensions of the HDM is that all `Straw`s must extend an object residing one level higher in the package hierarchy[5]. Formally: $\forall$ classes `a.b...i.j.K` $\exists$ class `a.b...i.J`, where $\{a,b,c,\ldots,z\}$ are parts of a package path and $\{A,B,C,\ldots,Z\}$ are class names. In our previous example, for `needle.filetype.Postscript` to exist, `needle.Filetype` had to exist.

The `StrawType` Class

To simplify the handling of type information, Haystack includes the `haystack.object.StrawType` class. This class includes a variety of static utility functions that

[5]With the exception of the base classes, `Bale`, `Tie` and `Needle`

make conversion between types and traversal of `Straw` hierarchies a trivial operation.

The main functions provided by `StrawType` are described in Appendix C. `StrawType` also provides a list of some of the predefined `Straw`s available in Haystack in a compact representation. For example, Haystack provides a service for creating objects dynamically, and to invoke the *create()* operation the programmer must pass in the full classname for the object he wishes to create. As the class names for `Straws` tend to be long, `StrawType` eases the programmers job by creating *obvious* naming shortcuts. For example, Instead of passing in "`haystack.object.needle.filetype.Postscript`", the static value `StrawType.PostscriptTypeNeedle` is passed. The shortcuts currently defined within `StrawType` are listed in Table 4.1 (more will very likely be added in the future).

AdditionalLocationTie	Bale	BodyNeedle	BodyTie
CausalTie	CcFieldNeedle	ChecksumNeedle	ChecksumTie
DateFieldNeedle	Directory	DirectoryNeedle	DirectoryTypeNeedle
DviTypeNeedle	EmailTypeNeedle	FiletypeNeedle	FiletypeTie
FromFieldNeedle	FrommailTypeNeedle	GifTypeNeedle	GzipTypeNeedle
HaystackDocument	HTMLTypeNeedle	LastArchiveNeedle	LastArchiveTie
LastIndexNeedle	LastIndexTie	LatexTypeNeedle	LocationNeedle
LocationTie	MailfieldNeedle	MailfieldTie	MessageIDFieldNeedle
MessageIDNeedle	Needle	PostscriptTypeNeedle	RMAILNeedle
RmailTypeNeedle	SubjectFieldNeedle	SummaryNeedle	SummaryTie
TextNeedle	TextTie	TextTypeNeedle	Tie
ToFieldNeedle	UnknownTypeNeedle	URLNeedle	UUETypeNeedle

Table 4.1: Shortcuts available within `StrawType`.

4.3 The `Weavable` Interface and the `Straw` Class

Every node and link type in our system originates from the `Weavable` interface. By providing one default interface and implementation it is possible to design methods in the system that will accept arbitrary node and link types. Additionally, the `Weavable` interface provides a number of vital methods including the usual constructors, accessors and modifiers for efficiently dealing with our object model.

The primary implementation (in Java terminology) of the `Weavable` interface is the `haystack.object.Straw` class. All derivative nodes and links extend (again, Java terminology) the `Straw` class. The `Straw` class contains three important data members, two `StrawTies` objects and a `HaystackID` object. One of the `StrawTies` objects maintains the pointers in the forward direction (i.e. `Straw`s pointed at). The other maintains pointers to the back objects (i.e. `Straw`s pointing to). We'll return to this shortly.

The `Straw` class is extended in a variety of ways, but is never instantiated on its own (that is, we never have an object of type `Straw` in a HDM graph). The key methods specified by the `Weavable` interface (and implemented by `Straw`) are described in the subsequent sections.

The `StrawTies` Class

The two `StrawTies` objects provide a one-to-many mapping between keys (labels), and other Haystack objects. This essentially provides us with a method to maintain the list of "edges" that any vertex is attached to. `StrawTies` therefore represent the function f, where $f(s) = \{$`TieWeavable` objects$\}$, where s is a string label.

`StrawTies` are internally implemented as two `Hashtable`s. One table maps a key to set of pointers (in the Java sense), and the other contains the mapping between a key and the `HaystackID`s (see below) of the objects. In order to save the HDM structure persistently we serialize[6] (see Section A.3.5) each `Straw` object to save on disk. Because the Java serialization mechanism will save not only the object being serialized but any objects that the serialized object points at, it is necessary to break apart `Straw`s before saving them. Maintaining the two hashtables is an optimization that allows us to break apart and reassemble `Straw`s easily.

We enforce the following invariant: let each vertex of the HDM graph (the `Straw`) possess a function, f_1 that accepts as input some key, k and produce a set of connected edges, A, labeled k. Let f_2 map some key j to a set of identifiers, B. Then $\forall$ keys k,

[6] Java serialization allows for an object, and its state, to be converted into a byte stream which can be written out to disk. These bytes can later be re-read and the object is reassembled

j, where $k = j$, $\exists$ a in A and b in B, s.t. a.getID() $= b$. Simply stated, the hashtable holding HaystackIDs as values can be used to produce the hashtable holding Straws as values.

StrawTies provide a subset of the features of the Java Hashtable. The function getVal(label) returns the Vector of values with the given label. To place a new element into the table, put(key,value) is used. Alternatively, remove(...) and removeKey(...) allow for pointers (i.e. edges) to be removed. getAllLinks(), getIDs(), isEmpty(), and toString() perform obvious functions.

4.3.1 Straw Constructors

Straws are created by one central process (HsObjectCreatorService, see Section A.3.1). This process ensures that the constructor for Straws is invoked properly. Specifically, this means generating a unique identifier for the Straw in the form of a HaystackID.

The HaystackID Class

Uniquely identifying nodes within the graph would be highly useful for a number of reasons. By tagging Straws with a unique identifier, services could quickly access different portions of the graph and the keeping track of the state of the Straws within Haystack would be easier.

In order to guarantee that our identification scheme was scalable enough, the actual identity information is encapsulated inside a HaystackID object. Currently, the internal representation for HaystackIDs is the java.math.BigInteger class. BigInteger allows for arbitrarily large integer values, which is sufficient for maintaining uniqueness within a give Haystack. In the future, when Haystacks interact together, additional user-specific information will need to be added.

The HaystackID class currently provides for a number of operations to extract the "low level" representation of the identifier, but use of these should be limited, and possibly prohibited in the future. The class also defines a number of comparison operators for judging equality between two identifiers. Because each `Straw` is assumed to be unique, the test for equality between two `Straw`s is done simply by comparing the two `HaystackID`s.

4.3.2 `Straw` Modifiers

The main modifier operations provided by `Weavable` are systematic methods of connecting and disconnecting `Straw`s.

For directly connecting a `Straw` to a `Tie`, the `Straw` class provides the methods `attachForwardLink(...)` and `attachBackLink(...)`. These functions directly add the `Straw` to the appropriate `StrawTies` object.

For connecting two objects that have some relationship to each other, say a `bale.HaystackDocument` and a `needle.Title`, the HDM model calls for the connection to be done by means of a `Tie`. This is true for a connection between a `Bale` and a `Needle`, between two `Needle`s or between two `Bale`s. Attaching a `Needle` (say a comment about a `Tie`) to a `Tie`, is done by means of a third `Tie`. This is due to the first class nature of edges in the HDM graph.

In designing an interface for dealing with the HDM objects it became obvious that a substantial amount of the time a programmer wants to connect two objects directly without dealing with the intermediate `Tie`. To facilitate this, `Straw` provides two functions `attachForwardObject(...)` and `attachBackObject(...)`. Both functions generate the intermediate `Tie` object and perform all the necessary adjustments to connect the two `Straw`s. `Tie`s are generated in a somewhat "intelligent" fashion. That is, a `Tie` is selected that most closely matches the object it is connecting. For example, if a `needle.filetype.Mail` object is to be attached to a `bale.HaystackDocument`, the `attachForwardObject()` operation will find the first valid object in the tie package that most accurately represents the needle. The function first verifies that the class `tie.filetype.Mail` exists. If the check fails, the function moves one level back in the type hierarchy and attempts to create a `tie.Filetype`. If after crawling up the hierarchy, no match is found, a basic `Tie` object is created to connect the two `Straw`s.

This lookup process is an optimization that simplifies the process of adding new `Straw` types. It is not necessary to code a matching `Tie` class for every `Needle` and `Bale` object. However, the creation of new types of `Straw` is simplified and/or automated this process will no longer be necessary and a one–to–one correspondance will exist between the connected `Tie` and `Needle` (i.e. for a `needle.filetype.Mail` there will be a `tie.filetype.Mail`).

`Straws` also provide a mechanism to detach links (`detachForwardLink(...)`, `detachBackLink(...)`). Use of these should be avoided as the HDM is intended to reflect changes over time, but they are provided for adept programmers who require such modifications to the HDM graph.

4.3.3 `Straw` Accessors

The `Straw` class provides numerous functions to traverse the HDM graph space. `forwardLinksOf()` and `backLinksOf()` return all edges propograting from the current `Straw` in the forward and backward directions respectively. Similarly `getForwardLinks(label)` and `getBackLinks(label)` return the links with a given label (i.e. of a specific type). For example, in Figure 4-1, invoking `forwardLinksOf()` on the topmost `bale.HaystackDocument`, the set: {`tie.Body`, `tie.Text`, `tie.Author`, `tie.Title`, `tie.Location`, `tie.Filetype`} is returned, or more specifically, the set of pointers to these objects is returned.

The functions `getAllForwardObjects()` and `getAllBackObjects()` skip the intermediate `Tie` objects between the current `Straw` and what is attached to it and returns all the “destination objects.” `getAllBackFilteredObjects(label)` and `getAllForwardFilteredObjects(label)`, filter the `getAll...Objects()` function and return only objects with the given label. `getLastForwardObject(label)` and `getLastBackObject(label)` filter the edge names and return most recent (i.e. the last) edge with the given label. The function `depthFirstSearch()` returns a list of objects traversed in a depth first search originating at the current `Straw` up to a given depth. In Figure 4-1, `getAllBackObjects()` invoked on the `needle.filetype.Postscript` node will return the set: {`bale.HaystackDocument`,`needle.Text`}.

Straws also provide methods to access the ID of the object, get the "supertypes" of the Straw, and to get the "type" of the current Straw: getID(), getSupertypes(), and getTypeString() respectively.

Finally, Straws provide functions to generate a string representation of the object (toString()), as well as a method of generating a text representation (toText()) that we will index in the Information Retrieval System. So for a needle.mailfield.Tofield containing "eytan@mit.edu," toText() will simply return "eytan@mit.edu." On bale.HaystackDocument objects we don't necessarily have text of their own we can call getIndexable() which will get the text of objects in the "collection" (i.e. the document cluster).

4.4 The TieWeavable Interface and Tie Class

The basic link/edge interface is defined in TieWeavable, and implemented in the haystack.object.Tie class. The TieWeavable interface extends the Weavable interface, and similarly the Tie class extends the Straw class. This means that some functions are overridden and functions unique to links are implemented.

TieWeavable provides the following methods: setForward, setBack, getForward, and getBack. Ties are different than standard Straws in one major way: in their jobs as links they connect two, and only two nodes. While it is possible to add fields to Ties (such as needle.Creator, or needle.Comment), Ties maintain a single back pointer and forward pointer that is seperate from the StrawTie objects. This idea is represented in Figure 4-3. Ties override the get...Links() functions of Straw so that they will not only include the links held within their StrawTies but also the link held within their "specialty" pointer field. In the sample figure, a Straw (1) points another Straw (3) by means of a Tie (2). The label used within the two Straws matches the type of the Tie. Similarly, the Tie (2) is connected to another Straw (5) by means of yet another Tie (4).

The benefit of the single back/forward pointer property is that it disambiguates the intention of the Tie. It is easy to determine what nodes the link connects and

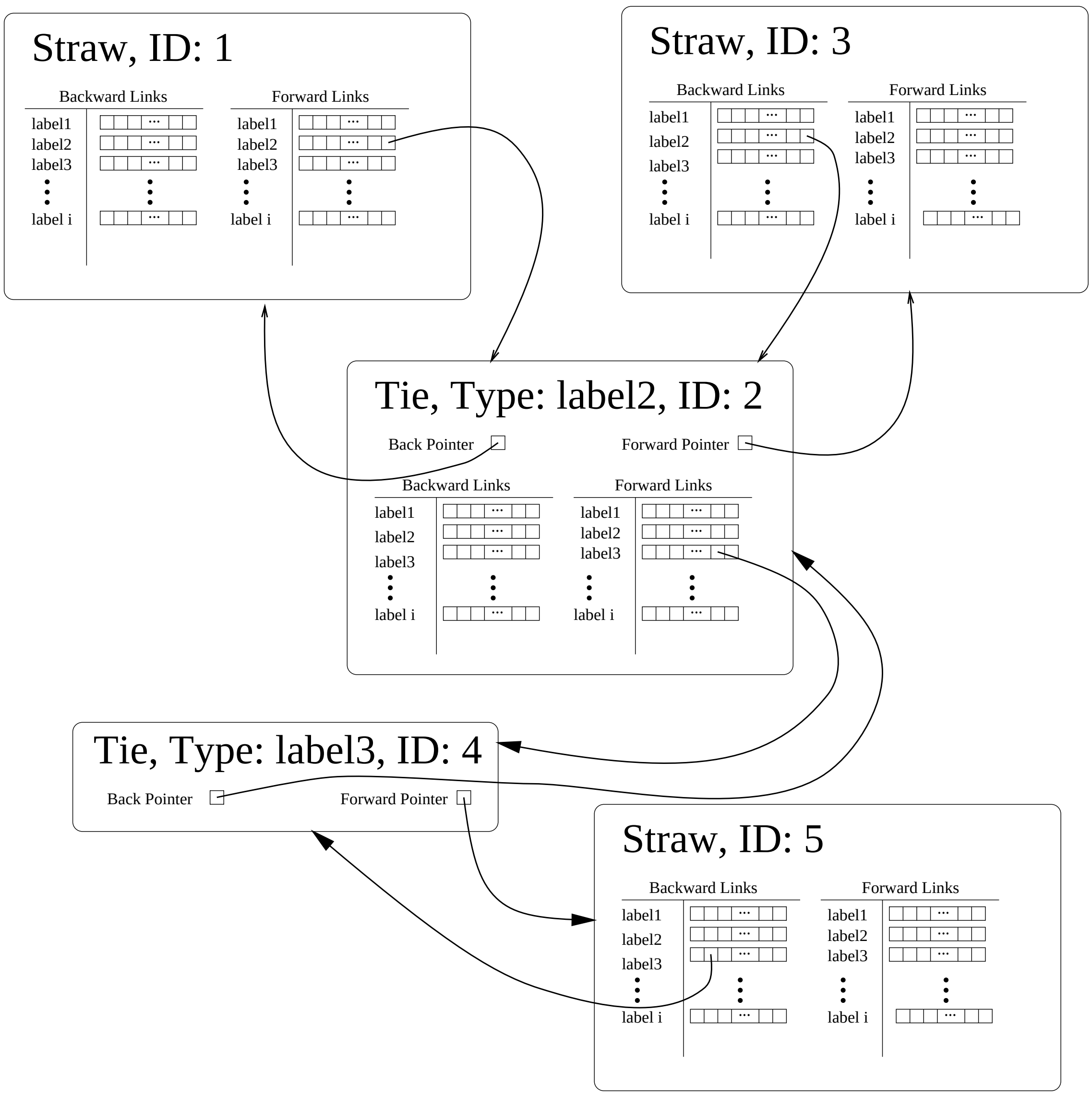

Figure 4-3: A sample `Tie` cluster.

what nodes are associated with the link in some other way (as before, the example is a comment on the link).

The `tie.Causal` Class

There are a number of special `Tie` extensions implemented within Haystack, the most important of which is `tie.Causal`. The class `tie.Causal` does not implement any additonal functionality, but we are interested in its use for the sake of the implicit information it provides in its type.

The use of `tie.Causal` is fairly simple. If a `Straw` A was created because of a `Straw` B, then we connect the two (starting at A) with a `tie.Causal`. For example, if we have a `needle.Location` that points at some file somewhere, and we retrieve the file and create a `needle.Body` to hold the bits, we draw a `tie.Causal` connection between `needle.Body` and `needle.Location`. This shows that the text exists because of the `needle.Location`. More examples of this are illustrated in Figure 4-1.

4.5 The `NeedleWeavable` Interface and `Needle` Class

An important issue that we have so far glossed over involves the content of `Straws`. `NeedleWeavable`, and its implementation `Needle`, provides us with the mechanism for holding actual “data.” One could say that these `Needles` represent atomic values within the Haystack graph. However, calling them “atomic” is slightly misleading. `Needles` have the same functionality as `Straws` in their ability to be linked from and to by other `Straws`. They are atomic in that they provide wrappers around pre-existing data (such as files, database entries, web pages, etc).

This difference should be obvious to those with knowledge of database systems. However, in Haystack we extend the concept of database primitive from the usual set of strings, ints, reals, etc. The Haystack data model primitive provides a wrapper around any conceivable data type. For example a `needle.Text` represents a wrapping around a particular bit of digital text. An `needle.Date` may wrap around a `long` representation of the date.

To access this information, `NeedleWeavable` defines the `getData()` method. The return type of `getData()` is of the Java `Object` class. Therefore it is up to the programmer to cast the `Object` to a more specific type. So for example, `postscipt-Needle.getData()` may return a String, or an integer, or some other primitive type. The `setData(...)` operation of the `Needle` will also take as argument an `Object`. However, the `Needle` programmer may throw a `NeedleDataException` if the data is not castable to the required format. For example, if the `Needle` is supposed to hold an integer, and is passed the string: "foo bar mumble," a `NeedleDataException` is throw.

It is our hope that this flexibility will not cause great difficulties to the programmers and users of our system. By properly specifying the behavior of all nodes we hope that the added functionality will allow for a richer set of services and applications.

The `Promise` Class

The data of the `Needle` is also loaded into memory at the same time as the `Needle` object, we do not want (for example) a 10Mb+ Postscript file loaded into memory. Instead we create a `Promise` class. The `Promise` class holds one function, `fulfill()` which returns a `Reader` (a Java stream type). It is the programmers job to implement the details of the `fulfill()` class for their particular "promise."

As an example, consider the case of some user creating a `needle.Body` that is supposed to hold a large Postscript file. The file resides on disk somewhere. Instead of setting the data field within the `needle.Body` to the bits of the Postscript file, we can set it to a special `Promise` class that holds the name of the file location. When `getData()` is invoked on the needle, the `Promise` is returned, and when `fulfill()` is invoked on the `Promise` the bits of the Postscript file are dynamically loaded from disk.

Some `Promise`s are designed to do "actual" work. For example, a `Promise` held within a `needle.Text` may actually extract the text from some other location on-the-fly. Re-running the `Promise` is highly inefficient, so in Haystack we provide the means for caching promises (see Section A.3.6).

4.6 The `BaleWeavable` Interface and `Bale` Class

In the initial design of the Haystack data model we have made the decision to include an interface for nodes that represent collections. The only obvious use for this `Straw` class (aside from the implicit typing information) is the ability to override certain `Straw` functions to provide ordering. For example, in Haystack, when a query is invoked a `bale.QueryResult` object can be generated pointing at the `Straws` that satisfy the query. What we note, however, is that query results are usually ordered based on some relevance metric. In invoking the `getAllForwardObjects()` for the `bale.QueryResult` object, we want the ordering maintained. This is achieved by overriding the `getAllForwardObjects()` to order the data based on some other information (i.e. a list of results and scores).

The `bale.HaystackDocument` Class

There is one `Bale` extension that is truly a vital building block for the HDM graph. As stated previously, we would like to group all the properties of a given document inside some "cluster." This is achieved by the use of the `bale.HaystackDocument`. This class does not implement any additional functionality beyond `Straws`, but serves the important purpose of marking document clusters in Haystack.

Chapter 5

Haystack and Lore Integration

Now that the core Haystack frameworks have been established, we can describe the mechanisms by which the HDM can be transformed into Lore for persistent storage and the ability to query.

5.1 Representing HDM objects in Lore

There are a number of ways in which we can represent the HDM inside the OEM model. A simple variation is provided here. It is sufficiently expressive and concise to efficiently described the HDM graph inside the Lore OEM model.

Figure 5-1 is a small OEM representation of a `bale.HaystackDocument` connected to a `needle.Body` by means of a `tie.Body` object. Although the figure looks unnecessarily complex, it is in fact a very compact representation.

The Lore OEM structure created for Haystack starts with a single pointer into a complex object. The label for this pointer is `Haystack`[1]. The generated OEM file begins with: "`<Haystack::Haystack {`" and ends with: "`}>`".

Because the period character has significance in Lore files, the *OEMStrawType* for any `Straw` is modified to use the "_" character. For example, `bale.HaystackDocument` becomes `bale_HaystackDocument`. The *HaystackID* corresponds to the `Straw`'s ID,

[1]What we depict here is the process of generating a complete OEM structure from an existing HDM graph. Incremental modifications to an existing Lore structure are described in Section 5.3

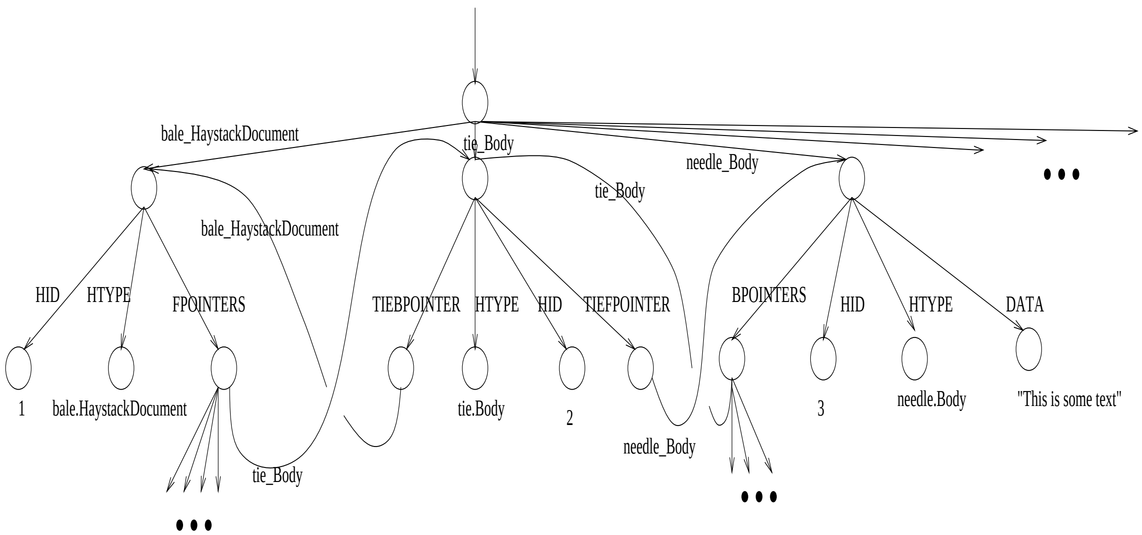

Figure 5-1: An OEM diagram for a small portion of an HDM cluster

```
<IDHaystackID:: OEMStrawType {
          <HID "HaystackID">
          <HTYPE "StrawType">
          <FPOINTERS {
                    <&IDForwardPointerID_1 >
                    <&IDForwardPointerID_2 >
                    <... >
                    <&IDForwardPointerID_n >
                    }>
          <BPOINTERS {
                    <&IDBackPointerID_1 >
                    <&IDBackPointerID_2 >
                    <... >
                    <&IDBackPointerID_n >
                    }>
          ...additional Tie/Needle information...
}>
```

Figure 5-2: An OEM representation of a `Straw`.

and $BackPointerID_n$ and $ForwardPointerID_n$ correspond to the HaystackID of the n^{th} back and forward pointers held within the Straw's StrawTies.

The first line of the code creates two pointers to a complex object. The first pointer is the globally named pointer "ID*HaystackID*" (i.e. ID followed by the Straw's HaystackID value). The second pointer emanates from the Haystack complex OEM object (see Figure 5-1) and is labeled with the type of the object. From this complex node we draw four additional pointers: HID which connects to an atomic object holding the HaystackID for the Straw. The HTYPE label points at a node holding the type of the Straw. The two pointers FPOINTERS and BPOINTERS are created if there are any backward or forward pointers emanating from the Straw. These objects are complex and point at other named Straw objects in the OEM graph.

When creating the OEM data for a Tie, two additional lines are added to the Straw definition in Figure 5-2 (in the space for additional information):

<TIEBPOINTER &ID*TieBackPointerID*>
<TIEFPOINTER &ID*TieForwardPointerID*>

TieBackPointerID corresponds to the HaystackID of the back pointer of the Tie (i.e what is obtained by calling getBack()). TieForwardPointerID is the same thing in the forward direction.

For a Needle we add the following to the Straw OEM code:

<DATA *needleData*>

The *needleData* field simply corresponds to the data held by the Needle. If the data held by a Needle is a String, the *needleData* is encapsulated in a quotation marks (so that Lore knows the data is a string). If the data is some number, it does not need to be enapsulated in quotes.

The Straw class in Haystack implements the basic functionality that generates the code specified in 5-2. To obtain the OEM representation of a Straw, the generateOEM() method is called on the Straw. The Tie class extends the code generated by Straw and adds the appropriate fields specific to objects in the tie package (i.e. TIEBPOINTER and TIEFPOINTER). Similarly, The Needle class extends the code generated by `Straw` by adding the correct `DATA` information. By default the data is assumed to be text. Specific `Needles` can override the code generation method to produce an OEM field of the appropriate type (for example, a `needle.Date` will produce an integer `DATA` field).

5.2 The `HsOEMGenerator` Class

In theory if a user has always had Lore attached to their Haystack it should never be necessary to regenerate the OEM graph. However, in the case where the data either becomes corrupt, or the Lore database is added to Haystack at some later time, a service is provided to regenerate the OEM graph from scratch. `HsOEMGenerator` is a simple utility service that is invoked with the `generateOEM()` procedure. This procedure checks to see which location is specified for the OEM file in the `HsConfigService`. It then creates the necessary header information and iteratively retrieves all `Straws` in the `HsPersistentObjectService`. For each of these `Straws`, the `HsOEMGenerator` invokes the `generateOEM()` method and saves the resulting OEM structure to disk.

A user may then use the OEM loading facility in Lore to reload the new data into the Haystack Lore database.

5.3 The `HsLore` Class

The `HsLore` class is a simple event listener class that expresses interest in `ObjectEvents`. This implies that `any` change in the HDM trigers an event. Once `HsLore` is notified of an event, it will invoke the `generateLorel()` method of the `Straw` that caused the event. The generated string corresponds to the Lorel query that will update the database to the new state of the object.

Upon initialization `HsLore` opens a stream connection to the Lore process executing either locally on a remote machine[2]. Once the Lorel update code has been generated by the `Straw`, `HsLore` will pass the query into Lore.

[2]This is necessary as Lore is not currently portable to all machines

Unfortunately, at the time of this writing the Lorel update language has not been completely defined. The following is the Lorel syntax for updating the Lore database for `Straws`, `Ties`, and `Needles`[3].

Lore allows us to refer to the globally named pointer directed at the current Straw by means of the `Global!`*Identifier* argument. For example, if a `Straw` with the `HaystackID` 253 is loaded into Lore, the Lore OEM object corresponding to that `Straw` can be accessed by `Global!ID253`. The Lorel to update a generic `Straw` with HaystackID x and type y is of the form:

`Global!ID`x `:= oem(HID:‘‘`x`’’, HTYPE:‘‘`y`’’,`

`FPOINTERS:oem(ID`$_{f1}$`,ID`$_{f2}$`,...,ID`$_{fn}$`),`

`BPOINTERS:oem(ID`$_{b1}$`,ID`$_{b2}$`,...,ID`$_{bn}$`));`

The values of ID_{f1} – ID_{fn} are simply the names of the global Lore pointers to the the forward objects of the `Straw`. For example, if the `Straw` has a forward pointer to some object with the `HaystackID` 494, then one of the ID_{fi} values would be replaced with the string `ID494`.

For a `Tie`, the form of the Lorel query is very similar:

`Global!ID`x `:= oem(HID:‘‘`x`’’, HTYPE:‘‘`y`’’,`

`FPOINTERS:oem(ID`$_{f1}$`,ID`$_{f2}$`,...,ID`$_{fn}$`),`

`BPOINTERS:oem(ID`$_{b1}$`,ID`$_{b2}$`,...,ID`$_{bn}$`),`

`TIEFPOINTER:ID`$_{tfp}$`, TIEBPOINTER:ID`$_{tbp}$`);`

Where ID_{tfp} and ID_{tbp} are the global OEM identifiers for the forward and backward pointers of the `Tie` respectively. Finally, an update for a `Needle` object is of the form:

`Global!ID`x `:= oem(HID:‘‘`x`’’, HTYPE:‘‘`y`’’,`

`FPOINTERS:oem(ID`$_{f1}$`,ID`$_{f2}$`,...,ID`$_{fn}$`),`

`BPOINTERS:oem(ID`$_{b1}$`,ID`$_{b2}$`,...,ID`$_{bn}$`),DATA:’’data’’);`

Currently, we do not provide a simple method for the deletion of objects as Lore does not provide a mechanism to delete a global name. However, it is possible to disconnect the `Straw` from the total OEM graph. If we desire, for example, to remove the `Straw` with ID 4565, the following query would work: `Haystack =- ID4656`.

[3]Based on the most recent language specification available at the time of this writing.

Chapter 6

Conclusions

In this thesis we have created a mechanism by which users can represent documents in a natural and expandable fashion. We have determined the nature of documents in terms of document state, and creating a data model to represent this state within Haystack. Furthermore we have shown the system context in which this data model resides.

Through Haystack a user is now capable of indexing documents in a powerful fashion. Haystack heuristically extracts a variety of information properties from the document and indexes those through a number of means.

We have also shown that information systems individually are incapable of answering the full set of questions a user may ask. Databases provide us with "what you ask for is what you get" functionality. Information Retrieval systems attempt to extract semantic meaning in textual documents. Finally, associative information systems facilitate the creation (both manual and automatic) of associations between documents. In combination these information systems are capable of handling the full spectrum of user queries.

In this thesis we have described mechanism to convert Haystack's internal data model to each of these systems.

6.1 Future Work

The one important aspect of functionality within Haystack that we have failed to address is the actual mechanism by which queries are made. Although there are independent methods to query each information system there is currently no uniform method to simultaneously query all of them.

Ideally, a natural language solution will one day exist that will be able to dispatch different aspects of the query to different information systems. For example, the question, "What documents about apples that I read yesterday?" would be split into two different questions: "find me all documents about apple" and "find all documents read yesterday." The first question is obviously based on some semantic considerations and so it will be dispatched to an IR system. On the other hand, the other question is inherently a database type question and can be dispatched to Lore. Result combination is also an open issue. Ideally, the user should be able to scale the results of the different information systems in some way.

The second alternative for querying utilizes the Lore DataGuide model. By taking a snapshot of the current database schema and representing it graphically a user's can simply fill in different parts of the graph and the appropriate query will be generated. A special box can hold free text queries (those intended for an IR system).

Regardless of the exact query mechanism, the hybrid-search system proposed here has the potential to solve a much greater variety of problems than any independent information system.

Appendix A

The Haystack Service Model Architecture

The bulk of functionality within Haystack is implemented by objects in the Haystack Service Model (HSM). Abstractly, Haystack can be viewed as standard three-tiered architecture consisting of three different layers, a user interface layer (the client), a server/service layer, and a database. Figure A-1 represented the layers (and components within the layers) graphically.

This chapter primarily concerns itself with components of the service layer. The Haystack service layer consists of a number of services and the Haystack Data Model (HDM) objects running within one `HaystackRootServer`.

Figure A-1 illustrates some of the relationships between services in Haystack. As stated previously, services run within the context of one `HaystackRootServer`. There are services that provide interfaces between the user and the indexing and archiving subsystems. There are services that provide utility functions for other services. For example, a name service allows for communication between other services, and a resource control service provides the mechanism for the locking of arbitrary named resources. There are services within the `HaystackRootServer` that "listen" for changes in the HDM. When those changes happen, the services are notified and they act upon the data.

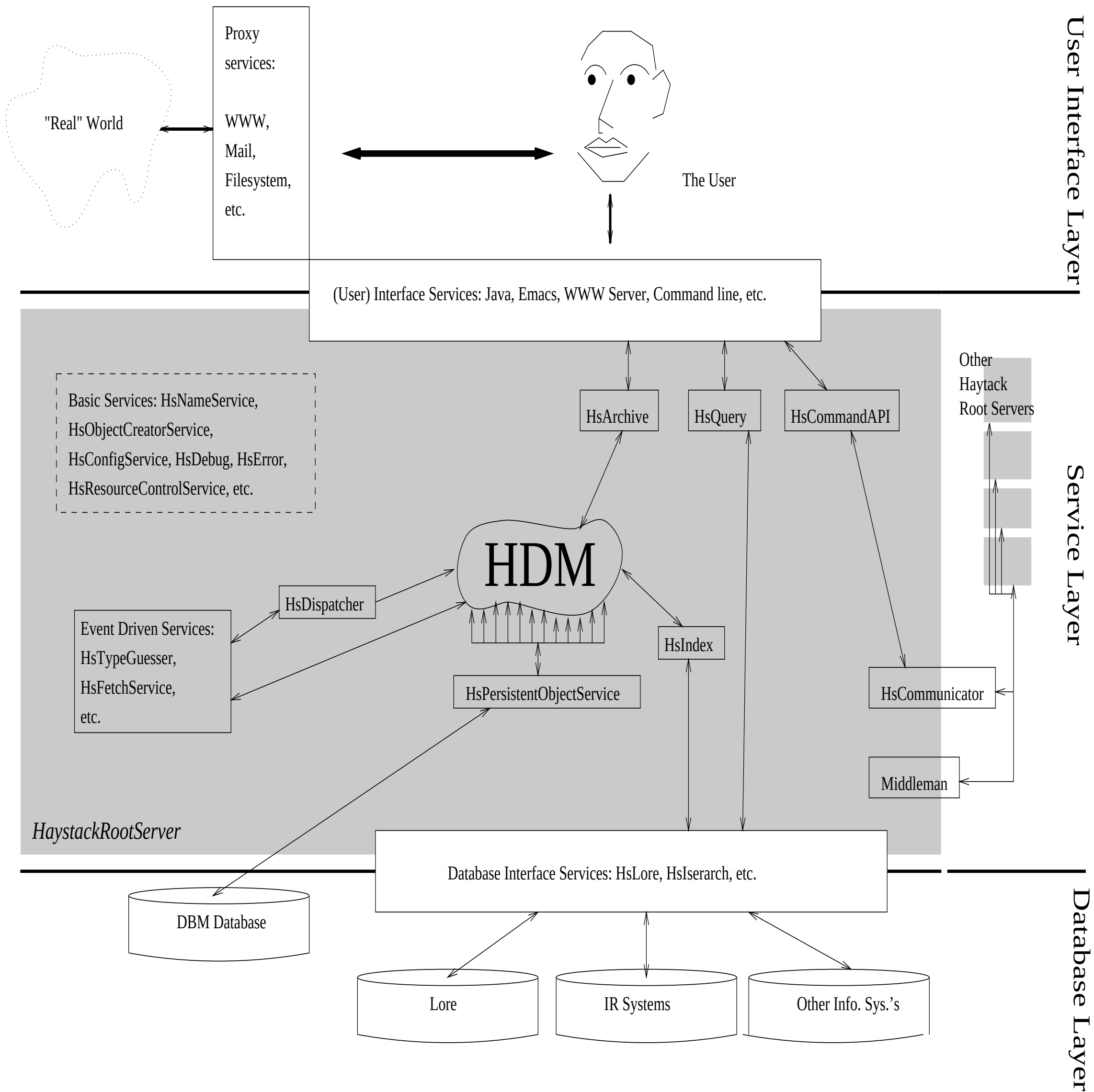

Figure A-1: Abstract inter-service communication model

A.1 Service Architecture

For those with an understanding of CORBA[31] the internals of the service layer should be fairly straight-forward. The Haystack service model utilizes an architectural framework that borrows a number of important ideas from the OMG specifications for CORBA.

In the Haystack notation, just as in CORBA, services are merely objects that act as clients, servers, or peers. A service encapsulates a specific functionality behind a well defined interface. This functionality can be readily invoked from any other service. A session can be represented as follows (see Figure A-2):

1. In the first step, service A and B are both loaded (i.e. they are instantiated) in a `HaystackRootServer` (see Section A.2). In Figure A-2 both use the same `HaystackRootServer`, but as we shall see, this isn't necessary. Both services invoke the `register(...)` method of the `HsNameService` (see Section A.2.2), with an abstract version of their name (represented by the `ServiceName` class).

2. In the second step, service A wishes to communicate with service B, so A will generate the `ServiceName` for B, and submit the name to `getService(...)` request to the `HsNameService`. The `HsNameService` will respond with a pointer to the instantiated version of B running within the `HaystackRootServer`.

3. In the final step, service A can now communicate with B. For now, it is assumed that A knows which methods B has made public. In the future services will be implemented as JavaBean objects [25] that define an introspection method (so that A can dynamically decide how to call B).

A.1.1 Service Types

The services loaded into the `HaystackRootServer` (discussed in Section A.2) fall into a number of categories:.

- *Basic services* (Section A.3) provide a number of vital functions necessary for maitaining the basic functionality of the Haystack. These include:

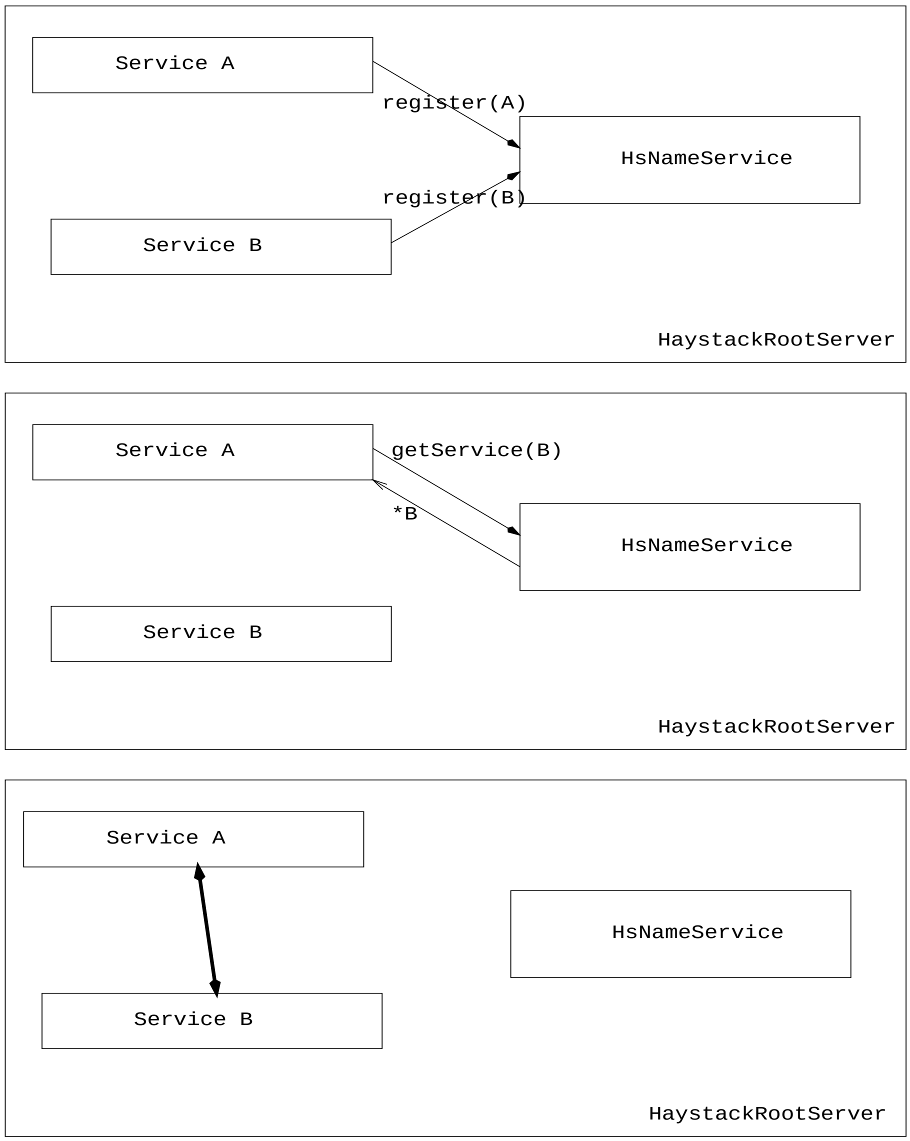

Figure A-2: Abstract inter-service communication model

- `HsNameService` (Section A.2.2) is a built-in naming service to initiate inter-service communication.
- `HsObjectCreatorService` (Section A.3.1) is a service that dynamically creates new Haystack data objects.
- `HsDispatcher` (Section A.3.2) is a service that defines relationships between Haystack data objects and services that are interested in them.
- `HsConfigService` (Section A.3.3) loads other services based on some user configured options.
- `HsResourceControlService` (Section A.3.4) is a service to provide locking on abitrarily named resources.
- `HsPersistentObjectService` (Section A.3.5) provides permanent storage of HDM objects.
- `HsCache` (Section A.3.6) implements a mechanism for the `Promise` caching described in the previous chapter.
- Other base services (Section A.3.7) are included for debugging and logging a variety of messages (including `HsDebug`, `HsMessage`, and `HsError`).

• *Interface services* (Section A.4) provide the mechanisms for external processes to talk to Haystack. These include: socket/stream communication services, an RPC service, and a WWW server.

  - `Middleman` (Section A.4.1) provides an RPC mechanism for inter-VM communication.
  - `HsCommandAPI` (Section A.4.2) A service to allow dynamic loading of "commands" that are made available to non-Java external processes.
  - `HsCommunicator` (Section A.4.3) In order to interact with Haystack from a non-Java program we provide a socket based protocol for communicating (on a limited basis) with Haystack services.

  - `HsCommandLine` (Section A.4.4) A simple interactive command line interface tool into Haystack. This service can also be loaded in a seperate virtual machine for access into Haystack as a client tool.

- *Event driven services* (Section A.5) provide the important function of responding to data as it enters into — and changes in — a user's Haystack. After expressing interest in specific graph structures in the Haystack these services are notified of changes to those graph structures. Once invoked they are free to augment the Haystack data structure in some fashion. Examples of this type of service are the text convertors and field extractors.

- *"Data manipulation" services* (Section A.6) include services for general control that allow for archiving and querying:

  - `HsArchive` (Section A.6.1) allows new documents to be archived into a Haystack.

  - `HsIndex` (Section A.6.2) derives textual information for documents and "pushes" the information into an Information Retrieval system.

  - `HsQuery` (Section A.6.3) faciliates queries between user interface components and Information Retrieval system.

- *Data layer (interface) services* provide the interfaces necessary to communicate with the variety of information systems that Haystack uses (currently IR systems and Lore). A sample IR interface is described in the context of `HsIndex` (Section A.6.2). The interface to Lore is decribed in Chapter 5.

- *Automatic services* work in their own thread space within the `HaystackRootServer` doing pre-programmed tasks. These include services that pre-fetch new documents, or recreate the Haystack indicies. Currently, there are no services that qualify as automatic services, but facilities for their creation have been included in the Haystack code (including timer objects that "wake up" services at specified intervals).

- *Observer services* actively attempt to watch the user interact with various external processes. For example a WWW proxy will observe the user's interactions with their web through their web browser[1].

A.2 The `HaystackRootServer`

The Haystack system depends on the instantiation of the `HaystackRootServer` object. This object performs much the same functionality as the object request broker (ORB) does in CORBA. We decided against a CORBA model due to the complexity of the CORBA architecture and the existence of very few, if any, full and public domain implementations. The `HaystackRootServer` object automatically starts itself with a number of services.

The `HaystackRootServer` is instantiated in one of two modes, root, and non-root. A user is always assumed to be running one `HaystackRootServer` running in root mode. The server running in root mode causes a number of event driven services and the HDM objects to be loaded automatically into the VM's memory space. HDM objects are currently loaded only into a root `HaystackRootServer` as Haystack does not currently support a concurrency model robust enough to allow for manipulation of HDM structures outside of the main `HaystackRootServer`'s scope. There are various pre-existing solutions that describe the implementation of distributed object management that may be suitable for intergration into Haystack in the future.

A non-root `HaystackRootServer` is used for the purposes of loading services that need to run outside the root-VM. For example, if a user has mutiple computers and wishes to archive the files on both into one Haystack, they can run a non-root `HaystackRootServer` with an archiver service on one machine and a `HaystackRootServer` in root mode on the other. All data then gets aggregated from the non-root to the root `HaystackRootServer` and the data from both machines is indexed together. For services that need to decide if they are running under a root or non-root context,

[1]At the time of this writing a WWW proxy service was in the process of being developed, but formal specifications are not yet available

HaystackRootServer provides a static method, amIRoot() to answer that question.

To successfully create a HaystackRootServer, a user must both instantiate and initialize the HaystackRootServer object. Instantiation is done through the standard new operator, and initialization through the init() procedure of the HaystackRootServer. It is important to note that the HaystackRootServer utilizes a variant of the singleton pattern[18] to insure that only one copy of the server is running in a given VM. The init() method takes a configuration file, and an optional port as arguments. The port is used for binding the Middleman service, and the configuration file is routed to the HsConfigService for further processing. Furthermore, the HaystackRootServer will invoke the init() procedure on the HsNameService as well as the HsConfigService (these are the two essential services we need to bootstrap). Additionally, we can terminate the HaystackRootServer by calling the close() method which will close the HsNameService.

The only other vital function of the HaystackRootServer is the ability to statically call for a number of the bootstrap services. Table A.1 shows the static data member in the HaystackRootServer and its corresponding value.

Static variable	Points at
myNameService	The HsNameService running under this HaystackRootServer. This allows us to lookup a pointer to any service.
myConfigService	The HsConfigService, holds global settings
myDebugger	HsDebug, a tool for debugging Haystack
myErrLogger	HsError, a tool to deal gracefully with errors
myPersistentStore	The HsPersistentObjectService
myObjCreator	The HsObjectCreatorService
myMsgLogger	HsMessage, for non-error messages
myDispatcher	The HsDispatcher, so that new services can register their interest in HDM subgraphs.
myCounter	The HsCounter service generates unique IDs
mySendable	The appropriate Middleman object for this root server.

Table A.1: Static variables within HaystackRootServer

A.2.1 The `HsService` Class

All services extend the `HsService` class. This is necessary in order to guarantee some common interface for all services. Since we are not as flexible about modeling our services as in IDL (the CORBA object definition language) we need to ensure some sort of base behavior that will allow us to dynamically load services. All services are implemented within the `haystack.service` package.

On instantiation, the `HsService` class will register its `ServiceName` (see Section A.2.2)s with the name service. Furthermore, as we have stated above, `HsService` guarantees the existence of the `init()` and `close()` operations for loading and unloading. All services that need to provide some sort of startup and/or shutdown operations can overload these methods.

A.2.2 The `HsNameService`

The first important bootstrap service, beyond the `HaystackRootServer` object is the `HsNameService` which is loaded on initialization of `HaystackRootServer`. The idea is that one service can ask the `HsNameService` for a pointer to another service. The returned pointer can be "cached" by the querying service (so that calls to the `HsNameService` are minimized). Abstractly, the `HsNameService` maintains one table, mapping service name to the instantiated service object.

The `ServiceName` field refers to the name of the service. This name is unique to the Haystack in question, and is persistently maintained with the service. So if there are two services that load information from the Internet (so-called fetch services), they can be named *fetch1* and *fetch2*, and the names will be bound to the particular code. If there is a collision between two names, the second service with the same name will not be loaded and the user of the system will be notified of the failure. It is therefore the programmer's responsibility to ensure that names for services are unique. Naming is elaborated on below. The service pointer field simply contains a pointer to an instantiated (in the Java sense) version of the service object.

Service Naming

The simple naming scheme described above is unfortunately incomplete. It does not address a number of the implicit goals of Haystack. Specifically, it becomes difficult to ensure that new services can be created dynamically, or existing services are modified. This is due to the inflexible nature of naming services. What we would like instead is to be able to "describe" services.

The reasoning for this extended definition is as follows. Within the context of a single user Haystack system, if a certain service serviceFoo is upgraded in such a way that the results from version 1 of the service are not the same as version 2. It would therefore seem that we should maintain both versions of the service. But what do we mean when we talk about serviceFoo? Are we refering to the new or the old? Our concept of flat naming breaks in this case. A quick fix is changing the way in which we assign names. The original serviceFoo is called (from creation on) "serviceFoo v1.0," and the new version is "serviceFoo v2.0."

Unfortunately, as soon as we switch to a multi-user environment we get a different flavor of the same problem. Specifically in dealing with services created by different users but having the same local name, say "serviceFoo v1.0."

To solve the more general problem we introduce the notion that an object's name is actually a set of metadata that define the characteristics of the object. This metadata is in no way as complicated as the Haystack object model, and is constrained to a predefined set of properties. Currently we believe the properties that should be formally part of the services name/description are:

- *name*: This is the general, human readable name of the service (i.e. serviceFoo).
- *version*: The version of the service (i.e. "v1.0"). There is no specified format for this. Each service can maintain it's own numbering scheme as this will reduce the likelihood of collisions between like-named services.
- *creator*: The exact syntax of this is an open issue, but a possible option in the first implementation is the email address of the service implementer. If the

service is part of the core Haystack package, the creator field will state "system" To preserve our ability to add additional packages to Haystack at a later time, "system" and any strings derived from it (i.e. "system_foo") are reserved creator names

In addition to the above properties, there is an additional set of metadata items that are important or useful, but are not important for naming (in its pure definition). These, as well as other fields, can be added to future versions of Haystack:

- *service type*: this is the fuzzy notion of what function the service performs. For example we can have a number of services that serve as interfaces to IR systems. Therefore the type of these objects is "IR system interface." The exact method of classification is an open issue.

- *function signature*: This describes the input/output behavior of the "main" function of the service. With the function signature, we can ask for all services that have a certain function and invoke them. For example, if the services have a cleanup() procedure, a cleanup daemon can collect all services that have this procedure and invoke them.

- *cost*: the monetary cost of using the service, usually 0.

- *trust*: a value assigned by the user as to the trustworthiness of information returned by the service.

- *reliability*: a value assigned by the user to indicate the reliability of the service in responding (i.e. how reliable the network connection is to the service).

It is very likely that we will find additional features that we want to describe services.

What we implicitly achieved by introducing the service metadata concept is the ability to map a description to a service rather than a name. We argue that the three primary fields for service naming partition the set of all services into single service sets. That is, there will most likely be only one service that maps to a given name, version, and creator. Beyond this, `HsNameService` is easily extensible to a much more flexible and powerful system. It will now be possible to ask questions such as: "give me all services that are of type 'IR interface service' and cost 0."

We can also improve our quality of service when answering user's queries to information needs. For example, if a user makes a query for an object that has two `needle.Type` attached to it, one saying "this object is of type Postscript" and the other saying "this object is of type Text." We can check to see which service created these *type* `Needle`s, then check the trust value of these services, and use these values in displaying the results in a fashion that is more useful to the user. In this example, if the service that created the "Postscript" guess is more trustworthy Haystack could pull up a postscript viewer instead of a text viewer.

The functionality we describe above provides us with a mechanism to do two types of name resolution. The first, is the use of a name that is guaranteed to be unique (with high probability), primarily the "[name] [version] [creator]" triplet. The second resolution is a "query" for services. We define a specific kind of object called a `ServiceName`. This class encapsulates all of the metadata characteristics of services as described above. To perform a query we create a new `ServiceName` and fill in the fields we know. For example, if we know that we want a query service (`HsQuery`) but know nothing else about it, we fill in the name field of the `ServiceName` with `HsQuery` and submit it to the `HsNameService` which returns a list (a `Vector`, specifically) of matching services. It is then up to the programmer to decide which service they want to use. It may be possible in the future to implement filters that provide some default decisions for choosing between services. For example, a filter could decide which was the newest version of the service, and return only that one.

A.3 The Basic Services

A.3.1 The `HsObjectCreatorService`

One of the basic functions of Haystack is to dynamically create new HDM objects. This task is handled by the `HsObjectCreatorService` service. The interface for the `HsObjectCreatorService` is fairly straight-forward. One function, `newObject(`n`)` generates a new straw of type n (where n is a string of the fully qualified class name of the `Straw`, see section 4.2). The `HsObjectCreatorService` then uses the Java reflection API to dynamically load the class definition and instantiate the desired object. If the class for a certain straw does does not exist, or for some reason can not be created, a `NoSuchStrawException` is thrown. By using the fully qualified class name `HsObjectCreatorService` allows for the loading of objects not directly in the `haystack.object` package. This is valuable as it may be desirable to keep one Haystack installation on a machine intended for multiple users. However, as users may want to generate new object types, these will necessarily sit outside the `haystack.object` package.

`HsObjectCreatorService` also serves two other important functions. First, it will contact the `HsCounter` to get a unique ID for the new straw. Second, it will contact the `HsPersistentObjectService` to register the newly created object so that persistent storage is guaranteed when the HDM data is flushed to disk.

Future Improvements

Although it is not the direct responsibility of the `HsObjectCreatorService`, for reasons we shall discuss below, adding functions to dynamically generate new class files for new `Straw` types would be an extremely valuable addition to Haystack. This functionality can be achieved in one of two ways. Either the `HsObjectCreatorService` will generate the Java code for the new `Straw` types or directly act as the bytecode generator and create the class file for the new `Straw`.

A.3.2 The `HsDispatcher` Service

A full description of the `HsDispatcher` is available in [6], but a general design overview is provided here. It is our intent that the HDM be expanded dynamically by a variety of tools. Essentially, we would like to implement services that express interest in some HDM sub-graph and are notified when new instances of the sub-graph come into existence, or existing instances change.

We could of course engineer the services to merely listen for any event in the HDM and then decide whether or not to act on it. However, this model is highly inefficient for the programmer who must decide if the object that triggered the event is one of interest. It would be much more preferable to describe interest in a sub-graph. To facilitate this, Haystack includes the `HsDispatcher` service. `HsDispatcher` acts as an intermediary between event driven services and the data model.

There are two orthogonal aspects of the `HsDispatcher`'s job. The first is to distinguish between the events triggered in the HDM. The second is to distinguish between structures in HDM. Events can be enumerated (as we do below), but the general graph ismorphism problem is believed to be NP-hard[2]. Therefore, allowing services to express interest in arbitrary graph structures would prove infeasible. Rather we concentrate on what we call star graphs.

The `StarGraph` Class

In order to express interest in a specific subgraph, a service instantiates a `StarGraph` object with information about the desired structure. Within Haystack, we define a star graph to consist of a central node with a number of attached rays. We can have any number of (uniquely labeled) rays which consist of `Tie` objects and the `Straw` object each `Tie` is connected to.

An example may clarify this model. The `HsTypeGuesser` service has the task of deciding the type of an object. Type can be guessed by two heuristic methods. One method is based on analysis of the file name or Uniform Resource Location (URL)

[2]Graph isomorphism corresponds to the problem of verifying that one graph is contained in another.

and the other method is based on the analysis of the content of the document[3]. The `HsTypeGuesser` is therefore interested in a `bale.HaystackDocument` object connected to a `needle.Location.URL` and a `needle.Body` object. Graphically, this corresponds to Figure A-3.

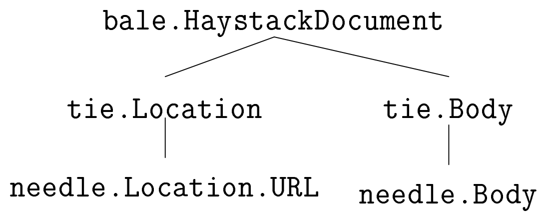

Figure A-3: A Sample star graph.

In this case, we don't actually care what the types of the two `Tie`s are. The graph is still of interest whether the `needle.location.URL` is connected by a `tie.location.URL`, a `tie.Location`, or just a `Tie`. Say we also have a new type of `needle.Location` extension. For example, a `needle.location.RMAILID`, which indicates the location of a specific mail message in an RMAIL file. We extend our `HsFetchService` to retrieve this type of document. Our fetch service is now not only interested in the `needle.location.URL`, but also other kinds of `needle.location` objects. We now want to see the `StarGraph` that includes the wildcard for of the `needle.Location` type, specifically `needle.Location*`[4] character). To allow for this behaviour, the `StarGraph` object is instantiated not with an explicit description of the sub-graph, but rather a regular expression that models all acceptable matches. Figure A-4 represents this graphically. Note also that we are assuming a case insensitive regular expression (i.e. tie* matches `Tie`, `tie`, and `tie.Location`). The complete defintion of the `StarGraph` regular expression syntax is available in [6].

[3]For example, we can guess a file is Postscript if it starts with "%!" We can also guess it is Postscript if the suffix of the file name is ".ps."

[4]When specifying a regular expression $s*$ means we want a string s followed by zero or more charaters

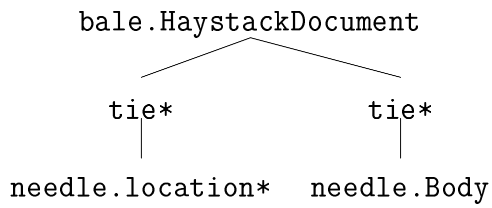

Figure A-4: A Sample star graph with regular expressions.

The HDM Event Model

Now that we have an undestanding of the type of structures we are trying to observe, what type of events are services waiting for? The following are the events generated by the HDM and their meaning:

- A `HaystackEvent` is generated for changes in the structure of the graph. Specifically, there are two extensions to `HaystackEvent`:
  - Whenever a change in graph structure happens (as indicated by the setting of both the "forward" and "back" pointers of a `Tie`) a `HaystackCreateEvent` is generated.
  - When a `Needle` that is connected to at least one other `Straw` has its data changed (by means of a `setData(...)` invocation) a `HaystackChangeEvent` is generated.
- An `ObjectEvent` is generated for changes and creations in any independent `Straw` (regardless of what it is connected to). This is useful for services that are interested in keeping track of all object changes in the HDM (for example the `HsLore` service will want to update each `Straw` entered in the database as it happens). There are three types of `ObjectEvents`:
  - An `ObjectCreateEvent` is generated whenever a new HDM object is created.
  - An `ObjectDeleteEvent` is generated whenever an HDM object is deleted.

- Finally, Haystack defines an abstract (i.e. non-instantiatable) `ObjectChangeEvent` class. This class is extended by two specific implementations:
  * An `ObjectChangeLinkEvent` is generated when any pointer (i.e. the pointers in `Ties` as well as the `StrawTies` in `Straw`) are changed.
  * An `ObjectChangeDataEvent` is generated when a `Needle`'s data is changed.

What we have indirectly achieved is the ability to differentiate between different granularity of events, from structure to individial "atomic" components.

A.3.3 The `HsConfigService`

One of the most important design concerns for Haystack is the customizability of the system for the preferences of a given user. It is important to provide a single point of entry for configuration of a wide variety of services. To this end, Haystack includes a configuration service, `HsConfigService`. A baseline implementation of the `HsConfigService` is disucussed below, as well as possible improvements for added flexibility and scaleability.

Currently, Haystack defines the following format for configuration files:

$$\boxed{\mathtt{Key}_i\ \mathtt{Val}_1\ \mathtt{Val}_2\ \mathtt{Val}_3\ \ldots\ \mathtt{Val}_n}$$

This format is *very* simplistic, and does not allow for repetition of keys. Each $key \rightarrow value(s)$ mapping must reside on a seperate line. Values are assumed to be seperated by single spaces. Examples of keys are: *HAYLOFT* (the location of the user's Haystack repository), *ObjectStore* (the file used by the persistent object store), and *CacheDirectory* (the directory in which to maintain cached information). Lines that start with the '#' character are comments and are ignored.

`HsConfigService` currently provides only two interesting interfaces: `getVal(n)` and `getVal(n,i)`. The first returns the `Vector` of values corresponding to the key n. The second returns the string corresponding to the i^{th} value for key n. Both return exceptions if the key is not found (see Appendix D).

The only other functional part of `HsConfigService` is the initialization routine which ensures that the user's Hayloft directory exists. If not, `HsConfigService` will create it, and check for read/write permissions.

Future Improvements

As the `HsConfigService` is currently implemented in the most simplistic functional model possible, some future improvements are discussed below. The first major improvement would be in the format of the configuration file. This file should allow for added robustness (in terms of allowed keys and values). Ideally, services in Haystack should be able to construct a set of rules for configuration items they will require. This rule could then be passed to the `HsConfigService` and it, in turn, would confirm that the key/value mappings it has loaded from disk adhere to the restrictions of the rules.

An example rule would be (as proposed by the `HsPersistentObjectService` to the `HsConfigService`): "Make sure the ObjectStore key exists, and has one textual value." Another example may be: "make sure that key `foo` can be read as an integer, and that it has one of the following values: 1,5,7." Once Haystack has a wide variety of services, the exact scope of the rules will become evident. It should then be possible to formulate a rule language for services.

Finally, a user interface that allows for changes and explains the properties of different keys would also be invaluable.

A.3.4 The `HsResourceControlService`

Although it is possible to implement synchronized methods within the context of a single Java VM by means of the synchronized key word, it is impossible to guarantee locks outside that scope. To allow for locking, Haystack includes a resource control service, `HsResourceControlService`. This service allows for locking and unlocking of arbitrarily named resources. Locks are held for a preset amount of time (currently 5 minutes).

The methods currently implemented by `HsResourceControlService` are:

- `testResource(`n`)` which returns true if the resource n is locked, and false otherwise.
- `unlockResource(`n`)` forces an unlock of the resource n. `HsResourceControlService` doesn't verify that the caller has the "right" to do this.
- `lockResource(`n`)` tries to lock the resource n. If the lock was successful it returns true, if the attempt to lock times out, false is returned.
- `nonBlockLock(`n`)` tries to lock the resource, and immediately returns (i.e. there is no timeout).

The `HsResourceControlService` internally maintains an extension to the `Hashtable` class called `ResourceControlHash`. The `ResourceControlHash` maintains a list of current resources as well as a list of their timeouts. The `ResourceControlHash` manages the synchronization necessary to ensure atomic lock functionality.

Future Improvements

As with most of the Haystack services, a number of improvements can be made to increase the functionality of the `HsResourceControlService`. Specifically, it would be value to provide some sort of deadlock detection or two-phase locking. The `HsResourceControlService` could easily maintain some sort of ordering on the resources.

This type of locking can be achieved by holding some function r mapping resource names uniquely to some sortable numerical value: $r(name) = i_{name}$, where *name* is the resource name, and i_{name} is the unique numerical value for that *name*. Instead of merely invoking the above functions with the a resource name, services would also send a pointer to themselves. The `HsResourceControlService` could then maintain a list of resources held by each service, and only allow a new lock to be allocated if *r(newName)>r(mostRecentLock)*. Releases are done in reverse. The function r would, of course, need to be predefined.

A.3.5 The HsPersistentObjectService

Once we have created HDM objects it is vital that we be able to save them persistently to disk and reload them between Haystack sessions. To this end, Haystack provides a HsPersistentObjectService class.

HsPersistentObjectService is built using a very simple model. Within the service a Hashtable is maintained mapping HaystackIDs to Straws. When a new object is created, the method register(...) is invoked by the HsObjectCreatorService. The new Straw is placed in the Hashtable. To save the Straw to disk HsPersistentObjectService makes use of a Java implementation of a standard DBM package developed by the WWW Consortium for a Java based web server [44]. In Haystack we extend the DBM functionality to a handle the serialization of Java objects. HsPersistentObjectService will pass the (key,value) pair of (HaystackID, Straw) to the DBM package. The DBM will serialize the Straw (i.e. transform it into bytes), and save it to disk keyed on the HaystackID. Recall from the discussion of the StrawTies implementation that the Straws "break" apart from the HDM structure on serialization. This ensures that objects are saved and loaded independently (rather than in complex subgraphs).

If a service requests an object from the HsPersistentObjectService by means of the getObject(...) method, the HsPersistentObjectService will first check in its cache for the object. If the Straw exists in the hashtable it is immediately returned. Otherwise, an attempt is made to retrieve the object from the DBM database. If that fails, an exception is thrown (ObjectNotFoundException). If the object is loaded from disk and reconstructed, it is placed in the cache and returned to the requesting service.

Because Straws are "disassembled" for saving, it is necessary to reconstruct all the pointers once the object is loaded from disk. To do this, Straws contain a boolean variable called loaded. If loaded is set to false, the Straw will repeatedly call the HsPersistentObjectService to obtain all the pointers to its outgoing edges. Recall, this is achievable beacuse Straws always retain the HaystackIDs for outgoing edges.

Future Improvements

One of the fundamental problems of the `HsPersistentObjectService` is its inability to clean its cache. This leads to the constant growth of the memory requirements of any running Haystack session. As it is impossible to count the number of services holding a pointer to any particular `Straw` we can never be certain if it is safe to remove the `Straw` from the cache. If we removed the `Straw` and someone modifies it at some later point, the change will not be saved. The simple solution is to rely on the Java garbage collection mechanism which will eventually remove a `Straw` that nobody is holding from memory. Unfortunately, as long as the cache holds the `Straw`, it will never be garbage collected.

Fortunately, Java 1.2 provides a new type of object reference (pointer). These references are known as "weak references." Weak reference objects maintain a pointer to an object. If nothing else has a strong reference (i.e. is holding a pointer) to the object, the garbage collector can remove the object from memory. If the `HsPersistentObjectService` cache maintained only weak references to the `Straws` the garbage collector could still remove the object from memory and allow the Haystack memory footprint to remain manageable.

A.3.6 The `HsCache` Service

As discussed in Chapter 4, the HDM at times makes reference to external sources of information. This allows the HDM to remain compact in size, but still robust enough to handle the variety of objects that a user may want to archive.

Recall that the `Promise` classes provide the mechanism for "re-generating" `Needle` data dynamically. This is inefficient, however, if the the same `Promise` is repeatedly asked to generate the same data. As a solution, Haystack provides a mechanism for caching data created by a `Promise`. Optionally, a programmer may modify the `getData()` operation for a `Needle` so that as data is being generated by the `Promise` it is simulatenously being sent to disk for temporary caching. The `Needle` requests this disk space by calling the `allocateCache(key)` command on the `HsCache`, where key is the Needle's ID.

When HsCache receives a request it returns a special extension of the Java stream Writer class, CacheFileWriter. This extension overrides a number of the Writer's functions so that the HsCache can check to see if the cache space is in use.

After the Needle has placed the data in the cache, it can request that the HsCache open a Reader to that cache space (a Promise holding Needle returns a Reader when getData() is invoked). This reader is also an extension to the standarad Reader class and is called CacheFileReader. As data is read using the CacheFileReader the "life" of the item in the cache is extended. As soon as data is not being actively read, the file will be removed after a period of five to ten minutes.

A.3.7 Other Base Services

The HsDebug, HsMessage, and HsError Services

As the HaystackRootServer allows for a multitude of services to run concurrently, it was necessary for us to create a mechanism by which we could log errors and messages in an intelligent fashion[5]. This would prevent the user from being barraged with a variety of status and error messages that might be a normal part of the system's function. The three services, HsDebug, HsMessage, and HsError provide this functionality through a very clean interface.

All three of these services extend the LoggerService (in the haystack.service.misc package). This service provides the basic functionality which allows messages to be printed to any arbitrary location (i.e. file, screen, etc.). Every time the print() command is invoked on any of the three services above, the LoggerService pretty prints the message to the appropriate location. Both HsMessage and HsError are trivial extensions to this. HsDebug, on the other hand, provides a slightly more complex set of functions.

The customary way to debug an application has usually consisted of setting some

[5]Intelligent both in terms of ease of the programmer to add debugging information, and the use in determining where problems may exist

boolean value (call it debug). When you want to debug your application you set the value to true. Inside your code, the programmer ordinarily places conditionals: if (debug) then print out some stuff to STDERR or STDOUT. `HsDebug` provides a much more refined logging mechanism. A user can set debugging on for anything as fine grained as a class. Within a class, the class developer can call the function `print()` on the `HsDebug` service. The arguments to `print()` are the reference to the object being debugged (i.e. the `this` in Java), and the message to print. Someone who is debugging the system can tell `HsDebug`, by means of the `debugClass()` method which classes they are interested in debugging (to turn this off you would use the method `unDebugClass()` with the class name as an argument).

Now, whenever `print()` is invoked, the `HsDebug` service checks to see if debugging is turned on for the invoking class. If it is, the debugging message is printed. `HsDebug` also allows the programmer to set where the debug messages are printed to. The method `toScreen()` resets messages to `System.err` (i.e. STDERR).

A.4 The Interface Services

A.4.1 The `Middleman`

This section provides a very basic introduction to communication between two `HaystackRootServer`s. The full implementation details for the `Middleman` are available in [6].

When initially selecting to work in Java it was necessary to decide how inter-process communication was to work in Haystack. Specifically, because Java processes run within the context of a Virtual Machine (VM), some mechanism is needed for communicating between processes running in seperate VMs. For example, Haystack specifies that a user always have one central `HaystackRootServer`. Perhaps this `HaystackRootServer` is running on the user's computer at home. If the user needs to query his Haystack from work, it is necessary to be able to interact with the main `HaystackRootServer` remotely. Additionally, as Haystack begins to support collaboration inter-VM communication will become vital.

The `Middleman` technology facilitates this inter-VM communication by implementing a form of remote procedure call (RPC). Java provides a built in method called Remote Method Invocation (RMI) for inter-VM communication. However, this implementation was evaluated to be two slow and complex for our purposes. Additionally, RMI is only provided in some implemenations of Java and may not be used in the future. Other solutions, such as CORBA and ILU [24] were too complex for our needs.

In implementing Haystack we wanted to provide a very simplistic mechanism for exporting the functions of one service so that a process running in a seperate VM would be able to invoke these functions as if they were in the local VM.

The general process for this communication model is fairly simple. When implementing a service that is to have exportable functions, three versions of the service are created. The first is the interface of the service. For example if we want to have an exportable service `HsFoo`, the interface will exist `haystack.service.HsFoo`. The implementation that does all the work will exist in `haystack.service.real.HsFoo`. The "stub" (or skeleton) implementation exists in `haystack.service.virtual.HsFoo`.

When the "root" `HaystackRootServer` is created it will load a copy of the `real.HsFoo` service and a `MiddlemanServer`. Figure A-5 describes a simplified interaction between a service A running in a non-root VM with HsFoo.

- In step A of the interaction, the `real.HsFoo` service registers itself with the `HsNameService`. At the same time, the `HsNameService` running in the context of the second `HaystackRootServer` connects to the "root" `HsNameService` by means of a `Middleman` service. `Middleman` interaction involve the setting up of TCP socket over which serialized Java objects are passed (i.e we convert Java objects into a byte stream, send it over the TCP connection, and reassemble the bytes into the Java object on the other end).

- In step B, Service A asks it's local `HsNameService` for a pointer to `HsFoo`.

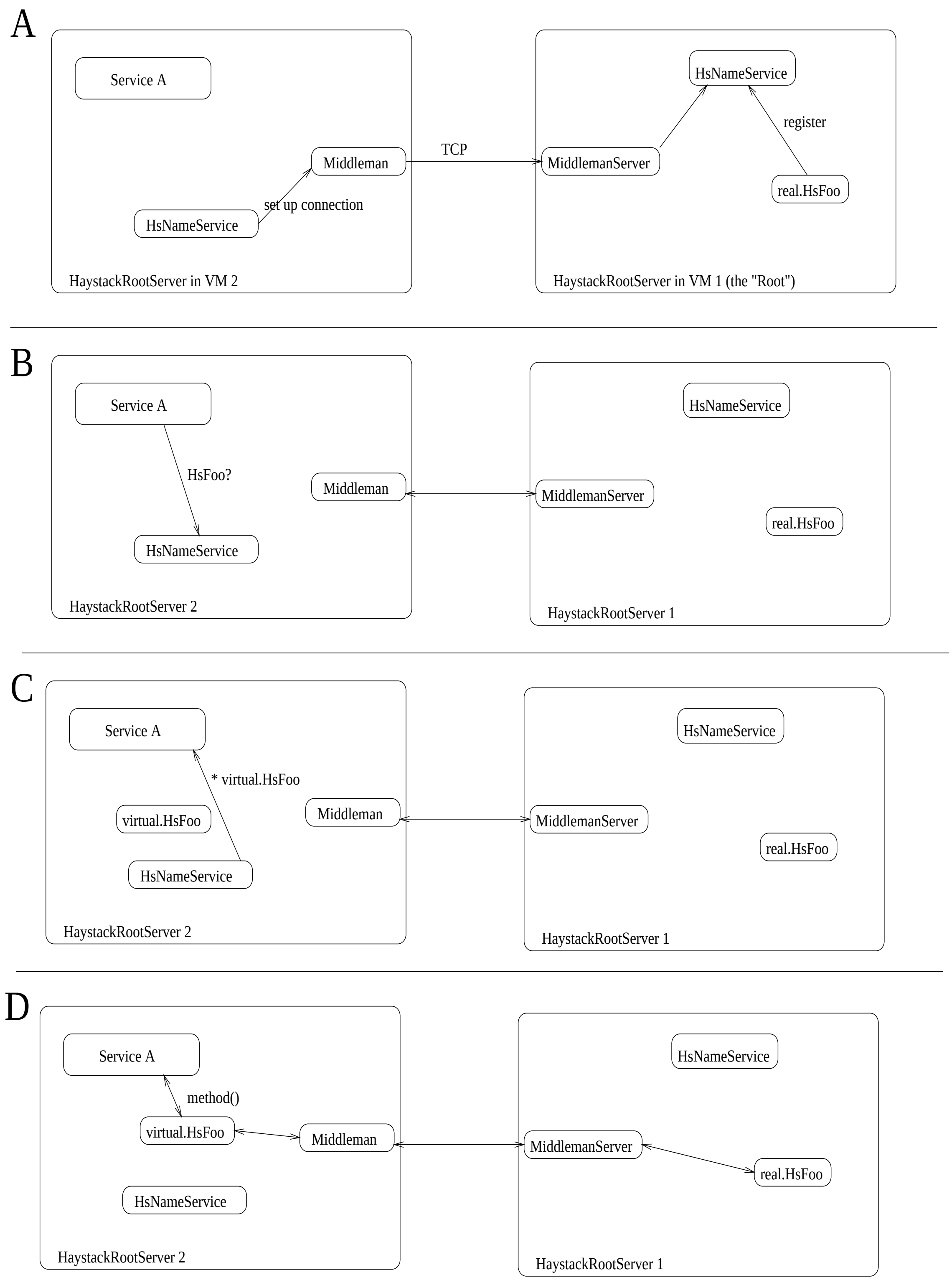

Figure A-5: Abstract inter-service communication model

`HsNameService`, after deciding that `real.HsFoo` object is running in the root `HaystackRootServer`, will instantiate a `virtual.HsFoo` object.

- The `HsNameService` will pass the pointer to the virtual.HsFoo object back to Service A in step C.

- In step B, Service A can now call the functions of `virtual.HsFoo` just as it would when communicating with the `real.HsFoo` object. The `virtual.HsFoo` object will then pass the method name and arguments it receives to the `Middleman` which in turn will pass the arguments to the `MiddlemanServer`, and these get applied to the `real.HsFoo` objet. The results are passed in the reverse direction.

The actual model for the `Middleman` is much more complex and allows for multiple connections between VMs and a more complex method for passing "packets" between the VMs.

A.4.2 The `HsCommandAPI` Service

Because we would like to see Haystack extended and used by non-Java systems the `Middleman` is not a perfect solution. In addtion to the `Middleman` we introduce the `HsCommandAPI` service. This service allows users to create a set of "scripts" (in Java) that are invokable from external processes by means of a textual command. The command scripts (which are of type `Command`) reside in the `haystack.object.command` package.

At any point after instantiation, it is possible to call the `register(...)` method on the `HsCommandAPI` with a `Command` object as an argument. Once an object is registered it is possible to invoke it through the `HsCommandAPI`. Abstractly, what we now have is a service to allow for virtual method invocations. This invocation is done by means of the `invoke(...)` method which accepts a string as an argument. The string is usually in the form (optionally with a new line at the end):

$$\boxed{\texttt{Function}_i\ \texttt{Arg}_1\ \texttt{Arg}_2\ \texttt{Arg}_3\ \ldots\ \texttt{Arg}_n}$$

Arguments are seperated by a white space character. For multi-word arguments, `HsCommandAPI` allows for the use of either quote or double-quote characters to specify the beginning and end of a multi-word string (i.e. `'this is an example of a multi-word argument'`). Additionally, `HsCommandAPI` handles a number of special characters:

Character	Evaluates to:
\t	tab
\n	new line
\r	return
\f	form feed
\x	x (i.e. any other character)

Table A.2: The interpretation of special characters within the `HsCommandAPI`

This leads the `HsCommandAPI` to invoke the `Command` object corresponding to the name Function_i with the set of parsed arguments. `HsCommandAPI` understands the special "`help [functionName]`" command which returns the set of functions it is capable of handling or the usage information for a particular function (with the optional functionName argument).

The `invoke()` command will return the string response generated by the invocation of the specific function. If there was an error in handling the function an exception (`BadCommandException`) is thrown with details of the error.

The `Command` Class

The `Command` interface is fairly straightforward. It contains a limited set of methods that allow for the basic abstracted functionality necessary to fulfill teir task. Currently, classes that implement `Command` provide:

- A `name()` method which returns the string name by which the specific command shoule be invoked.

- A `commandSig()` method which returns the "signature" of the command (i.e. how it should be used.

- A `help()` method which returns some textual information on the intent and usage of the command.
- And a `fulfill()` method which takes the arguments as parsed by the `HsCommandAPI` and either processes them or returns an error.

Haystack currently supports the following commands:

- `AttachCommand` (usage: `attach [forward | back] HaystackID1 HaystackID2`, returns: boolean) Attach allows us to connect two objects (with IDs *HaystackID1* and *HaystackID2* respectively). The forward/back relationship indicates the direction we want the link to take (i.e. $object1 \rightarrow object2$ or $object1 \leftarrow objec2$).
- `CreateCommand` (usage: `create type_of_new_object`, returns: HaystackID) This command allows us to create new objects of type *type_of_new_object*. On success it returns the ID for the new object. (Example: `create haystack.object.bale.HaystackDocument`).
- `SetCommand` (usage: `set HaystackID value_to_set`, returns: boolean), this command allows you to set the values of any `Needle` (indicated by `HaystackID`) to some value (specifically, *value_to_set*). It returns 1 on success, 0 on failure.
- `TerminateCommand` (usage: `terminate`, returns: boolean) This command causes the `HaystackRootServer` to shut down. This is a "clean" shutdown which calls the `close()` function on all services gets called.
- `ViewCommand` (usage: `view HaystackID1`, returns: string) View returns the string representation of the `Straw` with the given `HaystackID`.

Future Improvements

There are a number of obvious improvements to be made to the `HsCommandAPI` service. Specifically, it would be useful if commands could generate a grammar that could be compared by the `HsCommandAPI` to the incoming commands. That is, a `Command` should be able to specify it's function signature in such a way that the HsCommandAPI could perform the necessary type checking and argument checking for the Command.

A.4.3 The HsCommunicator Service

Now that we have a method of processing different types of external commands within Haystack we should briefly discuss the two services within Haystack that currently use the HsCommandAPI. We can view these two services as handling the "transport" layer between outside functions and the HsCommandAPI.

The first, the HsCommunicator service, acts as a network socket based interface. The service starts up within its own thread space and listens to incoming connections on a given port (5776 by default). These connections are passed off to CommunicatorThread objects (in the haystack.service.misc package). The CommunicatorThread objects are themselves threads which allow the HsCommunicator to handle multiple calls simultaenously. The CommunicatorThread objects simply read the input from the socket and submit the text to the HsCommandAPI.

There are a number of utility functions within the HsCommunicator that allow for the cleanup of all CommunicatorThread objects. We would like Haystack to shutdown when the close() operation is invoked on the HsCommunicator it will run through the CommunicatorThreads it knows about and shuts them down. To do this, the HsCommunicator maintains a list of all running CommunicatorThread objects.

Future Improvements

The one significant improvement that should be included in future releases is the addition of a security protocol for the HsCommunicator. There is currently no way of knowing who the remote caller is. This may lead to abuse and destruction of a person's Haystack by a malicious user. When an appropriate security protocol is designed it should be fairly trivial to incorporate into the HsCommunicator.

A.4.4 The `HsCommandLine` Service

As not all commands need to be executed over a socket connection, Haystack provides the `HsCommandLine` service. This service simply sits and listens to the `System.in` `InputStream` (same as STDIN). Whenever the user types a command and hits enter, the string is sent off to the `HsCommandAPI`.

The only two important functions of this service are: `run()` which starts the service listenting, and `setPrompt()` which the user to change the prompt that is visible on the command line (default: "`Haystack> `").

A.5 Event Driven Services

A number of event driven services are described in the next sections However, a brief summary of their design is provided here. Services interested in listening for events (i.e. changes in the HDM) extend the class `haystack.service.events.HaystackEventService`. This class provides the basic functionality necessary to register the service with the `HsDispatcher` to be notified of specific events (`HaystackEvent`s and/or `ObjectEvent`s).

Event driven services are different than other Haystack services in two ways. Services that extend the `HaystackEventService` class will also implement the interface `HaystackEventListener` and/or `ObjectEventListener`. The functions defined by each of these "handle" the events generated by the HDM. We also impose the invariant that any service modifying a piece of data in HDM must lock the HDM object first in the `HsResourceControlService`. This prevents a number of race conditions, and allows us to decide when an object has reached stable state (when it hasn't been locked for a long time).

A.6 Data manipulation Services

In [6] Asdoorian defines the implementation of the services necesary to archive, index, and query documents in Haystack. A full description of the implementation is beyond the scope of this thesis, but a description of the general design of these important compenents is provided below.

A.6.1 The `HsArchive` Service

In order to intitially "introduce" documents into a user's Haystack a basic service is necessary to create a kernel of an HDM subgraph. From this kernel Haystack services can begin the process of assembling a representation of the document in the HDM. `HsArchive` serves in this capacity by instantiating the "kernel" composed of a `bale.HaystackDocument` and a `needle.Location`.

Once the `bale.HaystackDocument` and `needle.Location` have been created, the event based service, `haystack.service.fetch.FetchService` (or a more specialized fetch service[6]) is informed of a new event in the HDM. The service will subsequently try to retrieve the data being pointed at by the `needle.Location`. When this data is obtained a `needle.Body` is attached to the `bale.HaystackDocument`. Additionally, the `FetchService` will also calculate the MD5 checksum and generate a checksum needle to attach to the `bale.HaystackDocument`.

`HsArchive` is invoked with a variety of options that indicate how the HDM cluster for the archived document should be constructed. A document's uniquness is measured by two values: its location and its checksum. `HsArchive` maintains two tables, one mapping locations to `needle.Location` objects with that location, and a table mapping checksums to the `needle.Body` objects that produced the checksum value. In invoking the `archive(...)` method in `HsArchive`, a user may specify Haystack's behavior in dealing with the four possible cases: location and checksum match, one of the two matches (2 cases), and neither matches. The user's specification for archiving behavior for a specific document are retained in a table.

After the `FetchService` obtains the object and makes the appropriate connections, another event-driven service is invoked. This service, the `HsFileChecker`, awaits the appearance of a `bale.HaystackDocument` attached to a location and

[6]As we have different `needle.Location` types it may be useful to provide different fetch services to obtain the different pieces of data.

checksum. The `HsFileChecker` will lookup the user specified archive options described above. It will also check to see if the location and checksum are already in the `HsArchive`'s databases. If both the URL and checksum match, the implication is that the user asked for the same document to be archived twice. The default behavior is to remove the new HDM cluster and abort. However, the user may specify that a new (duplicate) cluster be created. The last case (neither match) is simiarily trivial, as we know the object is entirely new, the cluster is left alone and archiving continues.

If the location matched but the checksum did not, the implication is that the document has changed. The default behavior is to connect the new `needle.Body` object to the old `bale.HaystackDocument` and in effect supercede the previous body. Alternatively, a user may specify that the HDM cluster continue to exist seperately. Finally, if the checksum matched but the location did not, one of two things may have happened, the document was copied or the document was moved. This is easily verifiable by looking at the old location and determining if the document is still there. The options available to the user in this state would be to merge, supercede, or abort the archive.

If the HDM cluster has survived the `HsFileChecker`, other services may begin to act on it. The `HsTypeGuesser` can decide type type of the object and attach the appropriate `needle.Filetype` object. Once a type is guessed, field finders and textifiers can start their work on the cluster. A field finder can parse through the `needle.Body` of a file, finding relevant pieces of data corresponding to the document state previously discussed. Trivially, a `HsMailFieldFinder` service can extract the *to*, *from*, *subject*, *date*, and *id* lines for an e-mail document and generate the appropriate `needle.Mailfield` objects. Finally, a textifier service specific to the type of document may extract the text form the document. This extraction does not need to be done immediately and the textifier may leave a `Promise` in the `needle.Text` object. At the time of this writing services have been implemented to extract text from Postscript and PDF files as well as some trivial ascii based file formats.

A.6.2 The `HsIndex` Service

The `HsIndex` service is an event driven service that is set to wait for a user configurable star graph. The most frequently used star graph for `HsIndex` is a `bale.HaystackDocument` attached to a `needle.Text`. When the `HsIndex` service is notified of the existence of such a sub-graph it will ad a pointer to the subgraph in a queue. Periodically it will test to see if the cluster is in stable state[7]. Once the `bale.HaystackDocument` has arrived in stable state, the `HsIndex` service invokes the `getIndexable()` method. The result of this call is some text.

`HsIndex` will check to see which IR systems the Haystack is configured to use, and will pass the text to that IR system for indexing. Currently the only defined IR system interface in Haystack is to the Isearch system [16], which is defined in `haystack.service.ir.HsIsearch`. `HsIsearch` extends the class `haystack.service.ir.IRSystemService` which has a large set of utility functions for taking the text generated by `HsIndex` and generating something that IR systems can index.

A.6.3 The `HsQuery` Service

Now that we have a number of indexed documents we would like to allow users to documents within the index. `HsQuery` is a simple service that provides this functionality. `HsQuery` accepts a string query and routes that query to the IR system currently being used. The match objects (generally of the type `bale.HaystackDocument`) are encapsulated in a `QueryResults` object and passed back.

[7] Recall from above that stable state is implied when the object is not locked

Appendix B

UML model of `Straw`

Figure B-1 is the basic block in a Unified Modeling Language (UML) model. Each block represents one object. Figure B-2 is a UML class diagram for the primary interfaces and implementations of objects in the HDM.

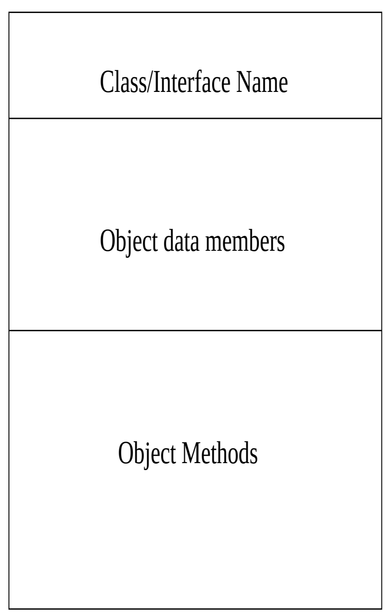

Figure B-1: A simple UML building block.

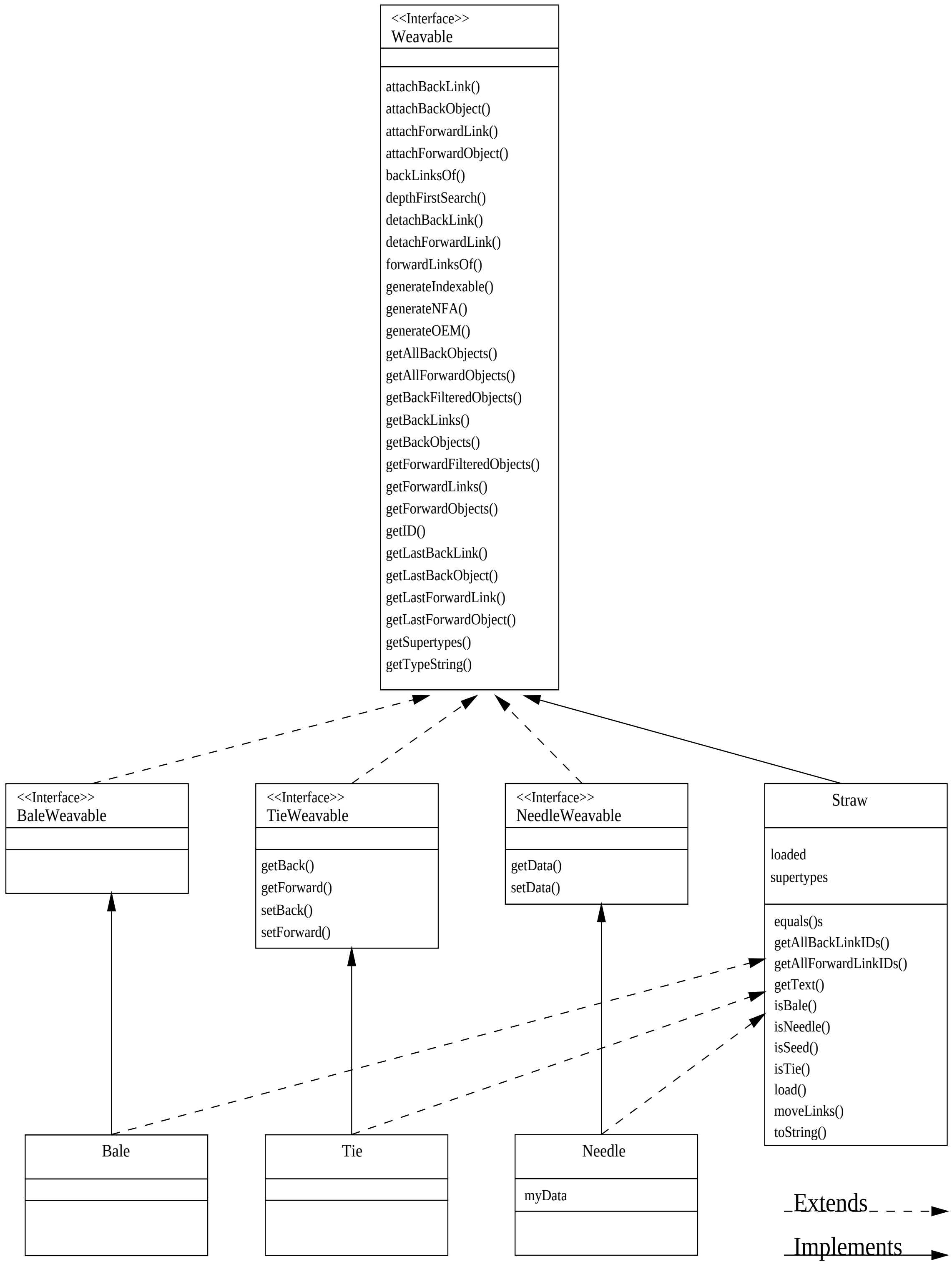

Figure B-2: A UML diagram of the core Haystack object interfaces/classes.

Appendix C

Functions of `StrawType`

These are some of the useful functions provided (statically) within the `StrawType` object.

Function	Purpose
`string2Vector(...)`	Breaks apart a string representation of a class name (with an optional delimiter, "." by default.).
`class2Vector(...)`	Breaks apart the name of the `Class` object into a `Vector`.
`vector2String(...)`	Merges a `Vector` into a String representation. For example: {a b C} becomes `a.b.C`. (the delimiter can be specified).
`vector2Class(...)`	Generates a `Class` object for the given `Vector` of `Strings`. So {a b C} will generate a class object for `a.b.C`. This will generate an exception if there if Java can't find the class file for the object.
`straw2Package(...)`	Generates the name of the package that holds extensions to the Straw. For example, if the argument is `a.b.C`, the function returns `a.b.c`.
`getPackageOf(...)`	Returns the package that this object is part of. So if the `Straw` is {a.b.C}, the function returns `a.b`.

`package2Straw(...)`	Returns the super-type of all objects in the package. For example, given {a.b.c}, the function returns `a.b.C`.
`package2Class(...)`	Same as above, but returns the `Class` object rather than a `String`. Throws an exception if Java can't find the class for the specific `Straw`.
`toTie(...)`	Returns the equivalent "tie" name for a `Class` or the `String` name of the object. For example, if the object is `bale.C`, the function returns: `tie.C`.
`toTieClass(...)`	Same as above but returns a `Class` representations. Exceptions as above.
`toValidTie(...)`	Same as above but will walk back through a `Straw`'s hierarchy until it finds a valid (i.e. Java recognizable) `Tie` extension. For example, submitting `needle.filetype.Postscript`, will return `tie.Filetype` (because `tie.filetype.Postscript` doesn't exist). The function returns `Tie` if it can't do any better.
`toBale(...)`	Same as toTie above but generates a bale.
`toBaleClass(...)`	Same as toTieClass above but generates a bale `Class`.
`toNeedle(...)`	Same as toTie above but generates a needle.
`toNeedleClass(...)`	Same as toTieClass above but generates a needle `Class`.
`getSupertypes(...)`	Finds all *valid* supertypes of a given Straw.
`isValid(...)`	verify that Java can load an object with the given name.

Appendix D

Exceptions

There are a number of exceptions issued by the Haystack implementation. They are provided here in table form as reference.

Exception	Extends	Description
`BadQueryException`	`Exception`	An exception for a malformed query expression.
`CannotFindOpenPortException`	`IOException`	When any service attempting to grab a port for communication can't find one.
`CommunicationException`	`IOException`	When the `Middleman` can't find an open port.
`DotfileException`	`Exception`	Exception when trying to archive a dotfile.
`DuplicateNameException`	`Exception`	`HsNameService` and `HsCommandAPI` exception when someone tries to register the same *named* object.
`DuplicateURLException`	`Exception`	Exception by the archive process when the user tries to archive the same object

`HsServiceCloseException`	`Exception`	If any service can't close properly it describes the problem through the HsServiceCloseException
`HsServiceCriticalInitException`	`HsServiceInitException`	A critical service initialization error. The `HaystackRootServer` will shut down.
`HsServiceInitException`	`Exception`	A service initialization error.
`KeyNotFoundException`	`Exception`	Used when a *database* type service can't find a certain key.
`NameNotFoundException`	`Exception`	Used by `HsNameService` on a `getService()` request if it can't find the service.
`NeedleDataException`	`RuntimeException`	If someone tries to place the wrong type of data into a `Needle` and casting can't fix it.
`NoSuchStrawException`	`Exception`	If the `HsObjectCreatorService` can't create the requested `Straw` because it doesn't exist.
`OEMGenerateException`	`Exception`	If the `HsOEMGenerator` gets errors while generating the OEM file.
`ObjectNotFoundException`	`Exception`	If `HsPersistentObjectService` doesn't have a specific `Straw` .
`SoftlinkException`	`Exception`	If the user tries to archive a softlinked file.

Table D.1: Exceptions generated by Haystack

Bibliography

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janel L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[2] Eytan Adar. Haystack: A personal information repository. Bachelor's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1997.

[3] Eytan Adar and Jeremy Hylton. On-the-fly hyperlink creation for page images. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, number 2 in DL, Austin, TX, June 1995.

[4] Maristella Agosti, Fabio Crestani, and Massimo Melucci. Design and implemenation of a tool for the automatic construction of hypertexts for information retrieval. Revised conference paper, Dipartimento di Elettroica ed Informatica, Università di Padova - Italy, 1995.

[5] Yigal Arens, Chun-Nan Hsu, and Craig A. Knoblock. Query processing in the sims information mediator. Technical report, Information Sciences Institute and Department of Computer Science, University of Southern California, Marina del Rey, California, 1996.

[6] Mark Asdoorian. Data manipulation services in the Haystack IR system. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1998.

[7] Michael K. Buckland. What is a "document"? *Journal of The American Society for Information Science*, 48(9):804–809, 1997.

[8] *Bulletin of the Technical Committee on Data Engineering*, 19(1), March 1996. Special Issue on Integrating Text Retrieval and Databases.

[9] Vannevar Bush. As we may think. *Atlantic Monthly*, 176(1):641–649, 1945. reprinted in [35].

[10] W. Bruce Croft and Lisa A. Smith. A loosely-coupled intergration of a text retrieval system and an object-oriented database system. In *Research and development in information retrieval: Proceedings of the fifteenth annual international conference*, number 15 in SIGIR, pages 223–232, Copenhagen, Denmark, June 1992.

[11] Douglass R. Cutting, David R. Karger, Jan O. Pedersen, and John W. Tukey. Scatter/Gather: A cluster-based approach to browsing large document collections. In *Research and development in information retrieval: Proceedings of the fifteenth annual international conference*, number 15 in SIGIR, Copenhagen, Denmark, June 1992.

[12] C[hris] J. Date. *An Introduction to Database Systems*. The Systems Programming Series. Addison-Wesley, Reading, Massachusetts, sixth edition, 1995.

[13] Samuel DeFazio, Amjad Daoud, Lisa Ann Smith, Jagannathan Srinivasan, Bruce Croft, and Jamie Callan. Integrating IR and RDBMS using cooperative indexing. In *Research and Development in Information Retrieval: Proceedings of the Eighteenth Annual International Conference*, number 18 in SIGIR, pages 84–92, Seattle, WA, July 1995.

[14] D[ouglas] C. Engelbart. Augmenting human intellect: A conceptual framework. Technical report, Stanford Research Institute, SRI, Menlo Park, CA, October 1962.

[15] R. G. G. Catell et. al. *The Object Database Standard: ODMG 2.0*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[16] Center for Networked Information Discovery and Retrieval. Cnidr isearch. http://www.cnidr.org/ir/isearch.html.

[17] Norbert Fuhr. Models for intergrated information retrieval and database systems. In Bulletin [8], pages 3–13. Special Issue on Integrating Text Retrieval and Databases.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[19] Roy Golman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the Twenty-Third Internation Conference on Very Large Data Bases*, number 21 in VLDB, pages 436–445, Athens, Greece, August 1997.

[20] Irene Greif. *Computer-Supported Cooperative Work: A Book of Readings*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[21] David A. Grossman, Ophir Frieder, David O. Holmes, and David C. Roberts. Integrating structured data and text: A relational approach. *Journal of The American Society for Information Science*, 48(2):122–132, 1997.

[22] Junzhong Gu, Ulrich Thiel, and Jian Zhao. Efficient retrieval of complex objects: Query processing in a hybrid DB and IR system. In G. Knorz, J. Krause, and C. Womser-Hacker, editors, *Proceedings of the 1st German National Conference on Information Retrieval*, number 1 in IR, pages 67–81, 1993.

[23] Haystack. Haystack homepage. http://www.ai.mit.edu/projects/haystack.

[24] Bill Janseen, Mike Spreitzer, Dan Larner, and Chris Jacobi. ILU 2.0alpha12 reference manual. Technical report, Xerox PARC, Xerox PARC, Palo Alto, CA, November 1997.

[25] Javasoft. JavaBeans. http://java.sun.com/beans/.

[26] Daniel Knaus and Peter Schäuble. The system architecture and the transaction concept of the spider information retrieval system. In Bulletin [8], pages 43–52. Special Issue on Integrating Text Retrieval and Databases.

[27] Joshua David Kramer. Agent based personalized information retrieval. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1997.

[28] Michael Lesk. The seven ages of information retrieval. In *Conference for the 50th Anniversary of* As We May Think, Cambridge, MA, October 1995.

[29] J.C.R. Licklider. Man-computer symbiosis. *IRE Transactions on Human Factors in Electronics*, HFE-1(7):4–11, March 1960. reprinted in [40].

[30] Clifford A. Lynch and Michael Stonebraker. Extended user-defined indexing with application to textual databases. In Francois Bancilhon and David J. DeWitt, editors, *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, number 14 in VLDB, pages 306–317, Los Angeles, August 1988.

[31] Object Management Group. CORBA/IIOP index. http://www.omg.org/corba/c2indx.htm.

[32] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[33] Douglas L. Medin and Brian H. Ross. *Cognitive Psychology*, chapters 7 and 8, pages 170–259. Harcourt Brace Jovanovich, Inc., Orlando, FL, 1990.

[34] Amihai Motro. VAGUE: A user interface to relational datbases that permits vague queries. *ACM Transactions on Office Information Systems*, 6(3):187–214, 1988.

[35] James M. Nyce and Paul Kahn. *From Memex to Hypertext: Vannevar Bush and the Mind's Machine.* Academic Press, Inc., San Diego, CA, 1991.

[36] Steve Putz. Using a relational database for an inverted text index. Technical Report SSL-91-20, Xerox Palo Alto Research Center, Xerox PARC, Palo Alto, California, 1991.

[37] Jeroen G. W. Raaijmakers and Richard M. Shiffrin. Search of associative memory. *Psychological Review*, 88(2):93–134, March 1981.

[38] H.-J. Schek and P. Pistor. Data structures for an integrated data base management and information retrieval system. In *Proceedings of the Eigth International Conference on Very Large Data Bases*, number 8 in VLDB, pages 197–207, Mexico City, Mexico, September 1982.

[39] Gabriele Sonnenberger. Exploiting the functionality of object-oriented database management systems for information retrieval. In Bulletin [8], pages 14–23. Special Issue on Integrating Text Retrieval and Databases.

[40] Robert W. Taylor. In memoriam: J. C. R. Licklider, 1915 – 1990. Technical report, Digital Systems Research Center, Digital SRC, Palo Alto, California, August 1990.

[41] C. J. van Rijsbergen. Information retrieval. Department of Computing Science, University of Glasgow.

[42] S. R. Vasanthakumar, James P. Callan, and W. Bruce Croft. Integrating inquery with an rdbms to support text retrieval. In Bulletin [8], pages 24–33. Special Issue on Integrating Text Retrieval and Databases.

[43] Marc Volz, karl Aberer, and Klemens Bohm. An OODBMS-IRS coupling for structured documents. In Bulletin [8], pages 34–42. Special Issue on Integrating Text Retrieval and Databases.

[44] World Wide Web Consortium. Jigsaw overview. http://www.w3c.org/Jigsaw/.