SOFTWARE REDESIGN OF THE HAYSTACK PROJECT

by

DAVID M. ZYCH

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June 2001

Author _____
Department of Electrical Engineering and Computer Science
May 31, 2001

Certified by _____
David R. Karger
Thesis Supervisor

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

To Mom and Dad,
who have faith in me even when I doubt myself.

SOFTWARE REDESIGN OF THE HAYSTACK PROJECT

by

DAVID M. ZYCH

Submitted to the

Department of Electrical Engineering and Computer Science

June 2001

In Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

Abstract

Most current large-scale information retrieval systems are impersonal, allowing uniform access to a non-modifiable corpus of information. The Haystack project is a personal information repository, which employs artificial intelligence techniques to adapt to the needs of its user.

This thesis outlines a new overall design for Haystack. The design includes an RDF-based data model, a transaction-based storage system, services which are triggered by patterns in the data, a kernel with the machinery to invoke and execute these services, and a query processing system.

In addition, the thesis serves as a valuable resource to future Haystack researchers by explaining and documenting the fundamental ideas of Haystack, as well as the reasoning for all design decisions.

Thesis Supervisor: David R. Karger
Title: Associate Professor

Acknowledgements

I would like to acknowledge and thank Professor David Karger for his sage guidance and unflagging support throughout this undertaking. His design suggestions have been invaluable to me, and his encouragement kept me going even when the project began to feel intractable.

I would like to acknowledge the rest of the Haystack research team, both past and present, for their hard work in producing so many of the good ideas that my own document has built upon. Without their contributions, this thesis would never have been possible.

Very special thanks go to my good friend Steven Niemczyk, who deserves a medal of honor for keeping me sane over the course of this past semester. His advice on the occult art of thesis writing has been indispensable on several occasions, and he has given me the moral support I needed whenever I needed it (i.e. all the time). Steve, I couldn't have done this without you.

I likewise appreciate the constant encouragement from my entire circle of friends and acquaintances; to all of you who urged, "work on your thesis!" every time you saw me this term, you have my thanks. To those of you who didn't, well, you have my thanks just as much for sparing me that little bit of pain.

I would like to thank my family back in Champaign, IL, for being the wonderful people that they are. Mom, Dad, Maria, and Veronica, it's the greatest comfort in the world to know that no matter where I go, what I do, or what happens to me, I will always have your love and support.

Most of all, I thank God for each and every moment of my life, and for the knowledge that nothing we do is ever truly meaningless.

Table of Contents

Chapter 1: Introduction

Current large-scale Information Retrieval (IR) systems, such as online search engines, are very impersonal both in behavior and in content; they allow uniform access to a generally non-modifiable corpus of information. Haystack, by contrast, is intended as a *personal* information repository; it stores mostly information that is of interest to the user, and it also stores information *about* the user. This allows it to support a more flexible space of queries, such as "the email about graduate housing that I received sometime last May" or "the document I was reading last Thursday afternoon about internet infrastructure" (conventional search tools could never process queries like these because they refer directly to the user, "I"). In addition, Haystack uses machine learning to deduce additional information from existing information, and also to customize itself so that it can adapt to better fulfill the needs of the user [6].

Haystack [5] is currently being rebuilt in pieces by a group of individual researchers, and needs a strong, coherent design to direct these rebuilding efforts so that the new pieces will fit together properly into a Haystack system that will be functional, elegant, and easily extensible in the future. This thesis provides such a design, as well as a thorough high-level discussion of Haystack in general.

1.1 Objectives

The primary goal of this thesis is to provide Haystack with a coherent, elegant, modular, and extensible design and system architecture. This will serve as a solid foundation for future Haystack research, particularly in areas such as machine learning and knowledge representation, by making it easy for future Haystack researchers to create powerful new components of Haystack.

The secondary goal of this thesis is to document as much as possible the thought processes that have gone into Haystack so far. This will allow future Haystack researchers to see the various design alternatives that have been considered and the reasoning for the choices that were made. By helping these future researchers avoid the problem of reinventing the wheel, this thesis will allow them to spend more of their valuable time considering only new ideas and new arguments, as well as focusing on the finer-grained issues of introducing new components into Haystack.

The final goal of this thesis is to serve as a teaching document for new members of the Haystack research team in order to familiarize them with the terminology, philosophy, and architecture and design ideas that they will need in order to eventually contribute their own pieces

of the Haystack project. To accomplish this goal, the document will discuss some of the basic ideas of Haystack in slightly more detail than might otherwise have been necessary for a design document.

1.2 Road Map

The remainder of this chapter gives a general overview of Haystack, and then discusses the requirements of this thesis as a design. Chapter 2 explains the Haystack data model in great detail. Chapter 3 presents the basic system architecture, also known as the "core" of Haystack. Chapter 4 discusses possible advanced features of Haystack (both specifically and in general), and outlines how these features could be implemented effectively using the data model and system architecture laid out in the previous chapters. Chapter 5 points out opportunities for related future work, and concludes the thesis. Appendix A details all of the interfaces defined and referred to throughout the thesis, and Appendix B contains the bibliography.

1.3 Haystack Feature Overview

This section describes the major features that might be associated with an *ideal* Haystack. Not all of these features actually exist right now even at the design level, and fewer still exist at the implementation level.

The most basic feature of Haystack is the ability to store and retrieve information. Individual data objects in a Haystack might be text, images, spreadsheets, or any other imaginable kind of data. Haystack provides a way to express the relationships between different objects, as well as the features and properties of an individual object. Finally, Haystack allows the user to view the Haystack graphically, and navigate (or "browse") through it in an intuitive fashion to look at data.

One thing that makes Haystack special is the fact that Haystack is personalized. It observes the user in action all the time, and uses these observations to learn about the user's interests. Haystack also accepts feedback from the user by allowing the user to specifically indicate his or her own interests, and to manually define relationships between objects. This feedback allows Haystack to learn even more effectively.

The most powerful and intriguing aspect of Haystack is its ability to *deduce* information. Employing a wide variety of techniques from machine learning and artificial intelligence, Haystack can deduce many properties of objects and relationships between objects. It can deduce information about the user's interests, both in general and with respect to a particular task that the user is currently working on. It can learn to identify features of an object that make the object

interesting to the user, and thereby deduce whether other objects will be interesting or not. Furthermore, it can attempt to package these deductions up into a *viewpoint* that embodies a particular notion of interestingness, and can be applied to different sets of data to determine which objects are interesting in a given context. For example, the statement "I like recipes with butter and broccoli" suggests a viewpoint in the recipe domain that could potentially be deduced from one set of recipes and then applied to a new set of recipes (e.g. a new cookbook) to predict which of the new recipes might be interesting to the user.

The practical applications of the features described above are most readily apparent in the area of filtering large amounts of information to identify those objects that are likely to be interesting. Haystack allows the user to perform powerful, advanced queries on what is essentially the user's entire collection of data, enabling him or her to find useful pieces of information much more quickly and easily than with conventional search tools. Haystack does make use of conventional search tools, however, to automatically expand its own corpus; it can seek out and import data from external sources such as the web and even other Haystacks.

The ultimate goal of Haystack is to serve as an intelligent, personalized information assistant. Simply by observing the user at work, it develops a model of the world which contains inferences about the kinds of things that generally interest the user, any project(s) the user is currently working on and the sorts of information that are relevant to those projects, and the behavioral quirks of the user. It uses this world model to effectively find interesting information in response to the user's queries, to filter the user's incoming information (such as electronic mail) according to projected interest level, and to proactively alert the user to information that is very likely to be of interest.

1.4 Design Requirements

This section discusses the scope of the design presented in this thesis, as well as several general guidelines for achieving good design (the guidelines which this thesis will endeavor to follow to the best of its ability). Together, these two things form the design requirements of the thesis itself.

1.4.1 Scope

The complete design of a system like Haystack is a tremendously large task. In order to make it more manageable, the scope of this thesis has been limited according to the following general guidelines.

1.4.1.1 Simplifying Assumptions

This design makes the assumption that computer memory, disk space, and processing power are in plentiful supply (though not infinite). The idea is to develop a system that does what we would like it to do, rather than worrying excessively about efficiency. Technology is advancing at a sufficiently rapid pace so that even if we wind up with a system that is excessively slow on today's computers, we will soon have better computers that will support the software. The most obvious example of this kind of thinking in Haystack (with respect to disk space) is that there is almost never any reason to truly remove a piece of information from Haystack; the common behavior is to keep everything. Clearly, this philosophy could never thrive in an environment where small hard disk capacities were a concern.

The second assumption made by this design is that Haystack, at least for the moment, will not be operating in a hostile security environment. We postulate the existence of very basic safeguards, such as operating-system-level user authentication and a filesystem with access control that will protect the user's data, but we will not attempt to make Haystack impervious to sophisticated malicious attacks.

1.4.1.2 Structural Design Only

This thesis is not about artificial intelligence, machine learning, query language specification, or any of the other specialized techniques that will be critical in implementing the new and interesting pieces of Haystack. Instead, this thesis provides a coherent design framework to handle the more mundane details of Haystack so that future Haystack researchers will be able to focus entirely on their areas of specialty in creating those innovative and exciting components, and not have to worry about whether the system will be strong enough to support what they want it to do.

In order to accomplish this goal, however, it will be necessary to consider the kinds of needs that advanced components are likely to have. To this end, once the key portions of the design have been laid out in Chapters 2 and 3, we will use Chapter 4 to address several feature sets that have been considered desirable in the past, and argue that these can be implemented as components which mesh very easily with the core system architecture.

1.4.1.3 Large Modules

The final necessary guideline for keeping this thesis at a manageable size is that, from an architectural standpoint, the goal will be to break the system down only into fairly large modules, as long as they are coherent and self-contained. The interfaces between modules should be

clearly stated, and should furthermore be as elegant and modular as possible, but it is not necessary to detail the internal design of each module down to the level of code specifications. If a module seems very large (as is the case with Persistent Storage, discussed in chapter 3), it may be desirable to *suggest* a way of breaking the module down further, but such details will be up to the implementer of the module; our goal is to define the abstraction barriers and the boundaries.

1.4.2 Qualities of Good Design

Before diving into the design with Chapter 2, it will be helpful to briefly enumerate some of the general qualities that make a design good. These are the qualities that we will be looking for as we explore the possible answers to each design question we are faced with.

1.4.2.1 Modularity

A modular design is one in which problems are solved by breaking them down into small, easily manageable tasks. Each individual task is accomplished by a single module, and no other module in the system performs that same task or depends at all on the details of that module's implementation, just so long as it performs the task it is supposed to perform. The consequence of modularity is that it is easy to change one module (if, for example, a better algorithm is discovered for sorting) without having to change any of the other modules. In certain kinds of systems (of which Haystack will be one), modularity also allows optional modules to be added to the system or removed from the system very easily.

1.4.2.2 Extensibility

The property of extensibility reflects the ease of adding completely new functionality to the system later, without having to redesign and re-implement everything else. This is particularly critical in a system like Haystack, because we really don't necessarily know as we're designing it what specific things we will eventually want it to do. Thus, the strategy is to make the design as extensible as possible, and satisfy ourselves that it is good by arguing (as we will in Chapter 4) that the classes of extra features we *can* currently think of wanting to add can, in fact, be added easily and straightforwardly.

1.4.2.3 Elegance

Elegance is a difficult property to explain in words, as it is a highly intuitive measure of the quality of a solution. The opposite of an elegant solution might be referred to as a messy

solution, or a clumsy solution; system designers develop strong intuitions for many of these words.

The concrete property that best explains elegance is simplicity. If a solution can be stated very simply, in words that are easy to comprehend and ideas that are easy to grasp, then it is an elegant solution. On the other hand, a complicated solution that takes a long time to explain and that doesn't intuitively make sense will be regarded as messy, usually because "it seems like there must be a simpler way to do that."

1.4.2.4 Coherence

A coherent design is one that appears to stem from one guiding philosophy, and one whose parts all make sense in the context of each other. In general, if there is a set of design questions that are all very similar to each other, a coherent design will answer them the same way unless there is very good reason to do otherwise. For example, one element of Haystack's guiding philosophy is the notion that *all* of the information Haystack uses is stored in the data model, along with the "actual" data. As we'll see, every time we need to represent some new status condition or relationship, we choose to use Haystack assertions. We could have chosen to represent some of these conditions with bit flags, or with a special table of kernel assertions that nobody else can see because it's hidden in a separate instantiation of the Persistent Storage module (described in chapter 3), but that would be at cross-purposes with the guiding philosophy of Haystack, and is therefore undesirable even if (in some cases) it might be substantially more efficient. The property of coherence is also sometimes referred to as Conceptual Integrity.

Chapter 2: Data Model

The Haystack Data Model is based on the Resource Description Framework (RDF), which is described extensively in a World Wide Web Consortium (W3C) Recommendation [2]. The data model begins with two fundamental concepts: **resources** and **assertions**. Haystack resources are essentially identical in definition to RDF resources, and assertions are very similar to RDF statements. Nonetheless, the Haystack data model is not just another name for RDF; rather, it is a layer built on top of basic RDF that is customized to better fit the guiding philosophy and semantic needs of Haystack. The sections of this chapter will explain resources and assertions in detail, describe the basic operations in Haystack for adding and removing data, and briefly discuss Haystack's philosophy on access control.

2.1 Resources

A resource, as in RDF, is basically any imaginable thing about which we might want to make a statement [2]. Resources range from the very concrete to the very abstract. Examples of resources might include a web page, an entire web site, a piece of text, a number, an MIT student, and the sound of one hand clapping. The notion of a resource is designed to be infinitely extensible. For the purposes of Haystack we will divide the resources into two subsets: Literals and Intangibles. These classifications are described later in this section.

Resources are named by Uniform Resource Identifiers (URI's) [2]. URI's can take many different forms. The URI of a web page (the intangible web page, not the HTML stored there – we'll come back to this in a moment) is typically the same as its URL, for example "http://web.mit.edu/dzych/", while the URI's for other kinds of resources may look very different. Just as in RDF, "anything can have a URI; the extensibility of URIs allows the introduction of identifiers for any entity imaginable." [2]

2.1.1 Literals

A **literal** is a resource whose entire identity can be expressed as a sequence of bits. This idea will become clearer when we contrast literals with **intangibles** in the next subsection. The most common literals are strings and numbers, but images, audio files, and many other things also qualify. Literals are immutable; if you change a literal it becomes a different literal.

In general, the first few bits of a literal are reserved for specifying a type. This allows easy identification of such primitive types as strings, integers, and floating-point numbers. An additional type code (the "non-primitive" type) is used for any literal whose type is not one of the

primitive types; normally such literals will have their types identified by assertions in the Haystack data model. It is of course permissible to include assertions that identify primitive types; however, these assertions will refer to other resources, which will themselves need types. To avoid infinite regression, at least some primitive types must be identifiable without the aid of an assertion.

Because literals are immutable and have their entire essences defined by strings of bits, there is no reason ever to construct two "different instances" of the same literal. Therefore it is impossible for duplicate literals to exist in Haystack.

Many of the literals referred to in Haystack are likely to be fairly large. The text of any document, for example, is a literal (it's a very long string), but we certainly would not want to have to embed that full text into every assertion that is made about it. To avoid this problem, Haystack stores every literal it needs to refer to, and defines a URI for it based on the MD5 cryptographic hash of its bitstring. For convenience, these URI's will be prefixed by the tag "md5://". This MD5 naming convention is beneficial because it enforces the invariant that no duplicate literals may exist, and even ensures that assertions made independently about the same literal in two different Haystacks will refer to that literal by the same URI.

2.1.2 Intangibles

Intangibles, in contrast to literals, are resources that defy adequate representation by nothing more than a sequence of bits. There are different kinds of resources that qualify as intangibles for different reasons; some of these (four-dimensional objects, aggregate objects, and abstract objects) will be discussed individually in a moment. However, all types of intangibles in Haystack will exhibit essentially the same behavior.

A given intangible is represented, not as a stored object that has data (the way literals are represented), but as *only* a URI. Assertions can still be made about the intangible just the same way they can be made about literals (an assertion needs only a URI to refer to), but any attempt to "get the data" of an intangible will fail because there is no such "data" defined for it.

The actual form of the URI's used for intangible resources is a decision left to the discretion of the entity creating them. As long as they don't overlap inappropriately with another defined URI space (for example, an intangible named by a URI that starts with "md5://" would not be acceptable), the only real requirement for most intangibles is that they have URI's that are unique. "Unique" in this context means that no two intangibles should have the same URI in the Haystack; it does *not* necessarily imply that each intangible has only one URI. One possibility would be to define a new space of URI's with exactly this philosophy, in which every URI would

consist of the prefix "unique://" followed by a unique string. Any given Haystack can provide a new "unique://" URI (one that does not yet exist in this Haystack) on demand, but the problem of mapping these URI's between different Haystacks in a sensible way (in order to import objects from one Haystack to another) is still unresolved.

2.1.2.1 Four-dimensional objects

Many resources, especially external (not part of the Haystack) ones, exist in four dimensions; that is, they have the capacity to change over time. These changes, in general, happen without Haystack's explicit knowledge. An example of a four-dimensional resource might be "David Zych's website". At any given time, this resource has a piece of HTML (a literal) associated with it that represents its current content, yet the essence of the website itself goes beyond the data that is currently there, and it is reasonable that we might want to make an assertion about the four-dimensional object itself and not about the HTML that happens to be there at some given moment. This concept is even more applicable to dynamically generated web pages, which may potentially have different content every time they are loaded (as opposed to static web pages, which change in content only as frequently as someone has time to change them).

2.1.2.2 Aggregate objects

Aggregate objects are resources that are fundamentally composed of other resources. The best way to explain this is by example. Consider the notion of a document. A document may have an associated title, author, blob of text, and (in a sufficiently thorough model) more complex properties like a revision history. None of these objects individually *is* the document, even though most people would agree that the text comes much closer to the essence of the document that anything else. In particular, it may be desirable to be able to change some of these properties (even, and especially, the text) and yet consider the result to be the *same* document that it was before. On that note, it is worth observing that aggregate objects are very often also four-dimensional; the distinction between the groups is made mainly for the sake of those objects that are four-dimensional but *not* aggregate.

2.1.2.3 Abstract objects

Abstract objects are something of a catch-all category, created to include any intangible object that does not fall neatly into the category of a simple four-dimensional object or an aggregate object. The basic idea of an abstract object is something that cannot even begin to be

properly represented by any amount of hard, concrete data. An individual MIT student is a perfect example; while he may have a name, an ID number, a transcript, a resume, and countless other pieces of information associated with him, none of these (even combined) come close to capturing the essence of his identity.

It is important to observe that the dividing line between abstract objects and aggregate objects is blurry to the point of being subjective. Clearly, a document (our example of an aggregate object) could also simply be thought of as an abstract object with properties. However, it is also possible to argue that, given a broad enough viewpoint, the MIT student can be thought of as an aggregate object (to see this, imagine constructing literals to represent his complete DNA sequence, the pattern of neurons in his brain, etc). The conclusion is not clear, but at some point the discussion ceases to be a systems design discussion and becomes a deeply philosophical discussion.

2.2 Assertions

An assertion is Haystack's version of an RDF statement. It expresses a relationship between two resources (a subject and an object) in the form of a triple: (subject, predicate, object). Each member of this triple is a URI, which represents a resource.

If any member of an assertion is a literal (i.e. has a URI beginning with "md5://"), then the literal must exist in Haystack in order for the assertion to exist. The reasons for this decision are described extensively in section 2.6.3; we will not go into them here.

Assertions are a subset of resources. Specifically, assertions are a subset of literals. Every URI is an immutable string, and an assertion is just an ordered triple of URI's (which can therefore also be represented as an immutable string). An important consequence of this is that assertions can be made about other assertions (but, because assertions are literals, it is not possible to make an assertion about a second assertion which does not exist in Haystack). Another important consequence of this is that "assertion" will be one of the basic primitive literal types (along with integer, string, etc); this ensures that it is possible to specify the type of a non-primitive using only primitives: (X, "type", "image/bmp").

Self-references (either direct or indirect) are not allowed. We say X **directly refers** (or just "refers") to Y if Y is the subject, predicate, or object of X. We say X **indirectly refers** to Y if X directly refers to Z and Z directly or indirectly refers to Y. Thus, it can never be the case that X directly or indirectly refers to X.

2.2.1 Differences from RDF model

There are several differences between the basic fundamentals of the Haystack Data Model, as described so far, and the RDF model. Firstly, RDF papers (e.g. [2]) choose to write out triples in text in the ordering (predicate, subject, object) rather than (subject, predicate, object). Haystack papers use the latter ordering, because we feel that it is more intuitive. This difference is entirely cosmetic; we mention it only to forestall a potential source of confusion. Secondly, RDF defines literals in a much more restrictive way; the object of an RDF statement may be a literal, but the subject may not [2]. Haystack does not make this restriction, and even permits assertions on two literals. Thirdly, RDF mandates that the predicate of a statement must come from a special subset of resources called Properties [2]. While we anticipate in general that the set of resources that are ever actually used as predicates will be very small compared to the overall set of resources in Haystack, we do not explicitly make any requirements to that effect, nor do we define the Properties subset. While it is true that an assertion with a predicate like "http://web.mit.edu/dzych/" seems odd and probably meaningless, it is allowed to exist in Haystack, in keeping with the guiding philosophy that irrelevant (and meaningless) data can simply be ignored. The last (and most important) major difference between this data model and RDF is related to Haystack's philosophy on reification, which will be explained by the next subsection.

2.2.2 Reification

RDF regards every RDF statement as a fact [2]. Suppose we want to create a statement (A) about another statement (B). In RDF, we cannot just create B and then have A refer to B, because this would necessarily imply that B is a fact. Clearly, there are many cases where this implication is far from desirable (for example, perhaps A states that "B is false"). RDF solves this problem by creating an entirely new resource (M) to *model* statement B. The elements of B (a subject S, predicate P, and object O) become properties of M due to three new statements which are created: (M, subject, S), (M, predicate, P), and (M, object, O). This process of modeling a statement is called **reification**, and the resulting model M is called a **reified statement** [2]. The reified statement M is not actually considered a statement as far as RDF is concerned, and therefore it is not expected to be a fact.

Haystack has a slightly different philosophy. The vast majority of interesting information in Haystack will take the form of assertions that are created based on machine learning and automation. These assertions are not facts and should not be considered facts; they are nothing more than educated guesses about what *might* be true. It is quite possible for such assertions to

be wrong, and it may even be the case that two different entities in Haystack (entities are defined in the next subsection) *disagree* on the truth value of an assertion.

One such piece of information could be the sentence: "The TypeGuesser service believes that the title of document X is Foo." The factual statement in this sentence (the one that RDF would choose to focus on) is about the TypeGuesser's belief, not about the title of X. The statement about the title of X would exist in RDF only as a reified statement. Haystack, however, considers "X's title is Foo" to be the more interesting statement in this sentence, and would like to be able to represent this directly as an assertion even though it is not known to be hard fact.

An additional justification for this philosophy is that the entities within Haystack are not completely arbitrary about the information they generate; the programmer of each entity worked hard to create an entity that would generate useful information as much as possible, and the entity probably would not have been put into Haystack if it had frequent tendencies to generate incorrect information. Therefore, our usual strategy in Haystack is proceed on the assumption that any assertion generated by another part of Haystack is probably correct, until and unless we are confronted with good reason to believe otherwise. As a rule, Haystack does not claim to know the absolute truth about anything.

The solution to the problem of reification in Haystack is as follows. Whereas RDF assumes by default that its statements are facts and has a special representation for statements that are reified, Haystack will assume by default that its assertions are reified and have a special representation for assertions that are factual. This special representation consists of the KernelFact assertion; any assertion X in Haystack that is actually considered a fact will have the assertion (X, "KernelFact", Kernel) attached to it to justify its existence. The kernel will be defined fully in Chapter 3; for now it suffices to say that the kernel is the master arbiter of information within Haystack, and that its assertions are authoritative.

An important caveat is that this solution doesn't actually apply to *all* instances of reification; it applies only to assertions that are reified for the purpose of stating that some entity believes them to be true. The restriction exists because it allows us to assume that any assertion in Haystack, while not necessarily factual, is at least "probably true" based on our philosophy of trusting the entities within Haystack. This is by far the most common case of reification in Haystack (which is why the solution is useful), but it does not work for other kinds of statements about statements. Consider, for example, the sentence: "The statement that David's website is good is a vague statement." Putting the reified statement "David's website is good" into the Haystack as such would imply that this is "probably true" when in fact the sentence we are given implies no such thing; the sentence is speaking to an entirely different issue (the issue of

vagueness). Because of this, Haystack would still require a more standard model of reification (like the one proposed by RDF) in order to represent this kind of reified statement.

2.2.3 Entities

Any statement that explicitly exists in the RDF model is considered by RDF to be a fact [2]. Haystack does not require that all of its explicit assertions be factual, but it does require that they be "probably true" (as explained earlier, in the reification discussion). An assertion is considered "probably true" if at least one **entity** believes it is true.

The definition of an entity is fairly vague, and comes mostly from the functional requirement stated in the previous paragraph; namely, that an entity is capable of making assertions and believing them to be true. One very common kind of entity is a Haystack Service (these are defined in Chapter 3) or even a set of several such services that were developed as a group to work in concert. The user is an entity. The kernel (also discussed in Chapter 3) is another.

The most important requirement for entities is that each entity must have a consistent set of opinions. As we've already mentioned, it is possible for one entity to believe that an assertion X is true while some other entity believes that X is false. It is also permissible for an entity to change its mind about an assertion over time (i.e. to believe that X is false where previously it believed that X was true). However, it should never be the case that a single entity disagrees with itself (i.e. simultaneously believes both that X is false and that X is true). If an entity seems in danger of doing this, what that means is that the entity needs to be split up into multiple different entities (which *are* allowed to disagree).

Entities in Haystack are abstract objects, and are represented as such. Most entities (particularly those consisting of services) should have properties indicating their name, version number, and author or creator. These three pieces of information are important to avoid collisions between distinct entities [7].

2.2.4 Affirmation and Rejection

Every assertion in Haystack (except factual assertions which are justified by KernelFact) exists because some entity believes that it is true. We refer to this belief as **affirmation**, and represent it with an affirmedBy assertion: (X, affirmedBy, E), where X is the assertion being affirmed and E is an entity that believes X is true. We use "affirmedBy" rather than reversing the direction and saying (E, affirms, X), even though it is slightly longer, because it places the focus on X (the interesting assertion) rather than on E.

It is not practical in general to allow one entity to remove assertions that were placed in Haystack by another entity. There are two main reasons for this. Firstly, removing the assertion will not stop the other entity from believing it, and so the other entity would almost certainly just create the assertion again. Secondly, in the majority of "disputes" among entities, there is no clear way to decide which entity is right and which is wrong. This leads to the problem of so-called "dueling services" in which one entity contains a service that continually and automatically tries to create a certain assertion, and a second entity contains a service that continually and automatically tries to remove that same assertion. This is not a desirable state of affairs; the solution is to devise a means whereby the second entity can actively express dissent *without* trying to remove the assertion.

An entity's act of explicitly expressing disagreement with an assertion is called **rejection**. We represent this with a rejectedBy assertion, which follows the same model as an affirmedBy assertion: (X, rejectedBy, E).

In our definition of entities we stated that an entity should never simultaneously believe and disbelieve an assertion; in our new terminology, this means that a single assertion should never be affirmed and rejected by the same entity at the same time. To enforce this invariant, we declare that the act of an entity E rejecting an assertion X cancels any previous affirmation of X by E, and likewise that the act of E affirming X cancels any previous rejection of X by E. This makes it easy for an entity to "change its mind" about an assertion.

It is worth noting at this point that the KernelFact assertion defined earlier, which identifies assertions that are considered to be facts rather than just "probably true", is very similar in meaning to an affirmation from the Kernel entity. In other words, the difference between (X, KernelFact, Kernel) and (X, affirmedBy, Kernel) is very slight. It would therefore be possible to design the system using only affirmations; we chose to create a separate KernelFact assertion only because it seemed simpler to conceptually distinguish factual assertions from "probably true" assertions on the basis of the supporting assertion's predicate (KernelFact or affirmedBy), rather than on the basis of the affirming entity. This is particularly relevant because these two predicates will be string literals, while the representation of an entity is defined only abstractly at the present time.

2.3 Removing Data

Data can be removed from the Haystack in a variety of different ways, for a variety of different reasons. The first subsection explains the normal desired behaviors, and the remaining subsections discuss the operations used to implement those behaviors (and, in the case of

expunge, to implement the other operations). However, in additional to the normal desired behaviors, it is important to note that all of the removal operations are placed at the direct disposal of the user to allow for damage control in the face of catastrophic errors or other unforeseen situations.

2.3.1 Desired Behaviors

This subsection describes the normal situations in which data should be "removed" from the Haystack. Beware that the term "removed" will mean different things in different situations.

2.3.1.1 Unjustified Assertions

In order to justify its existence in the Haystack, any assertion that is not factual (i.e. does not have a KernelFact assertion) must be affirmed by at least one entity. If at any time this is not the case, then the assertion should disappear from the Haystack (until such time, if any, as it is affirmed by an entity again). Any consequences of this assertion also disappear, because it is assumed by default that no entity should believe assertions that depend on an unjustified assertion. This behavior will be implemented using the delete operation.

2.3.1.2 User Rejections

The user is an entity, and may affirm and reject assertions like any other entity. Whenever the user rejects an assertion, (s)he should not have to see that assertion, or any of its consequences, anymore when viewing the Haystack. However, as long as at least one entity is still affirming the assertion, other entities should still be able to see it and make deductions based on it. This behavior will be implemented using the hide operation.

It is a seeming conceptual discrepancy that the user is able to make assertions invisible by rejecting them, while every other entity must continue to see the assertions it has rejected. This is explained by the fact that the philosophy of Haystack is to gather large volumes of information and then sift through it in an attempt to find the pieces of information that are interesting; the uninteresting and irrelevant information remains in the Haystack, and entities generally deal with it simply by ignoring it. However, the user is a human being with limited information processing speed, and is less easily able to ignore uninteresting information than the automated data services that make up most of the entities in Haystack. Thus the special behavior implemented on behalf of the user entity is intended simply to help the user ignore things more effectively, and does not represent any deeper distinction between the user entity and other entities.

2.3.1.3 User Disinterest

Ideally, Haystack machine learning will eventually become sophisticated enough that the risk of the user seeing something in which (s)he has no interest will be vanishingly small, even if such things exist in the Haystack. In reality, however, we recognize that there will be imperfections in the query system and other machine learning components. If it does happen that some particular object is being continually shown to the user because the artificial intelligence thinks it should be interesting, yet the user finds it completely uninteresting, the user should be able to manually ensure that (s)he will not have to see that object (or its consequences) anymore. As in the case of user rejection, other entities should still be free to see the object and make deductions based on it. The idea is that the user can dictate the terms of what the user sees without worrying that the consequences will affect the rest of Haystack's operation. Like user rejection, this behavior is implemented using the hide operation.

The difference between hiding and rejection by the user is somewhat subtle. In the above section on user rejection, we argue that user rejection should consequentially result in hiding (if the user rejects something, it follows that (s)he will not be interested in seeing it again). The reverse, however, is not true; the fact that the user is not interested in seeing an assertion does not necessarily mean (s)he believes that it is false. Hiding can also be applied to objects that are not assertions, whereas rejection has no meaning for non-assertions.

2.3.2 Operations

Haystack implements three different removal operations: expunge, delete, and hide. This subsection describes each operation in detail.

2.3.2.1 Expunge

Expunge is a very straightforward operation. When an object gets expunged, it is permanently and completely gone from the Haystack, just as if it had never been created at all. Expunge should be used in only two cases. Firstly, expunge is used by the kernel itself in order to implement the other removal operations (hide and delete), and for a few other select cases where deletion is not an appropriate alternative. Secondly, expunge can be used by the user to provide emergency damage control if Haystack encounters catastrophic errors (e.g. a machine learning component malfunctions and starts spewing bogus assertions).

2.3.2.2 Delete

Delete is the operation most commonly used when Haystack needs to get rid of something. The major difference between deletion and expunging is that a deleted object only *appears* to be gone from the Haystack; it can be undeleted later on. There is a special undeletion interface (implementing similar operations to those in the basic data interface) that allows deleted objects to be seen. This interface also contains the undelete operation used to restore deleted objects to the Haystack. If an attempt is made to create an object again from scratch while it is deleted, the result is the same as undeletion.

In general, deletion is only used to implement the behavior that unjustified assertions should disappear from the Haystack. Because of this, there is never really any reason to delete any object that is not an assertion. However, the operation is supported for all objects anyway, both for the sake of extensibility and because this is no more difficult than supporting it only for assertions.

Deletion is represented in Haystack with the deletedBy assertion. This assertion points to the cause of the deletion; sometimes this cause is an entity, but sometimes an object is deleted because of the deletion of another object (this is cascading removal, described later on in this section). In the latter case (the deletion of object A triggers the deletion of object B), the deletedBy assertion of object B points to the deletedBy assertion of object A (which itself points to the entity that deleted A). Undeletion simply consists of expunging an object's deletedBy assertion.

2.3.2.3 Hide

Hide is a different kind of operation that, as its name suggests, doesn't actually remove anything from the Haystack at all; it merely hides an object from the user. Hiding represents the user's desire not to see something anymore. Other entities continue to see the hidden object as there; agents that act directly on behalf of the user (i.e. the user interface) simply ignore it and do not display it to the user. The user may use this feature to avoid seeing an entire class of assertions (e.g. all assertions with the predicate "foo") by simply hiding the predicate itself; the cascading process (described below) will take care of the rest. Like deletion, hiding can be undone. Attempts are rarely made to create objects that are currently hidden, because most entities see that the object is there, and the user isn't likely to hide an object that would later prove interesting enough to create again. Nonetheless, if the user does attempt to create an object that exists but is hidden, the effect is the same as unhiding it.

The representation of hiding is an exact parallel of the representation of deletion. A hiddenBy assertion points from each hidden object to the cause of its hiding; sometimes this cause is an entity (almost always the user in this case), but sometimes an object is hidden because of the hiding of another object (this is cascading removal, described later on in this section). Unhiding simply consists of expunging an object's hiddenBy assertion.

2.3.3 Cascading Removal

Cascading removal means that when an object is "removed" from the Haystack, any assertions referring to that object are also "removed." This preserves the invariant that an assertion cannot exist if it refers to something that does not exist. The exact details of cascading vary slightly depending on the specific removal operation.

When an object is expunged, any assertions referring to that object are also expunged. This happens immediately, in the same transaction (transactions are introduced in Chapter 3) as the original expunge.

When an object is deleted, any assertions referring to that object are also deleted. The deletedBy assertion for each newly deleted assertion points to the deletedBy assertion for the original object that triggered the cascade. The reason for this is that it results in cascading *undeletion*: when the deletedBy assertion for the original object is expunged, the other deletedBy assertions that were created by cascade will now be expunged by cascade (thereby undeleting the other objects). Cascading deletions happen immediately, in the same transaction with the original deletion.

When an object is hidden, any assertions referring to that object are also hidden. This occurs, not because it would violate the model to have a non-hidden object referring to a hidden object (it wouldn't), but because the user will almost certainly not want to see any assertions that refer to an object that (s)he didn't want to see. The hiddenBy assertion for each newly hidden assertion points to the hiddenBy assertion for the original object that triggered the cascade; this results in cascading unhiding. Cascading hiding does *not* happen immediately; instead, a service (services are defined in Chapter 3) takes care of hiding assertions that refer to already-hidden objects. The reason for this is that new assertions that point to the hidden object may be created *after* the hide operation, and a service will be able to make sure that those new assertions also get hidden.

2.3.4 Removing Intangibles

The three removal operations discussed in this section can be applied to *any* resource, including Intangibles. An intangible exists in the Haystack only as a URI that allows the intangible to be referred to by assertions. Clearly, the notion of removing the intangible itself is immaterial; since there is nothing actually stored to represent the intangible, there is no work to be done in removing it. What does have meaning, however, is the cascading of that removal. Therefore, to remove an intangible from the Haystack is to remove all of the assertions that refer to that intangible (i.e. all assertions that contain its URI). The effects, as always, differ depending on which removal operation is used.

2.4 Access Control

The access control rules of Haystack are as follows:

- Read access is entirely free. Any entity can read anything from the Haystack, as long as it has the appropriate permissions from the operating system to read data owned by the user (this caveat applies to all access). Note that this iteration of Haystack is not designed to be multi-user and does not deal with multi-user access control issues.
- The creation of objects and regular assertions can be done by any entity. Assertions, however, must be affirmed or they will be automatically deleted due to lack of justification.
- The special assertion "KernelFact" can be created and removed only by the kernel (defined in Chapter 3).
- Each entity can affirm or reject assertions on its own behalf. It is worth noting that an entity A can also create, directly, an affirmedBy assertion suggesting that an assertion X is affirmed by some other entity B. However, this is not the same as B actually affirming X because the assertion (X, affirmedBy, B) will not be factual; it will merely be an assertion affirmed by entity A. This issue is discussed again in section 4.5.3.
- Expunging is restricted to the kernel and to the user.
- Other methods of removal (deletion and hiding) are restricted to the kernel, the user, and to kernel services (a special subclass of services with kernel privileges, discussed in Chapter 3).

2.4.1 Enforcement

The access control enforcement Haystack has in its current iteration is based entirely on abstraction barriers. Kernel-privileged operations, for example, will be enforced simply because the kernel will not provide any interface to the outside that allows those operations to be done. All other issues, however (such as checking the identity of an entity affirming an assertion) are currently not enforced at all. In future iterations of Haystack, entities might have cryptographic keys associated with them to provide a mechanism for authentication; in this iteration we simply trust entities not to masquerade as other entities.

2.5 Efficiency Optimizations

This section describes two specific ways in which, to improve efficiency (particularly when dealing with considerations of limited space), we may wish to *approximate* the behavior offered by the model rather than literally adhering to it. As a design, this thesis does not particularly endorse such optimizations (they exist in stark philosophical contrast to the ideas discussed in section 1.4.1.1), but we mention them here anyway for the sake of completeness.

One type of efficiency optimization is known as the "promise model". Haystack in general tries to derive as much information as it can from the information it has; this results in substantial space consumption which, while perfectly in keeping with our design philosophies, may cause present-day computers to run out of storage space quickly. The promise model proposes that it is not necessary to actually derive and store all of the information that we can derive, so long as we can "promise" to provide it dynamically on demand (i.e. when the attempt is made to look at it).

The second efficiency optimization has to do with deleted objects. In some cases, particularly when the objects are very large, it may be desirable to "garbage-collect" (expunge) an object that has no currently existing assertions pointing to it. This certainly saves space, but is a worrisome gamble because if a deleted assertion pointing to the now garbage-collected object subsequently becomes undeleted, it will then be pointing at a nonexistent object; this violates our model of assertions. Of course, there is another possible model that avoids this particular problem (see section 2.6.3 below), but it is less preferable from a design perspective.

2.6 Rejected Alternatives

The following design alternatives were considered and rejected, but were interesting enough to warrant inclusion in this document.

2.6.1 Delayed expunges

One rejected alternative was the idea that expunge operations shouldn't be carried out immediately, but should instead be scheduled to happen at some point in the future. The reason for this proposal was essentially to protect the user from expunging something accidentally by allowing a brief window of time during which it could still be retrieved. However, this proposal takes a very simple operation and complicates it greatly, and it also prevents the kernel from having quick access to the expunge operation. Other things that rely on expunge would also be delayed. Instead, the user interface should simply make it very difficult to expunge something by accident (a good idea anyway, since the user should use expunge rarely if ever).

2.6.2 Restricted Read Access

Another rejected alternative was the feature of restricted read access; that is, the ability to prevent one entity's private data from being seen by another entity. There are two possible reasons for this feature: information that is secret and should not be seen by others, and information that is irrelevant and uninteresting and that others would not want to see.

The basic idea of Haystack is to gather large quantities of information and use that information to deduce other new information. It may not be obvious, a priori, that a particular piece of information could ever be relevant or useful; nonetheless this does not preclude the possibility altogether. Any information that is irrelevant and uninteresting can and will simply be ignored by other entities; thus there is no real harm in making it publicly readable. It may be desirable in the future to allow entities to create database-style "views" of the data; this possibility is discussed in section 3.2.3 (Rejected Kernel Module: MetaFilter).

If any information is truly secret, then none of the interesting features of Haystack will be able to be used on it anyway (it is very difficult to deduce new information from a piece of information that is unreadable), so there is little point in storing it in the Haystack in the first place. If, for some reason, a piece of sensitive data absolutely must be stored in the Haystack, it should simply be encrypted before archiving.

2.6.3 Phantom Literals

Our design for the data model specifies that an assertion cannot be made about any literal that does not exist in the Haystack. The main reason for this is that the notion of a literal generally carries with it the notion of inlining. From a semantic point of view, there is no reason to represent literals separately from the assertions that refer to them; the assertion simply *includes*

within itself whatever literals it needs. An example of such an assertion with inlined literals might be: ("The cat sat on the mat", "hasWordCount", 6). Semantically, this is fine; there is no need to store each literal separately in Haystack and refer to it by an MD5-based URI. The reason for storing literals is simply that many of them are likely to be exceedingly large in size and have several different assertions referring to them, and it makes no sense (even with plentiful storage capacity and computing power) to constantly replicate this data. However, it would be semantically bothersome for the Haystack to contain (for example) an assertion like (<md5://KXP4416>, "hasWordCount", 6) if there is no record of what <md5://KXP4416> refers to. Assertions like these that refer to "phantom literals" have little meaning or value.

The rejected alternative to this design decision is of course to *permit* assertions to refer to these phantom literals. The primary advantage of this alternative is efficiency; if another Haystack has a literal that is very large, for example, this alternative allows us to import assertions about that literal without having to import the literal itself. Additionally, if all assertions referring to a literal were deleted, we could "garbage-collect" the literal to save space without worrying that undeleting one of the assertions would cause a violation of the model. However, knowing the URI of a literal does not allow us to reproduce the literal itself, and any assertion made about a literal has very little meaning when the literal is not accessible.

Proponents of the rejected alternative might argue that referring to literals whose data are not stored in the Haystack is no different from referring to intangibles. The difference we see is that, in the case of phantom literals, the fundamental identity of the literal *could* be in the Haystack, but isn't. By contrast, the fundamental identity of an intangible cannot ever be stored in the Haystack, precisely because intangibles are, by definition, intangible. There is no such thing as the "data" that <http://web.mit.edu/dzych/>, for example, refers to (the data that happens to be posted there at the present time is not the same thing). Also, since this "data" does not exist, there can be no semantic notion of inlining it into an assertion the way literals are semantically inlined, and therefore it makes perfect sense that the intangible is not explicitly stored in the Haystack.

2.6.4 Chain Deletion

A final rejected alternative was to represent the operation of undeletion, not by expunging the deletedBy assertion, but instead by deleting the deletedBy assertion. This would be represented by attaching a *second* deletedBy assertion to the first deletedBy assertion. Of course, deleting the object again would require *undeleting* the first deletedBy assertion, which would mean deleting the secondary deletedBy assertion, which would be represented by attaching a

third deletedBy assertion. The eventual result of this is that an object that had been deleted and undeleted many times would build up a chain of deletedBy assertions. The current status of the object is determined by the parity of its deletion chain; an even-length chain means the object currently exists, and an odd-length chain means the object is currently deleted.

This alternative was rejected because it is extremely cumbersome and does not provide any useful additional information. Representing undeletion by expunging the deletedBy assertion is much more elegant and simple, and solves the problem just as well.

Chapter 3: System Architecture

This chapter presents the basic system architecture, or "core", of Haystack. This architecture includes a transaction-based persistent storage system, services which are triggered by patterns in the data, a kernel with the machinery to invoke and execute these services, and a query processing system.

3.1 Persistent Storage System

Haystack data is stored in a persistent, transaction-safe subsystem called an RDFstore, which is designed to store generic RDF data for generic purposes. The storage system does not reflect any of Haystack's data model specializations; these can be found in the Basic Haystack Data Interface, which is discussed in section 3.2 and detailed in A.2. The RDFstore itself provides the interface detailed in A.1. Accesses to the RDFstore take the form of **transactions**. A transaction is a sequence of operations performed under the same transaction identifier (TID). Haystack will use integers for TID's. When all desired operations have been added to a transaction, the whole transaction can be **committed** (in which case the changes take effect) or **aborted** (in which case the changes are all undone). Like any transaction-safe system, the RDFstore has the following properties:

- **Atomicity**: a transaction is always either completely committed or completely aborted. It is never the case that some operations from a transaction will be executed and some will not.
- **Consistency**: each transaction takes the system from one consistent state to another. The result of executing several transactions concurrently should be equivalent to that of executing them in some serial order.
- **Isolation**: no other transaction can see a transaction in an incomplete (non-committed) state. Concurrent transactions are not aware of each other.
- **Durability**: once a transaction is committed, the changes it makes to the data will persist even in the event of a system crash, power failure, or other unpleasant circumstance. This is also referred to as **persistence**.

3.1.1 Sub-architecture

The internal architecture of the RDFstore can conceivably be broken down in different ways. The particular choice made may have a large impact on possible performance optimizations and other internal issues, and so it is critical that the rest of Haystack rely only on

the specification for the overall RDFstore abstraction. Nonetheless, for the sake of completeness, this subsection will present one fairly plausible way to subdivide the RDFstore into a hierarchy of smaller modules.

3.1.1.1 ObjectStore

The ObjectStore is essentially a transaction-safe coat-check. It accepts objects (coats) in the form of bit strings and archives them, returning keys (claim tickets) that can be used later to get the objects back. Note that the ObjectStore, not its client, specifies the keys.

3.1.1.2 TripleStore

The TripleStore is like an ObjectStore except that instead of storing bit strings, it stores ordered triples of keys. The TripleStore is implemented as a layer of encoding on top of the regular ObjectStore. As in the ObjectStore, the TripleStore specifies a key for each triple that can be used later to get the triple back.

3.1.1.3 URI Association Table

One important property of the RDFstore is that unlike our ObjectStore and TripleStore, it allows its client to specify the URI's under which its objects and triples are stored (in the case of Haystack, these are mostly "md5://" URI's). We achieve this behavior by implementing an association table that maps the URI's chosen by the user to the keys chosen by the ObjectStore and TripleStore. The association table is stored within the TripleStore so that it, too, will be persistent and transaction-safe. One possible representation of the table (there are others) is simply a collection of key-URI pairs joined by "hasURI" assertions.

If (as seems likely) our ultimate implementation of the RDFstore actually contains a separate TripleStore and ObjectStore, then in general it may happen that both stores use the same key (the TripleStore to store some triple, and the ObjectStore to store some object). In order to avoid such problems with key collisions, the association table must also store, along with each key, an assertion indicating whether that key should be looked up in the TripleStore or the ObjectStore.

3.1.2 Rejected Alternatives

The following design alternatives were considered and rejected, but were interesting enough to warrant inclusion in this document.

3.1.2.1 Storage Layer Query

One rejected alternative was to have the storage layer actually do the work of performing queries (queries are discussed later in this chapter). The reason for this proposal is that the storage layer is very likely to be implemented on top of an existing database system, which would have efficient search features that perhaps could be taken advantage of to make Haystack queries more efficient. The proposal was rejected because it is highly inelegant in that it makes the storage system less specialized and therefore less modular, and makes it much harder to integrate higher-level components of Haystack that change the way queries are processed (which is something we will almost certainly want to do, since queries are the single most important and interesting feature of Haystack).

3.1.2.2 Multi-user Storage Layer

A second rejected alternative was to create a storage layer that was intended to support multiple users. Such a storage layer would include notions of data ownership, access control, defining the interactions between different users' data, and other related issues. This proposal was rejected simply because this functionality is not necessary in order to support Haystack's needs, and would require substantial additional work in both the design and implementation.

3.1.2.3 Thread-based Transactions

Yet another rejected alternative was to entirely avoid the necessity of transaction identifiers (TID's) in the interface by simply associating one transaction with each Java thread that accesses the RDFstore. This proposal was rejected for two reasons. Firstly, it ties the interface firmly to a specific implementation language, which severely reduces its extensibility and reusability. It is quite conceivable that in the future we may want to allow non-Java programs to access an RDFstore; if we initially provide an interface that assumes Java thread structure for dealing with transactions, then we will have to provide a new interface in order to support other languages. Secondly, the freedom to create multiple concurrent transactions from the same Java thread may prove useful later on, and allow us to avoid unforeseen difficulties.

On the other hand, while it clearly is not appropriate for the storage layer to associate transactions with threads, the idea may be a useful and convenient syntactic sugar for other interfaces in higher levels of the system. Individual services, for example, should not have to deal with transactions explicitly.

3.2 Kernel

The Kernel is a symbolic name given to the collection of modules and services (these are "kernel services", not all services) that are integral parts of the Haystack system, working together to ensure the integrity of the data model and provide the basic functionality of Haystack. This section will describe the two major modules of the kernel, which are the Registrar and the Execution Queue. In addition to these two modules, the kernel provides the Basic Haystack Data Interface (detailed in A.2) that is used by the rest of Haystack to access the functionality of the data model.

3.2.1 Registrar

Before we can describe the functionality of the Registrar, we must first define the concept of a **service**. A service is essentially a procedure that should be run whenever a certain pattern is encountered in the data. Patterns in Haystack are discussed more fully later in the chapter (section 3.3.2.1); for the purposes of this discussion, a standard intuitive definition of the English word "pattern" will suffice. Every service is part of an entity, and its procedure may perform arbitrary reads and writes on behalf of that entity. All such reads and writes for a single invocation (of a single service) occur in a single transaction. A service is *not*, as the common usage of the word in other systems might mislead some readers to imagine, a continuously running process that can communicate back and forth with other processes; it is only a procedure, invoked on demand.

Services are divided into kernel services and data services. Kernel services are part of the kernel and act with kernel privileges on behalf of the kernel entity; they exist because there are certain tasks the kernel must perform for which the service model is very convenient. Data services (the vast majority) consist of all other services, and act with only the privileges afforded them in the access control section of Chapter 2; they exist, in general, for the purpose of automatically deducing information from other information.

The **Registrar** is a kernel module whose job it is to handle the triggering of services. Each service is registered with the patterns on which it should be triggered. The Registrar is notified of all write operations performed on the Haystack. Whenever one of the registered patterns is detected in the data (as the result of a write operation), the Registrar creates an invocation for the service on this pattern match and adds it to the execution queue (this process is explained in the next subsection). Optionally, when the service is first registered, the Registrar may use a query to find all currently existing instances of the pattern and immediately trigger the

service on these matches. This is left optional for two reasons: firstly because it is conceivable that some services might not want the feature, and secondly to avoid circular dependency between the Registrar and the query system (which is itself implemented using services).

3.2.2 Execution Queue

An **invocation** is an aggregate object in Haystack that is constructed to represent a single triggering of a specific service on a specific pattern match in the data. It is basically an unevaluated procedure call, created by the Registrar to be executed by the execution queue.

The **execution queue** is the kernel module that keeps track of all invocations that need to be run, and runs them. Because services are not necessarily trusted, there is in general no guarantee that a service will terminate in a reasonable amount of time (or even that it will terminate at all); nonetheless it is the job of the execution queue to make sure that every service is given a fair chance to run eventually.

The design of the execution queue is intended to be as simple as possible while appropriately handling two main classes of problems: the common case, in which services run in a reasonable amount of time and terminate properly, and the exceptional case, in which services may need to run for an extremely long time before termination (handling this case "appropriately" means that any invocation that terminates within a finite amount of time should eventually be allowed to do so, and this should not interfere with the quick-turnaround handling of the common case, even if there are some invocations in the queue that will never terminate at all). Our solution to this problem actually uses two standard FIFO queues, both of which are stored in the Haystack data model so that, like the rest of the Haystack data, they will be preserved in case of external failures.

In addition to regulating the time used by service invocations, it may also be desirable to limit the amount of *space* they consume, and to terminate them if they attempt to write more than some predetermined amount of data. This is not part of the current design for two reasons: firstly, it is not yet clear that there is any particular reason to want to do this, and secondly, limiting the running time of a service does technically limit the amount of space it can consume simply because it is only possible to write so much data per unit time (this is a very high limit, but it is unclear that any lower limit would be desirable).

3.2.2.1 Short Queue

The **short queue** is designed to handle the common case quickly. All incoming invocations from the Registrar are initially placed in the short queue. An execution loop takes

invocations from the front of the short queue one at a time, and runs them for a certain fixed amount of time (e.g. 1 minute) called the **time bound**. If the invocation terminates within its time bound, then all is well. If not, then its thread is killed, its transaction is aborted (recall that all operations performed by a single invocation of a service occur within a single transaction), and the invocation is added to the long queue. The time bound for each service in the short queue is chosen by the programmer of the service and passed as an argument to the registrar when registering the service for triggering. It should be enough time for "reasonable" invocations of the service to complete execution, but not much longer.

3.2.2.2 Long Queue

The **long queue** is designed to handle the exceptional case. Every invocation in the long queue has an associated time bound, which doubles upon every failed execution of the invocation. When an invocation is first added to the long queue (from the short queue), it has an initial time bound of twice the time bound it had in the short queue. An execution loop (running in parallel with the execution loop for the short queue) takes invocations from the long queue one at a time, and runs them. If an invocation does not finish within its allotted time bound, its thread is killed and its transaction aborted, its time bound is doubled, and it is put back at the end of the long queue with its new time bound.

3.2.2.3 Administrative Tools

Since both service queues are stored in the Haystack, they can be accessed by the user if need be. The user interface should include an administrative tool with which a knowledgeable user can monitor the long queue, look at the invocations and see how long they have been running, and cancel (remove) any that appear hopeless.

3.2.3 Rejected Kernel Module: MetaFilter

One idea for a possible kernel module, which was considered for a long time and eventually rejected, is the MetaFilter. The idea of the MetaFilter was to create a generic, customizable data filter to implement so-called exclusion rules of the form: "Things that match the pattern X should be hidden from all queries and data requests except for those matching the pattern Y." Each exclusion rule would have the effect of making certain kinds of data invisible unless it was asked for specifically (in a query matching pattern Y).

The MetaFilter is desirable because it would make many things simpler. It would implement the behaviors of deletion and hiding for free in a single exclusion rule for each ("Don't

show anything that has a deletedBy assertion attached to it"), and at the same time also provide the extra interface needed to see deleted resources ("unless the query has an includeDeletedItems assertion attached to it"). It would make the implementation of Time Versioning and Content Versioning (these are advanced features, discussed in Chapter 4) very clean and elegant, and would indeed support the later addition of *new* modules that need to prevent certain kinds of data from being seen under most circumstances. Like most generic mechanisms, the MetaFilter would be a great asset to extensibility.

In certain cases, the MetaFilter could also serve as a substitute for more sophisticated access control. An entity wishing to store its own private data about objects in the Haystack could simply set up an exclusion rule which would hide that data from any entity that doesn't know how to ask for it specifically. The kernel, in particular, would be able to hide all of its specialized data (such as the service queues) from other entities with a simple exclusion rule.

3.2.3.1 Rejection Rationale

Despite its great appeal from the point of view of extensibility and modularity, the MetaFilter was ultimately judged too expensive in time and effort to pay for itself. One contributing factor to its downfall was the decision to make virtually all Haystack data readable to everyone, rather than allowing private data to be hidden from other entities; this decision made moot a variety of problems that the MetaFilter would have otherwise been extremely well equipped to solve. It is also the case that features such as Time Versioning and Content Versioning, while it would be nice to implement them in a completely decoupled fashion from the lower levels of the system, are in some sense fundamental enough that having to integrate them a bit more tightly is not a disaster. Overall, the most powerful factor in this decision was simply the awesome magnitude of the task of building a generic filter for semi-structured data, using a pattern language that hasn't even been chosen yet.

Perhaps in the future, this decision will be re-evaluated and the MetaFilter incorporated into Haystack. It has many very nice properties from the perspective of design elegance, and it is quite possible that as Haystack evolves, we will discover that we want to implement an even wider variety of features for which the MetaFilter would be helpful. Another closely related possibility for the future is to implement a generic filter that is applied, not by default, but on a voluntary basis (by each entity looking at the Haystack), in order to create a database-style "view" on the Haystack data. A "view" in database terminology is a virtual table whose data consists of the results of some query being dynamically executed on another table. A view in Haystack would be a virtual Haystack consisting only of the data that matches a certain pattern.

3.3 Query

Queries are the single best embodiment of the purpose of Haystack; a query is the user's primary avenue of retrieving useful information.

3.3.1 Types of Search

There are several different types of search that could be relevant to Haystack. A fuller treatment of this material can be found in Svetlana Shnitser's recent thesis [1]; this subsection mainly summarizes the conclusions made there.

Text search finds documents (blobs of text in this case, not aggregate objects) that are "about" some topic. The topic is usually specified in the form of one or more keywords, and the most common approaches to text search (though more sophisticated approaches exist) rely on the heuristic that a document is generally "about" some keyword if the keyword appears frequently in the document.

Structured search is the type of search usually associated with relational databases. The data being searched is structured data conforming to some scheme, which means that every object in the search space has some certain set of attributes. The goal of the structured search is to find objects that have attribute values satisfying particular constraints.

Semi-structured search is like structured search except that the data being searched is semi-structured rather than structured. What this means is that the data does *not* necessarily conform to a rigid schema; the attributes that an object may have are not necessarily known ahead of time, and some objects may not have values for even the most common attributes. A paper by Serge Abiteboul [8] contains a detailed discussion of semi-structured data and queries.

What Haystack really needs, according to Shnitser [1], is not any single one of these search types, but rather a form of **hybrid search** that combines text search and semi-structured search into one. This can be effectively achieved by assigning numerical scores for text search and semi-structured search, and combine those into a total numerical score. An advantage of this model is that it allows for adaptation based on machine learning; the method of combining the different scores (perhaps a weighted average) can be tweaked artificially to improve the results.

3.3.2 Formulation

A query in Haystack is represented in the data model as an abstract object. Various assertions tied to the query object indicate the parameters of the search (both text and semi-structured). As the query is processed, the results will be tied to it with "result" assertions. This

notion of a query is extremely extensible; as new kinds of query parameters are developed, they are simply added to the query object as more assertions (and any older pieces of the query processing system can still process the new queries simply by ignoring the parameters they don't know how to deal with).

3.3.2.1 Patterns

The semi-structured aspect of a query is specified by a **pattern**. Patterns are to Haystack data essentially what regular expressions are to text data. In addition to using patterns for semi-structured query, Haystack also uses them for service registration. Each pattern has a "root object" identified; the URI of this object is used as a pointer to the pattern as a whole. When a pattern is being matched to pieces of data, a new copy of the pattern is created in Haystack to represent the matching. Each element of this pattern copy is connected to the appropriate element in the matched data with a "filledBy" assertion, and the pattern match can be referred to by the URI of its root object. This representation of a pattern match solves the potential problem of having multiple different pattern matches in the data that hang off of the same instance of the root object. By keeping these matches in the data model, it also provides us with an easy way to keep records of service triggerings (as it is this filled-in pattern that becomes part of the invocation for a service).

3.3.2.2 Pattern Language

This thesis will not even attempt to delve into the issues associated with defining a semi-structured query language. This is a very hard problem, and will require specific attention in the future. One similar project that has met with some success is the Lore project [3] at Stanford University. The Lorel query language, part of Lore, uses strong coercion and path expressions to represent semi-structured queries in an elegant form that is reminiscent of SQL [4].

3.3.3 Execution

To execute a query, first assemble the query object with all of the desired query parameters, and then attach a special "queryReady" assertion to it to signal that the query is complete. The query manager service, which is registered to detect all instances of the "queryReady" assertion, will find the query and start processing it. When the query manager has completed its job, it will attach the "queryCompleted" assertion to the query object. The reason for using assertions to signal the beginning and end of a query is so that query processing can occur effectively in the background; we can begin a query, go look at other data elsewhere, and

then come back to the query later (once it is finished) and examine the results. It also allows other services to notice the beginnings and ends of queries; this may be useful for invoking features such as clustering (described in section 4.1.3) on the results of queries.

It is very likely that the query manager service will be implemented in such a way as to delegate query-processing responsibilities to specialized "minion" services. The most obvious example in the case of hybrid search is that one service might know how to handle the semi-structured portion of the query, while another service knows how to handle text-based search. Each of these would produce a set of numerical scores, and it would then be the job of a third score-combining service to determine the overall rankings. However, to promote good modularity and extensibility, it is crucial that detailed decisions such as these be left to the discretion of the query manager service.

It is also likely that some of the machinery used by the Registrar to identify patterns in the data in order to trigger services can be reused as part of the query processing system; after all, the pattern language will be the same for both. However, it is essential that the service triggering mechanism in the Registrar not *depend* on the query system, because the query system is implemented using services.

3.3.4 Rejected Alternatives

The following design alternatives were considered and rejected, but were interesting enough to warrant inclusion in this document. Both provide alternative ways to combine text-based search and semi-structured search (instead of the chosen mechanism, which is combining numerical scores).

3.3.4.1 Text Search Indexing in Haystack

The first rejected alternative for combining search types is essentially to represent text-based search results as part of the semi-structured data, and then just use semi-structured search. The notion is that since most text-based searches work by creating indexes that help them find relevant data more quickly, it might be possible to implement one that writes its indexes in the form of Haystack assertions. These assertions would serve as an indication of what topics a particular blob of text is "about", and then one of the parameters of a semi-structured query might be the presence of the assertion: (X, isAbout, "cars"). It would even be possible to represent finer granularity results by attaching (to the isAbout assertion) another assertion specifying the quality of a topic match as a numerical score.

This alternative was rejected for two reasons. Firstly, it doesn't truly seem practical even in an environment that assumes plentiful storage and computing power. Secondly, dodging the issue this way for text search is only a temporary win because we would eventually like to imagine allowing more than two types of searches (for example, an image search for pictures of trees, or an audio data search for songs that have guitar solos).

3.3.4.2 View-based Hybrid Search

The second rejected alternative is almost exactly the opposite; its idea is to create a database-style "view" based on the semi-structured portion of the query, and then to use text-based search only on the data in that view. It was rejected for similar reasons. Firstly, text search on a dynamically generated set of data is woefully inefficient because indexing is almost completely ineffective. Secondly, this alternative fails to provide extensibility to additional forms of query even more than the previous alternative. Finally, this forces the semi-structured search to produce Boolean results: either an object is included in the view or it isn't. There is no way to represent the notion that some matches are better than others.

Chapter 4: Advanced Features

This chapter will discuss how we can elegantly implement the kinds of high-level features we want for Haystack, using the system architecture and data model laid out in previous chapters. The specific details of the high-level features are not the focus of this thesis; our goal is simply to show that they can be implemented better and more easily because of the design work we have done for the core system.

4.1 Machine Learning

As we saw in Chapter 1, machine learning (ML) is at the heart of what Haystack is all about. Many specific aspects of Haystack will contain ML components; some of these are discussed in various places throughout this document. The objective of this section is to discuss the role of ML in general.

The most common ML components will involve one or both of the following tasks: observation and deduction. In general, our design for Haystack strives to give the designers of these ML components as much freedom and power as possible. The suggestions made in this section represent only guidelines and possibilities, not rules and boundaries.

4.1.1 Observation

The task of observation is not strictly ML in and of itself, but it is closely tied to ML nonetheless. Observation components, as their name suggests, are responsible for observing something outside of Haystack (very often the subject of observation is the user), and recording these observations in the Haystack. The actual process of observation is outside the scope of this thesis; we are concerned about the ability of an outside application to report data to Haystack.

The Basic Haystack Data Interface (see 3.2 and A.2) can be accessed from outside as well as from within (provided that the operating-system-level security protecting the user's data is satisfied that the outside application has permission to represent the user). This interface allows any entity to write and affirm assertions (and to create literals) in the Haystack. Therefore, once the user observation component in question is assigned an entity, it should have no trouble recording its observations.

4.1.2 Deduction

The task of deduction is the major goal of ML in Haystack. Deduction components look at data in the Haystack, draw conclusions from it, and record their conclusions as new

information in the Haystack. The notion of a service was designed with precisely this type of behavior in mind. In general, we postulate that an individual deductive component will be interested in very specific classes of data; these interests can be described as semi-structured patterns, and registered so that the service will be automatically triggered upon the discovery of something fitting its definition of interest.

Some deductive ML components may also have substantial volumes of auxiliary information to store. This is no problem for Haystack. Annotations an ML service creates about pieces of data in the Haystack can easily be stored as assertions about those pieces of data, while generic state information for the service itself can be stored as assertions about the entity to which the service belongs.

One general consequence of the philosophy of storing all information within the Haystack data model is the problem of potentially cluttering up the user's view with lots of not-necessarily-interesting information. Auxiliary information from ML deduction components falls into this category (along with many other types of information retained by Haystack). Solutions to this problem appear in several different areas. Firstly, even the information that *is* likely to be interesting will exist in huge volumes, and it is the job of the user interface (discussed toward the end of this chapter) to find a way to display it for the user without overwhelming him or her; this responsibility extends just as easily to information that may be less likely to be interesting. Secondly, the user can intentionally employ the hide operation to prevent certain things from being shown to him or her in the future (as mentioned in section 2.3.2.3, it is possible with only one hide operation to avoid seeing an entire class of assertions). Finally, if the MetaFilter (a rejected alternative, described in section 3.2.3) were reincorporated into Haystack, it would provide a clean and elegant way of effectively shielding all other entities from one entity's auxiliary information.

4.1.3 Example: Clustering

Clustering is one straightforward example of an ML deduction component that has been proposed as a useful feature for Haystack. The basic goal of clustering is to organize the results of a query (or even just to organize existing data in general) into intuitive categorical groupings. Such a deductive component might use pattern registration to detect the kinds of information collections (the results of query objects, for example), and then assign them to groups in the Haystack by attaching to each item a "group" assertion pointing to its new group. Groups themselves could be represented as abstract objects, and would probably also have names (attached to the groups with "name" assertions).

Obviously, in order to actually make use of clustering in displaying the results of a query to the user, the user interface would have to understand the significance of the "group" assertions made by the clustering component; otherwise, the user would not be able to actually see the groupings in an intuitive fashion. This is a common problem for many systems; in general, whenever a new feature is added to a system, the user interface must be modified at least slightly to take advantage of this feature. Thus, the user interface should be designed in such a way as to make changes to it as easy as possible.

4.2 Data Dependency

One key notion in the data model is that we would like assertions to exist only if some entity believes them to be true. In many cases, an entity will not be well equipped to take the initiative in periodically revisiting the assertion it affirms to make sure it still believes them. Nonetheless, it would be beneficial for Haystack to force assertions to be reconsidered whenever a change occurs that has a reasonable likelihood of affecting that assertion. The idea of data dependency is to document the reason(s) why an entity affirmed an assertion and, if any of those reasons later cease to be true, to re-evaluate the affirmation.

This notion is related to the process of cascading removal described in section 2.3.3, but there is a significant difference. Cascading removal is a fundamental part of the Haystack core, and deals with propagation of removal operations only to assertions that *refer* to the object that is being removed (in all cases except hiding, this is necessary to adhere to the invariant that assertions cannot refer to literals, including other assertions, that do not exist in the Haystack). Data Dependency is a modular, *optional* feature that extends this notion to a completely new level, arguing that we should re-evaluate any assertion that was affirmed *because* of the object that is being removed (not just those that directly refer to the object).

4.2.1 Reason Tracking

The proper model for a reason is not immediately obvious. With sufficient thought and a lot of machine learning, it could be made arbitrarily complex. For now, however, we will consider the following simple model: a reason consists of a set of one or more assertions such that the presence of all of them in Haystack is sufficient for the entity to affirm the assertion (this is a logical AND of the assertions). If there are several different reasons for an affirmation, then the affirmation is considered reliable as long as at least one of the reasons is satisfied (this is a logical OR).

Clearly this simple model does not address everything; in particular, observation components (as described in the previous section on ML) may create assertions that depend, not on anything inside the Haystack, but rather on data that exists *outside* the Haystack. However, we will continue on with this model for now, in order to discuss the other things needed for data dependency to exist.

A reason is represented in Haystack by an abstract object. It has a "member" assertion connecting it to each assertion that is part of the reason, and the affirmation itself is connected to the reason by a "because" assertion.

There are two basic strategies for determining reasons. One is simply to have the entity state its own reason(s) when making the affirmation. This strategy is guaranteed to produce highly accurate results, but is extremely undesirable because it places a great burden on every assertion-making entity to change its own behavior due to the presence of the data dependency system; this is not the mark of a modular, well-designed component.

The other strategy, which lets assertion-making entities remain blissfully unaware of the data dependency system, is to attempt to deduce the reasons for an affirmation by making observations. In this case, the information available that may be of use is the transaction record. By looking at the read operations performed in the same transaction as the affirmation, the data dependency system can conclude that the set of all the resources that were read probably constitutes a reason for the affirmation. Of course, this alternative still allows entities to optionally specify their own (additional) reasons simply by adding another "because" assertion.

4.2.2 Re-evaluation

The next idea we need is a clarification of what it means to "re-evaluate" an affirmation. Semantically, the ideal behavior is that after a change in the data, Haystack should somehow determine whether the entity would still affirm the assertion if it had the choice to make over again, starting from the new data. If so, then the affirmation remains valid; if not, then the affirmation no longer belongs there and should be removed.

Practically, this behavior as such is very tricky to implement. However, because of the way data removal works in Haystack, we can simulate this behavior very closely by simply deleting the affirmation that is suspected of being invalid. If the affirmation truly is valid, then the entity will simply affirm the assertion again on its own, thereby undeleting the affirmation and also (by virtue of cascading undeletion) automatically undoing all deletions that cascaded from our original deletion of that affirmation. If the affirmation is not in fact valid, then it will simply remain deleted (which is the correct behavior in that case). This solution requires the ability to

trigger services on delete operations as well as on regular writes. However, this is not a problem because the end result of a delete operation *is* a regular write; deletion is represented by a deletedBy assertion, which must be written to the Haystack.

The last problem is that of detecting unsatisfied reasons. This is made easy by our representation. Cascading removal ensures that if one of the members of a reason is removed, the "member" assertion connecting it to the reason itself will also be removed. Therefore, a reason becomes unsatisfied when any of its "member" assertions are removed.

4.3 Time Versioning

The idea of time versioning is to keep track of when pieces of data in Haystack were created and deleted and, using this information, to allow the user to look at the Haystack as it was at any particular point in the past.

4.3.1 Time Tracking

Every resource in Haystack except for factual assertions (i.e. except for assertions that have "KernelFact" attached to them) has a "creationTime" assertion attached to it that indicates the time at which that resource was put into the Haystack. Every deleted resource also has a "deletionTime" assertion attached to it; this indicates the time at which that resource was deleted from the Haystack. These creationTime and deletionTime assertions are themselves factual.

When a deleted object is undeleted (or created again, which is equivalent), it gets a second creationTime. If it is deleted again, it gets a second deletionTime, and so on. An object that has been deleted and undeleted many different times has the history of its existence recorded in the form of multiple creationTime and deletionTime assertions. Therefore, given an object X and a time T, we say that X existed at T if and only if:

1. X has at least one creationTime before T.
2. If X has a deletionTime T_1 before T, then X also has a creationTime T_2 that is after T_1 and before T.

To determine creationTimes and deletionTimes, simply modify the implementation of the basic data interface so that all operations that create and delete objects also put in creationTime and deletionTime assertions (pointing to the time at which the operation takes place).

4.3.2 Crystal Ball

To allow the user to look back in time, the time versioning system must include a specialized enhancement for the user interface. This extra module hooks into an interface called

the crystal ball, which serves as a filter to let through only resources that existed in the Haystack at some particular time T. The crystal ball is an additional layer built on top of the deletion-enhanced data interface, and implementing (as usual) the same read features as the basic Haystack data interface, as well as an operation to set the time T to which the crystal ball is attuned. It does not re-implement write operations, however; under no circumstances does time versioning allow the user to change the past. All write operations must be carried out in the present time.

In addition to the simple data operations afforded by the crystal ball interface, time versioning should allow the user to perform a query on the set of information that existed at some particular point in the past. Intuitively, this feature might be referred to as making a query "through the crystal ball", and this is exactly how we will implement it. First, we add a method to the query formulation interface that assigns a time parameter to the query. This parameter is, of course, represented as an assertion on the query object; if the assertion is not present in a query, then the default value assumed is the current time. Secondly, modify the query manager so that it can read this time parameter from the query and use it to attune a crystal ball to the appropriate time. Finally, have the query manager perform all of its reads through the crystal ball interface rather than the standard interface.

4.4 Content Versioning

The idea of content versioning is to recognize the relationship between an old version and a new version of something (e.g. a document). The purposes for which this information will be used are unspecified as of yet, but some possibilities are described later in this section. Objects directly stored in Haystack are literals, and it makes no sense to talk about one literal being a new version of another literal; therefore, the kinds of objects for which content versioning is relevant are intangibles: four-dimensional objects, abstract objects, and aggregate objects.

One problem that naturally comes to mind is that of deducing the versioning relationship (one object is a newer version of another object) automatically. The solution to this problem, however, is entirely ML and therefore outside the scope of this thesis. Nonetheless, it seems likely that pattern-triggered services will be helpful in implementing that solution.

Versioning relationships can be represented in Haystack with a "previousVersion" assertion, which points (as the name implies) from an object to the previous version of that same object. If there are several different versions, then each newer version is connected to the previous one, resulting ultimately in a linked-list-like chain of versions.

4.4.1 Possible Features

It is not yet clear what should be done with this version information once it is known. Several possibilities (all of which can be implemented fairly easily in Haystack) are as follows:

- Automatically hide all versions of an object except the current one. This is easily done using a kernel service to detect and then hide every object that is pointed to by a "previousVersion" assertion.
- Automatically transfer (copy) other assertions about a previous version of the object to the new version of the object. Again, this is easily implemented using a service, although the question of which assertions to cope and which assertions not to copy must be addressed when determining the design requirements for such a feature.
- Allow the user to roll back to an older version of an object. This is easily implemented in one of two ways, depending on the desired behavior of the rollback. The first alternative is to install a new copy of the old version as if it were a brand new version; the advantage of this alternative is that it allows the user to later roll back again to the more recent version using exactly the same mechanism. The second alternative, which is much less messy and probably preferable, is to simply delete the version immediately after the one being rolled back to, and unhide the version being rolled back to. If the user decides (s)he has made a mistake, (s)he can still get back to the more recent version by undeleting it.

4.5 Spider

The basic goal of the spider component is to seek out interesting information from external sources (including websites and other Haystacks) and incorporate it into this Haystack for the user's benefit. This breaks down into three distinct tasks: determining what is interesting, searching for it, and importing it.

4.5.1 Interest

Determining what is interesting to the user is one of the basic machine learning goals of Haystack. As a machine learning problem, it is outside the scope of this thesis, but the generic discussion of deduction in section 4.1.2 is applicable to it.

4.5.2 Search

The strategy for searching external sources is also outside the scope of this thesis, but the capability of a Haystack to *be* searched by another Haystack is not. This ability is provided by the basic Haystack data interface, which (as discussed in section 4.1.1) is accessible from outside Haystack provided that the proper authorization from the operating system is in place.

4.5.3 Import

The most noteworthy task of the trio is importing interesting information into the Haystack once it is found. The problem lies in Haystack's requirement that all assertions be affirmed by some entity that believes them; when an assertion comes from external information, it is not usually believed directly by any entity that is part of Haystack (and therefore available to vouch for the assertion to satisfy the kernel).

Suppose in particular that the spider is trying to import data from another Haystack. We can't necessarily trust that any of the data in the other Haystack – even the "factual" assertions – are true. The fact that the other Haystack's kernel is willing to accept that some assertion X belongs in the other Haystack does not (and should not) imply that *this* Haystack's kernel will accept X. The spider itself is an entity that is capable of affirming assertions, but semantically it has no business affirming in our Haystack some assertion that, in the other Haystack, was made by (for example) a TitleGuesser service. All that the spider knows for certain is that, according to the other Haystack, the TitleGuesser service believes the assertion X.

As it turns out, this is enough to solve the problem. We will model the information for this example precisely as given in the sentence: "The spider believes that the other Haystack's kernel believes that the other Haystack's TitleGuesser service believes that X." The following procedure is used to import an affirmed assertion from another Haystack:

1. Directly import the data assertion X that has been determined to be of interest.
2. The remote Haystack contains the assertion Y: (X, affirmedBy, TitleGuesser). If this Haystack does not currently have the "TitleGuesser" entity, import it from the other Haystack. Then import the affirmedBy assertion.
3. In the remote Haystack, the affirmedBy assertion Y we've just imported is considered factual; it has a "KernelFact" assertion attached to it. In our Haystack, this does not imply that Y is factual; rather, it implies that Y is *affirmed by the kernel of the remote Haystack.* We represent this in the standard way, with the assertion Z: (Y, affirmedBy, RemoteKernel). RemoteKernel is an

entity defined in our Haystack to represent the kernel of the remote Haystack; if this entity doesn't already exist, it must be created before the assertion Z is made.

4. Finally, Z is the assertion that our Spider entity can vouch for. This results in one last affirmedBy assertion Q: (Z, affirmedBy, Spider). Q is factual, so our kernel adds on (Q, KernelFact, Kernel) in performing the affirm operation.

The entire result of this process, written out in nested form, is as follows: ((((X, affirmedBy, TitleGuesser), affirmedBy, RemoteKernel), affirmedBy, Spider), KernelFact, Kernel).

This procedure may seem very messy; in fact, it *is* messy. The reason for this, however, is that it accurately reflects the knowledge that our Haystack has (namely, the knowledge that our Spider affirms that the other Haystack believes that its TitleGuesser believes that X), and that the knowledge itself is messy. Even so, the assertion of interest (X, which might say something like "the title of document J is Foo") ultimately ends up in a position where it is accepted within our Haystack to be "probably true", and can therefore be seen by the user or used as a basis for further deductions by our services; this is precisely the desired effect.

4.6 User Interface

Designing a good user interface (UI) for a system like Haystack is an extremely hard task. This thesis is about systems design and architecture, not about UI design, and so the issue will be touched upon only briefly and then left for future researchers to tackle.

Throughout this document, we have described the set of features that we currently feel Haystack should have for manipulating data, and for each such feature we have at least outlined a possible implementation within the framework of the Haystack core. One of the primary requirements for a user interface is that it provide access to a given set of system features in the most intuitive possible way; the features the Haystack UI should provide access to are essentially those detailed in this document (also including, of course, any new features conceived of after the time of this writing).

As possibly its greatest challenge, the UI assumes the burden of presenting the contents of a Haystack to the user in some manageable form. Clearly the very simple put/get interface that can only examine one object at a time shows too little information, but attempting to display the entire graph structure all at once almost certainly shows too much information for the user to handle. The question of finding an effective compromise on this issue will be very important in the design of a Haystack UI.

In conclusion, the system architecture has provided the UI itself with a sensible interface to the data, and to the other features of Haystack. It will now be up to the user interface designer(s) to find the right way to bridge the gap between these interfaces and the user.

Chapter 5: Conclusion

This final chapter discusses the many opportunities available for related future work, and concludes the thesis.

5.1 Future Work

Haystack is a fairly large system, as we've seen over the course of the past several chapters. Because of the overall-design nature of this thesis, it turns out that almost any issue pertinent to Haystack is also pertinent to this thesis, and almost any future work that is relevant to Haystack is relevant to this thesis. This explains the relatively large amount of future work that will be outlined in this section.

5.1.1 Implementation

This thesis is entirely a design thesis, and so the most obvious opportunities for future work lie in the implementation of the design it has provided. The system has been broken down into modules with clearly defined purposes so that each module can be implemented separately from others. Now, each individual module must be transformed into concrete method specifications and finally into actual working software.

5.1.1.1 Black-box Internal Design

A few of the core modules delineated by this design are large enough, or not yet well enough understood (because of issues that this thesis did not have time to resolve), that they will need some additional internal design work before implementation can begin. They are currently understood only as "black boxes." A black box in design terminology represents a module that performs some operation or implements some feature according to a specified interface, but whose contents (i.e. the strategies used to solve the problem) cannot be seen from outside. Design theory teaches us that it is good to construct modules that view each other as black boxes (this is a good measure of modularity), but the contents of each box must still be specified in order for the box to be built.

The first such black box in this Haystack design whose internals are not yet fully specified is the RDFstore, described in section 3.1. The RDFstore is a large module that must implement a substantial set of features in order to ensure transaction-safety. This document has described one possible breakdown of the RDFstore into three separate layers; if this hierarchy is accepted by the RDFstore's designer, then the lowest level (the ObjectStore) will still require

some additional thought. One promising approach is to adapt an off-the-shelf conventional database (which already provides transaction-safety) to fit the interface we require of the RDFstore or ObjectStore, but the designer may also with to explore other possibilities.

The second black box whose contents are unclear is the Registrar (described in section 3.2.1). The feature of service registration is fairly clearly defined by this document, but is it not yet at all obvious how the registrar will examine each incoming write operation to the Haystack and determine what, if any, pattern matches it will result in (in order to trigger the appropriate services). This problem will of course depend greatly on the modeling problem of determining a pattern language (which is also unresolved, and described later in this chapter).

Finally, the third module from the core system that still needs substantial work before it can be implemented is the query execution system, discussed in section 3.3.3. Again, some general guidelines for sub-architecture are presented in this document, but additional work will be needed in order to finalize the details of the mechanism. This problem will depend on the determination of a query language (an issue related to the pattern language), which is also mentioned later in this chapter.

5.1.2 User Interface

The design and implementation of a good user interface for Haystack is a substantial future project. Some ideas are discussed in section 4.6 of this thesis, but to all intents and purposes the problem remains completely unsolved at the moment.

5.1.3 Knowledge Representation

This thesis leaves a few questions unanswered in the area of knowledge representation and data modeling. Some are more complicated than others, but answers to all of them will be very helpful in continuing the Haystack effort.

5.1.3.1 Intangible URI's

Intangible resources, described in section 2.1.2, do not currently have a clear form for the kinds of URI's that should be used to refer to them. This thesis argues that, in the general case, the only inherent property of an arbitrary intangible object is uniqueness. Based on this, it proposes a possible URI form consisting of the prefix "unique://" followed by a unique string. The problem with this is that it is often very easy to end up with multiple instances of what should be the same object, because it is difficult to tell whether the one we want to refer to with a new assertion already exists in the Haystack or not.

The problem is compounded when we consider importing such objects from other Haystacks. In general, the "unique://" URI's in another Haystack may overlap with the ones in our Haystack; this means that when we import a new intangible object from a remote Haystack, we must not simply use the URI it has in that other Haystack, but rather create a brand new "unique://" URI for it in this Haystack. However, if we then go back to the remote Haystack later on and try to import an assertion about that object, we have no way to tell that it is about the same object we had imported previously (this is a general difficulty with intangibles). Of course, we could work around this issue by keeping around an association table of imported intangible resources from other Haystacks, but in addition to being messy, this solution cannot cope with the possibility that a remote Haystack uses some URI to refer to one intangible, and then later on winds up using that same URI to refer to a different intangible (this scenario is not common, but it is possible).

It would be nice to have a more clear-cut way of identifying intangibles to avoid some of these problems. One plausible approach that suggests itself is to define identity rules for certain types of intangible objects. For example, we might say that two locations are the same if they have the same geographical coordinates (assuming that locations are considered intangible and coordinates are literal); this would allow us to adopt a "canonical form" for the URI of a location that was some encoding of its coordinates (perhaps "intangible://Location/xxxxx,yyyyy"). This type of scheme would be particularly useful for naming Entities, because it is practically a guarantee that whenever we import anything from a remote Haystack, we will also have to import at least one entity (see the discussion of importing Haystack data in section 4.5.3). Alternatively, if we continue using "unique://" URI's, this type of identity rule would allow us to create a service whose job it is to find duplicate locations (those with the same coordinates) and "merge" them into a single location.

Clearly, this can only be done for specific classes of intangible objects, one at a time; by definition there is no way to digitally represent the identity of an *arbitrary* intangible object. Furthermore, this project can only proceed so far without running afoul of the philosophical question of identity, as illustrated by the proverbial question of whether it is ever possible to step into the same river twice. Nonetheless, it is an area in which some future work would be quite valuable to Haystack.

5.1.3.2 Reasons

The model proposed in section 4.2.1 of this thesis to explain an entity's reasons for affirming an assertion is currently inadequate. Most obviously, it does not deal with causes

outside of the Haystack (including almost all affirmations made by the user entity), but it has other shortcomings as well. If Data Dependency is to be implemented as a module of Haystack, this model must be rethought much more painstakingly.

5.1.3.3 Pattern Language

As discussed in section 3.3.2.2, this thesis does not even attempt to delve into the issues associated with defining a pattern language for semi-structured data. This is a hard problem that needs to be addressed specifically before the majority of Haystack's core (specifically, anything involving services or queries) can be made to work.

5.1.3.4 Query Language

In addition to the pattern language mentioned above, Haystack needs a full query language before any of the query functionality can be implemented. The pattern language itself is the biggest hurdle (the semi-structured-search parameter of the query will be a pattern), but the remaining types of query parameters need to be enumerated (and assigned names, so that the query formulator and the query executor can agree on what assertions should be examined to parse the parameters of the search). Obviously one of the biggest assets of the query's representation in Haystack is that it is easily extensible to include more parameters; thus, this first enumeration of parameters will be only the initial set and not an exhaustive list.

5.1.4 Machine Learning

There are two categories of problems to be solved by future work in the area of machine learning. The first is a set of problems that have already been posed in Haystack, and are mentioned elsewhere in this document. The second category involves brainstorming new features to add to Haystack.

5.1.4.1 Existing Problems

Throughout this document, we have mentioned a variety of different features that we would like to see in Haystack. A substantial number of these involve machine learning in some way, and because this thesis is not an artificial intelligence thesis, all such ML components have been left as future work. The following is a list of interesting ML problems mentioned in this document (with section references):

- Query score combination (3.3.1): adjust the way in which numerical scores for different search types are combined into an overall search type, based on observations of which results the user preferred.
- Clustering (4.1.3): organize data into intuitive categorical groupings.
- Reason tracking for Data Dependency (4.2.1): automatically deduce the reason(s) for an entity's affirmation of an assertion.
- Version relationships for Content Versioning (4.4): automatically deduce that one object is a newer version of another.

5.1.4.2 New Features

The primary goal of this thesis was to provide Haystack with a solid design foundation that will be able to easily and conveniently support future Haystack research, especially in the areas of machine learning and knowledge representation. Therefore, another key area of future work related to this thesis is to brainstorm and implement new ways to further the goals of Haystack using machine learning that we have not thought of yet. The goals of Haystack are summarized in section 1.3, and the applicability of ML to those goals revolves around the general problem of deducing what kinds of things the user is interested in, for the purpose of determining how interesting a new piece of information will be to the user, and also the problem of generating more information from existing information.

5.1.5 Generic Filtering

The idea of the MetaFilter, described in section 3.2.3, was rejected largely on the basis of not realistically being able to build it conveniently at this point in time. However, it has many very nice properties from a design standpoint and would be an enormous asset to the extensibility of Haystack. Once some of the other related issues (e.g. the pattern language) have been resolved, it would be worthwhile to take another look at the MetaFilter and reconsider including it in the core of Haystack.

Section 3.2.3 also describes the possibility of creating database-like "views" in Haystack; a filtering problem very similar to the MetaFilter problem but applied on a voluntary basis by clients of the data interface rather than by default by the kernel. It is likely that a good solution to one of these problems would be easily adaptable into a good solution to the other problem, and so it would be productive to consider both problems as potential future features for Haystack.

5.1.6 Security and Access Control

This iteration of Haystack was designed essentially without security. However, any system to which we eventually expect users to entrust their data must be able to protect and safeguard that data. This section briefly discusses the basic security concerns of Haystack.

The ability to access the Haystack is granted to entities. Certain entities (i.e. the kernel and user) have greater privileges than other entities, and outside entities (those that have not been "installed" by the user) have only guest-level access. The definition of guest-level access is one unresolved question. The problem of entity authentication (and consequent enforcement of the access control rules discussed in section 2.4) is another. Currently, we simply trust entities not to masquerade as other entities; this strategy is good enough to build and test a system, but not good enough to inspire people to use it.

There are two extensions of Haystack's access control situation that must be kept in mind when designing security safeguards for it. The first is that not all entities needing to access the Haystack data will be inside Haystack. Some may take the form of other applications running on the same machine, and others may even be running on different machines, across the network. The second is that Haystack may eventually want to support multiple users, as described below.

5.1.7 Multiple Users

Enabling Haystack to support multiple users is certainly not a high priority in the development of the system, but it may eventually be useful. If so, the current existence of entities should be helpful in making the transition. The key questions in creating a multi-user Haystack will be the resolution of disputes between users (e.g. one user wants a particular service installed in the Haystack, and the other user doesn't), and the interaction of one user with the other user's data. It is worth noting here that if the answers we determine to these questions point in the direction of having one dataset for each user that is not modifiable by other users, then a multi-user Haystack would be a waste of our time to implement because it would simply be a collection of virtual single-user Haystacks. In that case, the energy that would have been spent on making Haystack multi-user should instead be spent on finding new ways for single-user Haystacks to collaborate.

5.2 Conclusion

The objectives of this thesis were threefold: first, to provide Haystack with a coherent, elegant, modular, and extensible overall design that would serve it well in the future; second, to

document as much as possible the thought processes that have gone into this design so that future researchers could avoid spending more time on the same issues; and third, to serve as a general teaching document to introduce new members of the Haystack research team to the basic concepts and terminology of Haystack.

In the course of this document, we have broken Haystack down into a data model based largely on RDF but with its own special enhancements, a persistent transactional storage system for generic RDF data, a kernel containing the machinery to invoke and execute services, and a query processing system. We have explained the purpose and functionality of each module, and documented our design reasoning in all applicable ways. We have argued that the major classes of features currently under consideration for Haystack can be implemented effectively and conveniently within the framework of this core architecture. We have endeavored to explain all of the important concepts and ideas mentioned (not just about Haystack, but about good design in general) clearly enough for them to be understood by newcomers to the Haystack system. Finally, we have discussed the opportunities for future research that are available in connection with Haystack, particularly in the areas of implementation, machine learning, and knowledge representation.

We are confident that the contributions of this thesis to the Haystack project will strengthen and guide the process of its imminent rebuilding, as well as providing a foundation for a wide variety of valuable future research.

Appendix A: Interfaces

This appendix specifies the various data interfaces defined at different levels of Haystack.

A.1 RDFstore Interface

The following methods are provided by the RDFstore (discussed in section 3.1), and used only by the kernel. Recall that the RDFstore is designed to store generic RDF data, not specifically Haystack data, and so the customizations of the Haystack data model (e.g. delete) are not found in this interface.

`Integer newTransaction()`
Creates a new transaction, and returns a Transaction Identifier (TID) that can be used later to refer to this transaction. This TID is given as the first parameter of all read and write operations that are to be part of the transaction.

`Void commitTransaction(Integer theTID)`
Commits the transaction with TID <theTID>.

`Void abortTransaction(Integer theTID)`
Aborts the transaction with TID <theTID>, rolling back all associated changes to the data.

`Void putObject(Integer theTID, String theURI, BitString theData)`
Creates an object in the RDFstore with object data <theData> (a sequence of bits) and URI <theURI>.

`BitString getObject(Integer theTID, String theURI)`
Returns the serialized object data of the RDFstore object with URI <theURI>.

`Void putAssertion(Integer theTID, String assertionURI, String subjectURI, String predicateURI, String objectURI)`
Creates an assertion in the RDFstore with subject <subjectURI>, predicate <predicateURI>, and object <objectURI>. The URI of the assertion itself is <assertionURI>.

`String getAssertionSubject(Integer theTID, String assertionURI)`
Returns the URI of the *subject* of the assertion with URI <assertionURI>.

`String getAssertionPredicate(Integer theTID, String assertionURI)`
Returns the URI of the *predicate* of the assertion with URI <assertionURI>.

`String getAssertionObject(Integer theTID, String assertionURI)`
Returns the URI of the *object* of the assertion with URI <assertionURI>.

`Enumeration getAllAssertionsWithSubject(Integer theTID, String subjectURI)`
Returns a generator for the URIs of all assertions whose *subject* is <subjectURI>.

`Enumeration getAllAssertionsWithPredicate(Integer theTID, String predicateURI)`
Returns a generator for the URIs of all assertions whose *predicate* is <predicateURI>.

Enumeration getAllAssertionsWithObject(Integer theTID, String objectURI)

Returns a generator for the URIs of all assertions whose *object* is <objectURI>.

Void expunge(Integer theTID, String theURI)

Expunges (irreversibly removes) the object or assertion with URI <theURI> from the RDFstore.

A.2 Basic Haystack Data Interface

The following methods are provided by the Kernel to the rest of Haystack, and are referred to as the Basic Haystack Data Interface. This interface reflects the specific behaviors defined for the Haystack data model in Chapter 2.

Void newTransaction(String entityURI)

Creates a new transaction for the entity with URI <entityURI>. The transaction is associated with the current Java thread to eliminate the need for explicit TID's. There can be at most one transaction associated with each thread at a time; if newTransaction is called while another transaction is still in progress, the old transaction will be automatically aborted before beginning a new one.

Void commitTransaction()

Commits the transaction currently associated with this Java thread.

Void abortTransaction()

Aborts the transaction currently associated with this Java thread.

String putLiteral(BitString theData)

Creates a Literal in Haystack with object data <theData> (a sequence of bits). Returns the URI of the Literal, which consists of "md5://" followed by an MD5 cryptographic hash of <theData>.

BitString getLiteral(String theURI)

Returns the serialized object data of the Literal with URI <theURI>.

String putAssertion(String subjectURI, String predicateURI, String objectURI)

Creates an assertion in Haystack with subject <subjectURI>, predicate <predicateURI>, and object <objectURI>. Returns the MD5-based URI of the assertion. Note that this operation does *not* automatically affirm the assertion. Also note that unless the entity associated with the current transaction is the Kernel, predicateURI may not point to the literal "KernelFact".

Void affirmAssertion(String assertionURI)

Affirms the assertion with URI <assertionURI> on behalf of the entity associated with the current transaction.

Void rejectAssertion(String assertionURI)

Rejects the assertion with URI <assertionURI> on behalf of the entity associated with the current transaction.

String getAssertionSubject(String assertionURI)

Returns the URI of the *subject* of the assertion with URI <assertionURI>.

String getAssertionPredicate(String assertionURI)

Returns the URI of the *predicate* of the assertion with URI <assertionURI>.

String getAssertionObject(String assertionURI)
Returns the URI of the *object* of the assertion with URI <assertionURI>.

Enumeration getAllAssertionsWithSubject(String subjectURI)
Returns a generator for the URIs of all assertions whose *subject* is <subjectURI>.

Enumeration getAllAssertionsWithPredicate(String predicateURI)
Returns a generator for the URIs of all assertions whose *predicate* is <predicateURI>.

Enumeration getAllAssertionsWithObject(String objectURI)
Returns a generator for the URIs of all assertions whose *object* is <objectURI>.

Void expunge(String theURI)
Expunges the object or assertion with URI <theURI> from the Haystack. Valid only if the entity associated with the current transaction is Kernel or User.

Void delete(String theURI)
Deletes the object or assertion with URI <theURI> from the Haystack. Valid only if the entity associated with the current transaction is Kernel or User.

Void hide(String theURI)
Hides the object or assertion in the Haystack with URI <theURI>. Valid only if the entity associated with the current transaction is Kernel or User.

A.3 Undeletion Interface

The Undeletion Interface (first mentioned in section 2.3.2.2) provides the same methods as the Basic Haystack Data Interface specified above, except that the "get" operations in this interface are able to see deleted items (whereas in the Basic Haystack Data Interface any deleted item appears to be completely nonexistent, in this interface it appears as a normal item with a deletedBy assertion attached to it). This interface also provides the undelete operation:

Void undelete(String theURI)
Undeletes the deleted object or assertion with URI <theURI>, restoring it to the Haystack.

A.4 Crystal Ball

The Crystal Ball (discussed in section 4.3.2) is part of the Time Versioning system. It provides the same methods as the Basic Haystack Data Interface specified above, except that the "get" operations in this interface are able to see only the items that existed in the Haystack at the time to which the crystal ball is "attuned." Note that any operations that *write* to the Haystack still happen in the present time; the past cannot be changed. The crystal ball also provides the following additional operation:

Void setViewingTime(String timeURI)
Attunes this Crystal Ball interface to the time represented in Haystack at <timeURI> (which should refer to a Literal).

Appendix B: References

[1] Shnitser, Svetlana. *Integrating Structural Search Capabilities Into Project Haystack.* Master's Thesis, Massachusetts Institute of Technology, Department Of Electrical Engineering and Computer Science, May 2000. http://haystack.lcs.mit.edu/papers/.

[2] *Resource Description Framework (RDF) Model and Syntax Specification.* W3C Recommendation 22 February 1999. http://www.w3.org/TR/REC-rdf-syntax/.

[3] *Lore* website. http://www-db.stanford.edu/lore/. 2001.

[4] Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. *The Lorel Query Language for Semistructured Data.* International Journal on Digital Libraries, 1(1):68-88, April 1997. http://www-db.stanford.edu/lore/pubs/.

[5] *Haystack* website. http://haystack.lcs.mit.edu/. 2001.

[6] Adar, E., Karger, D.R., and Stein, L. *Haystack: Per-User Information Environments.* Conference on Information and Knowledge Management, 1999. http://haystack.lcs.mit.edu/papers/.

[7] Adar, Eytan. *Hybrid-Search and Storage of Semi-structured Information.* Master's Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1998. http://haystack.lcs.mit.edu/papers/.

[8] Abiteboul, S. *Querying Semi-Structured Data.* International Conference on Database Theory, 1997, 1996. http://dbpubs.stanford.edu/pub/1996-19/.

Brown's Wood Architects

Morton Braun (with Walter Hill)

Braun House, 19 Moccasin Hill, 1960

Arthur H. Brooks, Jr.

Freeman House, 5 Laurel Drive, 1956

Freud-Loewenstein House, 34 Laurel Drive, 1956

Healy House, 15 Moccasin Hill, 1956

Compton and Pierce

Allen House, 38 Laurel Drive, 1958

Grover House, 14 Moccasin Hill, 1956

Harris House, 1 Laurel Drive (37 Conant Road), 1957

Hill House, 37 Laurel Drive, 1957

Swanson House, 26 Laurel Drive, 1956

Earl Flansburgh

Stevens House, 11 Laurel Drive, 1964

Ronald Gourley

Eckhardt House, 27 Laurel Drive, 1956

Ann and Ranny Gras

Gras House, 40 Laurel Drive, 1956

Hoover and Hill

Rawson House, 8 Moccasin Hill, 1960

Wales House, 18 Moccasin Hill, 1959

Carl Koch and Associates

Ritson House, 1 Moccasin Hill, 1956

Krokyn and Associates

Morgan House, 30 Laurel Drive, 1956

Stanley Myers

Novak House, 12 Laurel Drive, 1957

Smulowicz House, 7 Moccasin Hill, 1957

Norman I. Paterson

Balser House, 41 Laurel Drive, 1960

Nyna Polumbaum

Polumbaum House, 31 Laurel Drive, 1956

Richard Reese

Shansky House, 11 Moccasin Hill, 1975

Harvey Spigel

Kramer House, 8 Laurel Drive, 1957