# A Model for Indexing and Resource Discovery in the Information Mesh

by

Lewis D. Girod

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

October 1995



Author .....
Department of Electrical Engineering and Computer Science
October 31, 1995

Certified by .....
Karen R. Sollins
Research Scientist
Thesis Supervisor

Accepted by .....
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

**A Model for Indexing and Resource Discovery in the Information Mesh**

by

Lewis D. Girod

Submitted to the
Department of Electrical Engineering and Computer Science

October 31, 1995

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

## Abstract

In this thesis, I designed and implemented a unified resource discovery system which is accessible to a wide range of potential users. Users who have neither experience nor expertise with existing information services, and may not even have direct network access to these services, can gain the benefits of a world-wide information infrastructure by using this system. Ideas from the developing MESH system have been used to create a framework for a distributed base of knowledge about network services. This knowledge-base is implemented by a network of "Referral servers" which interpret questions and suggest good places to look for answers, and "Index servers" which provide a more standardized interface to existing Internet services. In the Internet, heterogeneity seems inescapable, so the Query language supports a variety of popular query languages (however not all Referral servers will parse all languages). There are benefits of heterogeneity as well, such as increased fault-tolerance and, in the case of query languages, increased expressiveness. A third type of server, the "Sponsor", provides automated management of a search as it progresses, requiring of the user neither a high-powered network connection nor excessive amounts of expertise.

Thesis Supervisor: Karen R. Sollins
Title: Research Scientist

# Acknowledgments

There are a great many people without whose help this thesis would not exist. The initial choice of direction I owe primarily to Brian LaMacchia and to the people of Switzerland, under whose expert tutelage I determined to choose a difficult yet highly interesting problem and to learn a great deal from the experience.

To my advisor, Karen Sollins, I owe the bulk of my thanks, for helping in every way possible: discussing and developing ideas, narrowing topics and solidifying concepts, reading the endless revisions, dealing with all of the bureaucratic hassles involved with the generally incomprehensible guidelines, helpful chats about issues unrelated to this work, and a zillion others too numerous to mention.

Everyone else working on the MESH project (Bien Veléz-Rivera, Tim Chien, Matthew Condell, and Jeff VanDyke) deserve much credit as well, since disscussions of the project at group meetings and in the office always unearthed new and useful viewpoints. The other members of the ANA group likewise provided invaluable support: Dave Clark, John Wroclawski, Garett Wollmann, Tim Shepard, and above all Lisa, whose telephonic talents worked the usual miracles.

The provision of moral support and various threats on my life was undertaken by the following motley crew: my parents, who strongly encouraged me to finish, or at least die *trying*, my former housemates, notably Kathy, Frank, Aditya, and Stacey, who prodded and cajoled me to convert this thesis from a dream into an actual document, and my current housemates Jonathan and Matt, who tried unsuccessfully to get me to remodel our kitchen instead.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Context and Motivation

The initial motivation for this work stemmed from the difficulties of finding the information you want in the World Wide Web and the related Internet infrastructure. Generally, data is discovered either by accident or by a long, frustrating search. After playing around for a while one can become fairly expert at knowing where to look, and a multitude of search tools exist to make this easier. But for the novice user, resource discovery is very difficult if it is possible at all.

From the perspective of an end user the Web appears to be a powerful and user-friendly infrastructure with almost limitless potential for growth, poised to lead the way to a brighter and more informative future.[1] However, anyone who has looked at the structure of the Web and the suite of protocols on which it is based knows that this perspective is naive at best. Given that the Web is expected to grow vastly in size and in clientele, it seems a very bad sign that right now things already tend to get clogged up. Consider the hapless user who attempts to use Lycos to find a Web page that he remembers having seen before. The first obstacle this user will encounter is that Lycos has been in use to 100% capacity continuously ever since it was awarded a prize for being a useful service. This means that even though there are a number of

[1] For more information about this version of the future, see the latest series of AT&T commercials shown on network television.

very fast machines providing the service, they are hopelessly overloaded and getting a connection is unlikely. Of course, even if the user gets a connection, he still may have difficulty locating the page. Lycos uses standard keyword search techniques to resolve queries, so the user will need to remember with some degree of accuracy some words from the page in order to find it. Of course, having seen the page before this may not be so hard – finding things with keyword search is much harder if you don't know exactly what you are looking for.

At the present time, a surprisingly small amount of information is actually stored on the Web. The implementors of Lycos estimate that the total amount of data accessible via the Web is only about 30 Gigabytes, distributed among about 4 million documents, of which one million are Web pages. [23] This collection of data is still small enough to be indexed in one monolithic chunk, which is fortunate for services such as Lycos which essentially create one giant index. As the commercial world begins to discover this vast new marketplace, and as more people begin to get involved with it and start to publish their own data, the Web has the potential to increase dramatically, making monolithic indices unworkable for a number of reasons. Not only will the volume of information be much too large to handle in one chunk, but the number of users needing to search that information will cause any single indexing site to immediately collapse under the load. The construction of mirror sites is expensive and will not serve well to stave off the influx of new customers.

The technique of using mirror sites is also highly wasteful. In any given archive of general information, the hit rates vary greatly from one piece of information to the next. Specifically, most of the information in an archive with wide enough coverage will be hit very infrequently if at all, while some relatively small subset will account for most of the total hits. This means that when a site is mirrored, most of the space on the new server is devoted to information that will rarely be used but still slows down the search process.

Worse yet, given current searching and query technology, a monolithic index cannot function if the amount of data in the index is very large. At the moment, the techniques available for taking a query and searching a large database of heteroge-

neous data are unfortunately still very rudimentary. The sort of Natural Language processing that has always been dreamed of as the way to analyze queries is still a dream, *except when the database being searched covers a very narrow domain.* Technology for implementing "expert systems" has been made to work well only when the domain of expertise is narrow enough that the semantics of the vocabulary can be adequately expressed. Without the constraint of a narrow domain, the remaining technologies currently available for query resolution are keyword search and statistical techniques derived from keyword search.

While these methods have the benefit of being fairly simple conceptually, they have the drawback of focussing entirely on simplistic syntactic analysis and of therefore being only incidentally affected by context and semantics. Keyword search is the simplest and most obvious technique for matching a query to a document. No one would argue that the technique is very effective, but on the other hand no one has come up with a better solution which is still simple (i.e. not dependent on huge amounts of data). Even on small data sets keyword search tends to locate spurious documents, but usually also locates the document that was required. As the size of the database grows, the number of spurious documents also grows, since it becomes statistically more likely that a spurious match will be found.

Recently a number of statistical approaches have been introduced which make analyses of the occurrences of words in sets of documents.[24] These techniques are essentially more powerful versions of the same keyword search idea, in which the statistical analysis reveals syntactic similarities between documents which hopefully correspond to similarities in topic. This has been shown to work to a degree in small experiments, but it is a more effective tool for document clustering[25][11] (that is, forming clusters of related documents) than it is for resolving queries, since queries tend to be terse and pithy. It is likely that the degree of error involved with these statistical techniques will become a problem to an increasing degree as the size of the database grows.

For these reasons we suspect that a monolithic index cannot scale with our current technology. A monolithic index has essentially an unbounded domain, and the

available technologies for operating in such a domain become more dubious as the size of the database grows. There is a certain intuition which supports these suspicions; namely, that the problems related to the navigation of a complex world-wide information infrastructure can only be attacked with a likewise complex and similarly organized solution.

## 1.2 The meshFind Proposal

To solve this problem better we need to construct distributed indexing systems. This helps our situation in many ways. If we build smaller indices that cover more limited domains, we have a much better chance of being able to take advantage of the semantics of the query. A distributed collection of smaller indices can also mean more efficient ways of counteracting loading problems, since the servers which are hit most often can be assigned additional resources independently of the other servers. Indices of limited domain also provide a good way to modularize the problem of searching, since each index in the system can have its own methods of query resolution. This means that the system as a whole is highly complex and rich with specialized techniques, but is neatly broken down into manageable parts.

Once a large number of small indices are constructed, the problem is still far from being solved. The problem now becomes one of finding the right server to answer a given query. Perhaps one way to attack this problem is to implement layers of meta-indices above the base indices. These meta-indices would take a query and try to determine which of the indices on the next level down could best answer it. It is possible that these meta-indices could be mechanically generated. For example, if each base index produces an abstract of some kind, perhaps these abstracts might be processed using a document clustering technique, and that analysis might be used to form a layer of meta-indices.

Given these analyses, it is clear that the information in the net must be reorganized and made accessible to the public in an efficient manner. The reorganization can be partially done by mechanical means, but much of the work will need to be done

by humans because at the present time only they understand the content of the documents. This thesis presents an architecture for a network of servers which loosely follows the ideas described in the previous two paragraphs. A network of "Referral servers" and "Index servers" make up a framework for the construction of a distributed index system. This system can be organized in the way that is described above, by creating smart Referral and Index servers which cover limited domains. Then layers of Referral servers may be constructed above the bottom layers, perhaps using mechanical means to do so. A layer of "Sponsors" sits above the Referral network, providing a gateway or proxy service to clients who do not have the resources to own a high-bandwidth network connection.

There are a great many problems to be solved in this domain; other researchers are currently working on parts of it, notably Mike Schwartz of the Harvest project.[5] Harvest is designed to produce a distributed network of archives and caches which can store and organize data in a distributed fashion, but there is less emphasis placed on the process of searching and of finding the right archive.

## 1.3 Formatting Conventions

In this document, we use a number of formatting conventions. As a rule, whenever we identify a part of a system using an actual Scheme identifier from our prototype, the `typewriter` typeface is used. When we discuss a structure from a more abstract perspective, we use the normal Roman typeface, so that the discussion will flow more smoothly. For example, in Chapter 4, we will discuss the structure of an Address object. In our implementation, the name of the Address structure is abbreviated to `addr`. For aesthetic reasons the Address structure is only written as `addr` when the discussion needs to be very specific.

Throughout this document, tables are used to describe data structures and procedures. When we are describing a data structure, we provide a table which lists the fields of the structure and identifies the type of data which is expected to fill each field. For example, Table 1.1 shows the format of a structure of type `example`. (This

structure might underlie an Example object.)

| Structure `example` | |
|---|---|
| `field-1` | number |
| `field-2` | string |
| `field-3` | boolean |

Table 1.1: The Example structure.

The first line of a table describing a data structure gives the name of the structure, and the following lines detail the fields. The left column gives the Scheme symbol associated with that field in our prototype system, and the right column gives the type that is expected in that field.

When the implementation of procedures is discussed, sets of related procedures are described in a tabular format. These tables give the name of each procedure, the names and types of the arguments that procedure expects, and the type of that procedure's return value. An example of one of these tables is shown in Table 1.2.

| `example/...` | Arguments expected; return type |
|---|---|
| `create` | `f1`:number, `f2`:string, `f3`:boolean; Example |
| `f1` | `self`:Example; number |
| `set-f1!` | `self`:Example, `val`:number; – |

Table 1.2: Procedures implemented in the Example module.

In general, procedure names in our implementation follow the convention of appending the name of the operative data structure and a slash to the beginning of the procedure name. That is, procedures operating on an Example structure would all have names beginning with "`example/`". In order to keep the tables smaller and easier to read, these prefixes are factored out and placed at the head of the table. In Table 1.2, the left column of the top line reads "`example/...`", signifying that the names listed on the following lines are all prefixed by "`example/`". Each procedure's arguments and return value are also specified. The actual argument identifiers from the implementation are given in `typewriter` face, and the types are given in Ro-

man face except in cases where the exact Scheme identifier is necessary for clarity. The arguments are specified in the format `identifier`:Type. The arguments form a comma-separated list, terminated with a semicolon. After the semicolon the type of the return value is given. In the event that a procedure does not have a specified return value, a dash is substituted for the type.

## 1.4 Roadmap

The following list summarizes the topics covered by each of the chapters:

**Chapter 2** describes works related to this one, and also explains some more of the motivation for the work.

**Chapter 3** gives a good first-order look at the design and structure of the meshFind system. This chapter isn't very detailed, and covers the entire project.

**Chapter 4** talks about the data structures and messages used in the implementation of the meshFind protocol, and along the way highlights some of the design choices that helped formulate the protocol. This chapter also presents the structure of Filters and the implementation of the Filter interpreter.

**Chapter 5** describes the design and implementation of the Sponsor server. This chapter also explains the interface between a Filter and the Sponsor; that is, it tells how Filters are written, how they are used, what parameters they take and what results they may return.

**Chapter 6** describes the design and implementation of Referral and Index servers. Two further related topics are covered: first, the tools used to connect meshFind to the rest of the world, and second, the overall topology of the Referral server network.

**Chapter 7** covers the implementation issues involved with writing Client software for meshFind. It proposes two separate implementation strategies, one which is

text-based and might run on very inexpensive equipment, and one which uses the World Wide Web.

**Chapter 8** concludes this work with many ideas for continuing and redesigning the project.

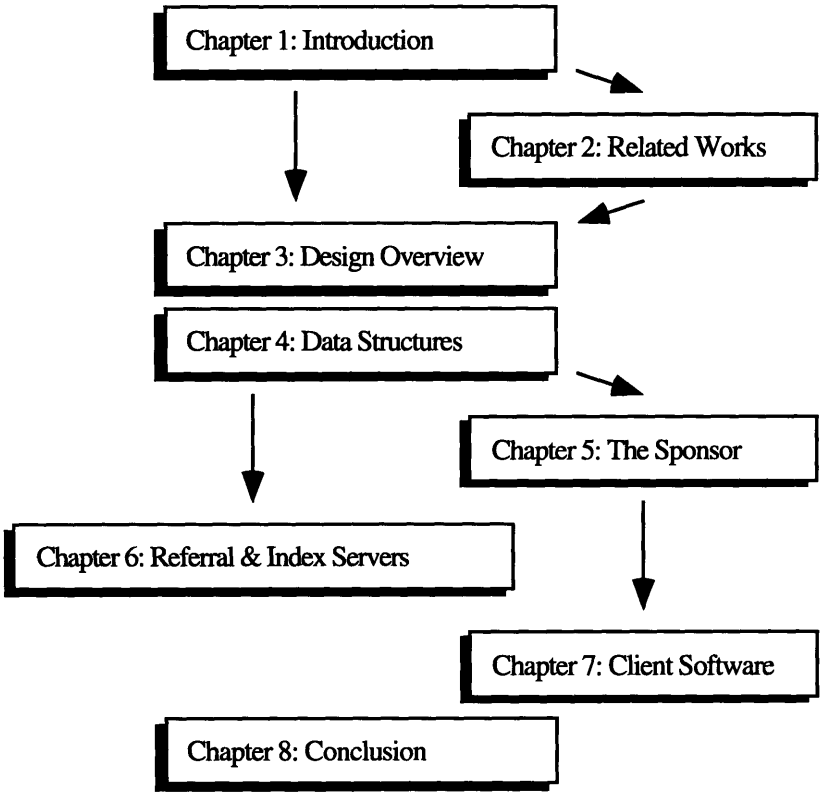


Figure 1-1: Roadmap for this document.

In general, this document was written so that it can be read from start to finish, but certain deviations from this order should not pose a problem (see Figure 1-1 for a dependency graph). Topics are therefore covered in earlier chapters which are not explicitly defined in later chapters. The core of the work appears in Chapters 4 through 7; before reading any of these chapters it is advised that Chapter 3 be read, so that the overall structure of the system may be understood. Chapters 5 through 7 all rely on the parts of the protocol which are explained in Chapter 4. Since the

design of the Sponsor is highly related to that of the Client software, Chapter 5 is a partial prerequisite for Chapter 7, especially the parts which describe Filters and the servicing of Clients. In general, in order to get the best understanding of how the protocol operates, Chapter 5 is the best choice, since it forms a gateway between the Referral server network and the Client. Chapters 2 and 8 may be read independently of the rest of the document. They contain more of the vision of where the project is directed, while the middle chapters cover the details of the implementation.

# Chapter 2

# Related Work

Recently there has been a significant interest in the general area of Resource Discovery, mainly due to the impending problems laid out in the previous chapter. But the overall problem is far from new, and people have been trying to tackle it ever since the earliest collections of written works grew large enough to present problems of organization. Thus a long heredity of librarians stretches up to the present day, but the tools that have been developed are far from adequate to tackle the problems of the near future.

## 2.1 Internet "Fish"

Perhaps the project most intimately related to this one is the Internet Fish project being developed by Brian LaMacchia of the MIT AI Laboratory.[19][18] He is presently working on a PhD thesis which attacks the same essential problem as this work, although from what is in some sense the opposite direction. In his system, the user builds up a personalized search tool on his local machine. This is accomplished via a continuing dialogue between the user and the Fish system. Over the course of this dialogue the Fish system adapts to the user's needs and gains an understanding of the sort of information that is interesting to the user. Eventually, a successful implementation of the Fish system would actually begin bringing tidbits of interesting information to the user without being explicitly asked, as well as being generally able

to resolve successfully the user's queries.

The system works by collecting chunks of data and by transforming the chunks using a collection of rules. The user directs the search by indicating which chunks of data look most interesting, and the Fish then explore the avenues of the search that seem most promising. The rules that have been developed often do some amount of parsing on a data chunk, and based on the results of the parse continue searching in new directions. For example, one rule might take an HTML page and pull out the link structures, while another rule might take link structures and pull out the URLs, while a third rule might follow the URLs and bring back the pages to which they refer. The network of data chunks is kept organized as a graph in which the edges represent the invocation of rules. This way the dependencies are kept around so that if underlying information changes or is invalidated the information that relies on it may be located.

One of the central parts of the Fish system is the "interestingness" metric. This is a difficult problem to solve, and represents a major part of the work which is presently unfinished. The hope is that such a metric could be constructed after a long-standing conversation has been maintained between the user and the system.[18]

The defects we see with this approach mainly surround the fact that this system does not build a shared infrastructure. All the intuition about the network is stored locally and only has meaning locally. One of the founding goals of our project is the construction of a shared infrastructure from which many people can build and benefit. Furthermore, the "Fish" system requires a high-bandwidth network connection which essentially runs continuously, which may be beyond the reach of many people and which cannot be offered cheaply in the near future. Another founding principle of our system is the minimal network connectivity needed to serve a given person, as well as the possibility of walking in off the street to a library and using one of the terminals, both of which are clearly contrary to the design of the "Fish" project.

## 2.2 The Content Router

Another group working on a project very similar to ours is the Programming Systems Research Group (PSRG) of MIT's Laboratory for Computer Science, headed by Prof. David Gifford. They are constructing a system which attacks essentially the same problem as ours, from a very similar direction, but they differ slightly in their approach and philosophy. The PSRG is developing an architecture in which a hierarchy of "content routers" helps a user cull responses from a collection of 504 WAIS servers. The user runs an interface program on a local machine, or uses shell commands, to interact with a content router. This interaction is carried out using the abstraction of the Semantic File System, or SFS, which makes a heterogeneous collection of information servers and content routers appear to be accessible file systems. Interaction with a content router amounts to sending a query to the router, specified in a standardized query language. The router then returns all items matching the query along with suggestions for possible query refinement. These suggestions are presently generated by listing other terms which are frequently collocated with items that have already been discovered. In the SFS paradigm, any results of a search are returned as virtual directories which are computed by the server on demand. Some of these result items may be actual documents which are directly accessible through an SFS-WAIS gateway. Other items may be pointers to other content routers which may be selected and searched, or "expanded".

A content router is intended to provide access to a collection of items which may be documents or other routers. Since the content routers are arranged in a hierarchy, they collectively provide access to a hierarchy of collections, in which the leaf nodes are actual documents. Associated with each collection and each document is a content label, which is a boolean combination of terms intended to describe the scope of the collection or the topic of the document. Since the implementation uses WAIS servers to store and index the documents, for each WAIS server a gateway has been constructed that interfaces to the server, simulates a collection of documents, and provides a content label describing the scope of the server. The lowest level of a

search then relies on the WAIS search engine rather than on content labels. At the other levels of the hierarchy, searching is accomplished by matching incoming queries against the content labels of each item in a given router's collection. As a result of the search, the content router returns the set of items in the collection that matched the query, along with suggestions for refinement. The protocols between the client and the servers are stateless at the granularity of user requests to the SFS. The local SFS software is responsible for keeping the state of the session and for expanding the search as requested (when the user opens a virtual directory).

There are a number of benefits to the content router system. First, the system implements a fairly efficient solution to the problem of searching a large number of information servers. The servers are not required to keep any state about the clients' session. The space overhead of the content labels has been shown to be low, and is in any case highly distributed, so the system does not use up excessive amounts of disk space. The system will also scale well from a standpoint of speed. Second, there is reasonable hope that the content labels may be generated mechanically, making a large hierarchy possible to construct. At the moment, content labels for the routers are generally at least partially constructed by hand, but the labels are well enough defined that it is probable that it will not be difficult to construct the required hierarchy.

Third, the SFS and the content routing system together provide a very uniform environment for queries and searching, which could be considered to be an improvement. Fourth, the system is designed to package all the state in one place, making the protocols simple and making the implementation and operation of servers much more convenient. Since the state is kept local to the user, the user pays the overhead involved with his searching. Fifth, the system supports both browsing and searching rather elegantly. The query refinement process essentially amounts to browsing down the hierarchy, doing a search at each level. Pointers into the hierarchy could be maintained by the user, similar to the "Hotlist" features supported by WWW browsers. Finally, the system can be easily and efficiently made accessible to larger populations of users by mirroring the top level parts of the hierarchy, while having fewer mirrors of the lower level or less popular parts.

Despite the many good features of the content router project, there are a number of reasons which indicate that it may not be the best solution to the problem we are attacking. The two main problems we see are the hierarchical nature of the routing network and the assumption of an *über*-query-language. In the design of the SFS and the content router, there was a very clear omission of issues relating to the impact of economics on the infrastructures of the future.[1] A hierarchy of servers, while convenient to the paradigm of directory structures, has several drawbacks. First, horizontal browsing of the network is constrained to siblings in the tree, so in order to browse to another branch requires an external network of pointers, such as a Hotlist or a parallelly constructed WWW-like network of references. In either case these must be independently maintained when new documents and servers are made available, which may pose problems as we will see shortly.

Second, a hierarchical structure cannot conveniently support competition, and therefore must be maintained as one single entity with a single beneficiary. Any competing servers (i.e. servers that cover essentially the same collection) must be on the same level of the hierarchy since they may not reference each other and continue to maintain an acyclic network. This serves to make the construction of competing services more difficult, since in order to add a service to an existing collection it first must be tested to ensure that the addition does not create a cycle. If server A references server B as well as a lot of other useful stuff, server B cannot give its clients access to that stuff in A by suggesting server A; it is forced either to duplicate A's functionality or to create a new server higher in the tree and to transfer its clients to the new server. Furthermore, if a significant amount of new data is put into the system, the need for a restructuring of the tree may develop, since it may become overly lopsided and bottlenecks may form. This can only occur in a planned fashion if there is some controlling body to mandate such a restructuring, although restructuring in an unplanned fashion could arise through competitive forces. It seems as if the intention is to generate the content router hierarchy mechanically, which would make programmed restructuring a possibility, but restructuring in this way would

[1] This strikes us as peculiar given the recent proliferation of Internet commerce.

disallow any sort of competition. As a side effect, any browsing techniques based on pointers will fail after a restructuring, since many servers will change dramatically or disappear.

The other major objection to the system regards the choice of a single standard query language. One might argue that standardization is not a bad idea, especially if the language is sufficiently extensible, since it will be a unifying force in the development of the infrastructure. However, creating a sufficiently extensible language has a high probability of failure, since we do not currently know what the future holds in terms of the types of data indexed and the types of indexing and searching that may be invented. The query language they currently use is well suited to a network of WAIS servers which take English text queries, but it may not work as well for searching databases of images, video, or music. It also may impose undue restrictions on databases which have more intelligent query interfaces, since the content router query language is keyword or attribute based. Although these restrictions are not particularly relevant now, fixing the query language in such a restrictive way may become a problem in the future.

Besides the two main objections, there a few minor qualms. The first has to do with the way the system works from the perspective of the user interface. It is unclear how deep the tree will become, and whether navigating the tree will become too tedious. This issue can perhaps be resolved with additional software on the user's end by making a program which takes a richer query and does the navigation, but in such a case new problems arise, such as the difficulty of making effective use of this richer language and of ensuring that the user does not spend too much money, assuming that the use of the servers is not free of charge. It seems that a richer language implemented in this way will lead to the problem this system intends to avoid, namely focussing the work of routing a query in one monolithic system. Such an interface would be difficult to implement if it is feasible at all. The last question regards the difficulty of making new types of service accessible to the network. At the moment, only WAIS servers are interfaced to the context router network, although plans have been made to extend this domain to include other information providers.

It is unclear how difficult it is to build these new interfaces.

In conclusion, there is no question that the content router system is an interesting experiment and that it has many redeeming qualities as a means of organizing a large collection of information services. As a global infrastructure it has some flaws, but as a means of providing access to a closely maintained collection of services it has great potential. The hierarchical structure lends itself well to automated construction, and also to carefully organized, browsable collections of data, similar to libraries and encyclopedias. However, this rigidity does not lend itself to the anarchic behavior of a growing on-line society, which is neither carefully organized nor particularly reliable. A period of anarchy will probably be necessary for the society to learn a good tactic for organization, and this system, while it might implement the eventual tactic well, does not provide the needed interim of flexibility. It also does not seem to lead to increasingly intelligent solutions to the problem of searching, but appears to implement a larger and more distributed but still attribute-based search technology. A good exposition of the idea of content routers is found in [27]; a follow-up paper is found in [8]. A description of a WWW gateway for accessing the system is found in [26]. This gateway is accessible from the PSRG home page.

## 2.3 Harvest

Harvest was designed and built by the IRTF-RD, or Internet Research Task Force Research Group on Resource Discovery. The principal investigator is Michael Schwartz of the University of Colorado at Boulder. The goal of the Harvest project was to design and build a system which constructs a large distributed index of documents available in the Internet. The system is formed from four basic parts: Gatherers, Brokers, Caches, and Replicators. Gatherers are programs that run on the machines which house information. They digest the information into summaries which are stored in SOIF objects (Summary Object Interchange Format). A manual entry describing the SOIF object can be found in [14]. This digestion is accomplished using the Essence software, which knows a great deal about different file formats and can

parse them to extract key information.[15]

The SOIF objects are collected and dumped to Brokers using bulk transfers. By transferring summary data in bulk, the Harvest system operates orders of magnitude more efficiently than other analogous systems. Other systems typically employ agents which download information from a source through the interfaces available to the users of that source, in the process not only initiating a huge number of transactions but also transferring whole documents and throwing most of the data away on the other end. Because Harvest's summary objects are built on the machine storing the data, significantly less data is transferred over the net as compared with the systems that download entire documents and summarize them on the other side. But the biggest savings comes from the fact that the Gatherer resides on the Archive machine and can thus efficiently detect new information as it is added to the store, rather than needing to search the entire archive using only the interface available to users.

Brokers accept summary objects from Gatherers and compile the information they want to keep into an index. Brokers then provide a query interface to the index and an retrieval interface to the archive. The actual indexing is done by a back-end index which is interfaced in a generalized way – WAIS, Nebula and Glimpse are currently supported. The query language supports boolean combinations of attribute queries. A number of Brokers have been implemented at the present time. These Brokers are accessed through the World Wide Web using standard Forms interfaces. A Harvest Broker entitled the "Harvest Server Registry" maintains a list of all registered Harvest Brokers, and that list is growing.

When the user decides to download an object, he sends a request (usually by clicking on a hyperlink in the report generated by a Broker) to the Broker asking to retrieve that object. Here another good feature of the Harvest design comes into play: object caching. The Broker which receives the retrieval request immediately begins trying to download the object from the archive where it is stored, and simultaneously sends identical requests to the nearest object caches. In order to improve retrieval time, reduce network traffic, and reduce the load on information servers, Harvest maintains a network of object caches. Each cache is connected to its nearest neighbors

and to nearby Brokers. The request bounces around the network, and if a cache is found which contains the requested object, that object is returned and all other searches are cancelled. After sending out the parallel requests, the Broker accepts the first response and cancels the remaining requests. If the object cannot be found in one of the caches, eventually it will be downloaded from the object's home location.

In order to reduce the load on Brokers, the Replication subsystem provides means of keeping a group of replicated Brokers consistent with a single master site.

The Harvest system is providing a large, distributed index of information in the Internet, and will hopefully continue to increase the scope of its coverage. However, as increasing numbers of Brokers come on line, some managing increasingly narrow collections of data, the central Directory of Harvest Brokers will become increasingly unwieldy. While Brokers can index other Brokers, the Harvest project has not devoted much effort towards the problem of navigating a network of meta-Brokers. Harvest is an excellent indexing system and has much potential in terms of scalability, but it doesn't really attack the problem which forms the topic of this thesis: How does the end-user discover which Index can help answer a given query?

The functionality offered by the system developed in this thesis could have been implemented within the Harvest model; however we chose to invent a new system, more focused on this specific part of the problem, so that we would have fewer constraints in designing the protocols and the system in general. One major constraint which we maintained was to make sure that Harvest can be used as a back-end indexing and search engine. The meshFind system has Referral servers which refer users to appropriate Harvest Brokers, mingled among Referral servers which refer users to appropriate servers of other varieties. At the present time, this seems like a reasonable way to solve this problem, because new, special purpose brokers are being invented regularly (one example is the AT&T 1-800 Telephone Directory Broker). A good technical discussion of Harvest can be found in [5] and [6].

## 2.4 Lycos

The Lycos search engine is one of a number of WebCrawler-like systems. Robots are sent out to retrieve and summarize Web pages, and a giant index is formed from the whole collection. For each document Lycos scans, it collects and returns information including:

- Title string
- Headings
- The most important 100 words
- The first 20 lines
- The size of the document in bytes

This information is brought back to Lycos central and is formed into a gigantic index. Fortunately, there is only about a total of about 40 GB of data on the Web, so constructing a single index is still possible for the time being. If the Web continues to increase in popularity at the present rate, however, this will no longer work (of course, the failure of the infrastructure will help to prevent the continued expansion of the Web).

There is some evidence to support the claim that the designers of the Lycos system eventually plan to invent a more distributed index, but there have been no overt attempts to do so. In one of the original design documents, the designer of Lycos states:

> I also subscribe to the dream of a single format and indexing scheme that each server runs on its own data, but given the current state of the community I believe it is premature to settle on a single format. Various information retrieval schemes depend on wildly different kinds of data. We should try out more ideas and evaluate them carefully and only then should we try to settle on a single format.[22]

# Chapter 3

# Design Overview

As with most systems of reasonable complexity, much of the details of the meshFind system's operation stem from the interaction of its various component parts. This makes it difficult to understand the system in a detailed way without first getting acquainted with all of the building blocks. The goal of this chapter is to give a macroscopic view of the system and its components, leaving most of the details to be covered in the following chapters.

The meshFind system is a distributed network of servers which work together to answer a client's query. The servers forming this network fall into three categories: Sponsors, Referral Servers, and Index Servers. Each of these three types of server has associated responsibilities and a set of protocols. The Sponsor has the responsibility of managing the search, deciding where to look, and getting the results back to the client. A Referral server has the responsibility of taking a query and suggesting an appropriate plan of action for solving it. An Index server is simply a gateway to an underlying source of data, which can be stored in just about any format or system as long as there are sufficiently powerful search capabilities. So in a typical session, a client would initiate a connection with a Sponsor, and send a query. The Sponsor then forwards that query to Referral servers which return their suggestions to the Sponsor. The Sponsor chooses from among the suggestions based on the client's preferences and requirements, and continues the search. Some of the suggestions from a Referral server will point to other Referral servers, and some will point to Index servers.

When pointers to Index servers are followed, actual documents will be retrieved and sent back to the Sponsor, which in turn forwards them to the client. The search ends when either an agreed-upon time limit is reached or the client sends a completion or cancellation notice. This process of forwarding the documents back through the Sponsor may in some cases be an unnecessary layer of indirection, but it does have two positive characteristics. First, the Sponsor might then have the option of providing a document cache service which could save time and network bandwidth. Second, the extra layer of indirection allows Clients to talk to the Sponsor with a point-to-point connection rather than a connection with full connectivity. The possibility of instead returning the result documents directly to the Client is explored in Chapter 8.

We will now take a slightly more in depth look at the types of servers and the types of data that are passed between them. After these building blocks are understood, the chapter will conclude with a short but detailed example of the operation of the system as a whole.

## 3.1 Servers

### 3.1.1 Sponsors

Of the three types of server, the Sponsor has the most complex specification. As we have seen, the overall purpose of the Sponsor is to accept a query from a client and attempt to send back an acceptable answer. However, the real job of the Sponsor is not so much finding places to look as it is deciding the relevance and feasibility of the available resources to the client. In order to answer the client's query, the Sponsor must certainly know a collection of good places to look, but much of the work of understanding the organization of the information services can be left to the Referral servers. Hence, it might very well suffice for a Sponsor simply to have a standard set of Referral servers which it always uses.

If we assume that the Referral servers do a reasonably good job, they will return Proposal objects containing all of the relevant places to look. However, some of these

suggested services may cost money, some may have better or worse time guarantees, and some may have been tried unsuccessfully by the client in the past. One way to solve this would be to have the Sponsor ask the client whenever a decision must be made. Unfortunately, that does not scale well when hundreds of services are being searched, and it may be the case (as with many novice users) that the client does not want to look at the process of searching at that level of detail anyway.

In order to resolve this difficulty, the client sends some additional information along with the query information, in what is called a Query object. This information includes the maximum amounts of time and money reserved for the search, and also a piece of Scheme code called a Filter. Filters will be described more later on, but essentially the Filter has access to the list of searches that the Sponsor could perform, and it chooses which searches to do, and the order in which they are done. Novice users would probably choose from among several standard Filters (*e.g.* Don't spend more than $X on any one service, always choose the fastest service first, don't spend at a rate greater than $Y/hr) but there is plenty of flexibility for more complex Filters. However, there is good reason to keep them fairly simple, since Filters are only allowed to run a certain number of steps before they are suspended so that another client's Filter can run.

With Filters taking care of all the decision-making, the Sponsor can sort through huge amounts of information, and check hundreds of services, finally boiling it down to (hopefully) a few documents which it sends over to the client. The Sponsor does all the work requiring a high bandwidth network connection, and might be connected to the client by something as inexpensive as a modem or a serial line. Because Sponsors can service many clients, it would be possible to take advantage of that fact by providing object caching at the Sponsor and bulk transfers between the Sponsor and the more popular Referral servers.

There is a facility built into the system which allows any of the participants in the search (i.e. Sponsor, Referral servers, etc.) to be given the question and answer that were eventually matched, if the client chooses to release the information. Many interesting things can be done with this information; for example, a sufficiently clever

document matcher could take new questions and check to see if something like it had been asked before. Another possibility along similar lines is to have the Sponsor provide more personalized searching by keeping track of a specific client's questions and answers and learning what sorts of things interest him and where these sorts of things are found. Although these issues are well beyond the scope of this thesis, the system was designed to allow for such possibilities.

### 3.1.2 Referral Servers

The Referral servers have the responsibility of taking a Query and resolving it into a Proposal which is sent back to the sender of the Query. The hope is that the Proposal that is sent back refers to services which can answer the query. The interpretation of the Query is done only by Referral servers, so the syntax or form of the question is really specified by the design of the Referral servers which get the questions. Since Referral servers are all essentially independent, except in as much as they refer to each other, there is little stopping new and more powerful Referral servers from being implemented and used.

In order for the Referral system to work, a fairly large network of Referral servers is needed. In the system we implemented at the Mesh Project, we constructed a network of about ten servers. Because of its small size, this system was able to answer questions in only a fairly limited domain. The construction of a full-size system would be a large project, not outside the realm of possibility, but probably requiring a significant capital investment.

The structure of the Referral network is fairly simple. At the "top" are general purpose Referral servers. A general Referral server is supposed to be able to take any question and direct the search in a good direction. These servers need not be terribly clever, but the problem is fairly difficult to do well. The general server that we implemented made the referrals based on the presence of certain keywords. Unfamiliar words were looked up in the index of the Encyclopedia Britannica, and that information was correlated to the known keywords. Given sufficient funds and manpower, we would suggest designing the Referral network in a way similar to

the organization of encyclopedias and libraries, with carefully chosen categories and subcategories.

Underneath the layer of general Referral servers are layers of successively more specific servers. These servers are not necessarily linked hierarchically, although that may be desirable in some cases. Most of these servers will cover a specific area of interest, for example U.S. Space Flight. In addition, there is a special class of Referral servers, called Leaf servers, each of which covers a specific information source, such as a Harvest or WAIS server. Searching such sources is the eventual goal of the Referral network; Leaf servers are especially necessary because they convert the meshFind question, which is not required to conform to the format of the specific underlying indexing services, into something that services such as WAIS or Harvest will understand. Like any other Referral server, a Leaf server will return a Proposal to the Sponsor, but the object returned will contain references to Index servers rather than to other Referral servers. The Sponsor may then send a Local Search Request to the Index server to retrieve the documents. Since it may often be convenient for a Leaf server to pre-fetch the suggested documents (that is, to cause the Index server to fetch and temporarily cache in anticipation of a Request), the capability to do so has been designed into the system.

### 3.1.3 Index Servers

Index Servers are the simplest servers in the system, and only deserve a brief note. When an Index server receives a Local Search Request, it uses the query data contained to perform the search or to send the search request to the underlying search engine. The results are then collected and returned as a list of documents or abstracts. The facility for pre-fetching has been designed into the system although we did not in fact implement it.

### 3.1.4 Client-side Software

The software needed to run a client is not very complex. The interface runs perfectly well using text only in an Xterm window. Alternatively, by running a web server on the machine running the Sponsor, it is possible to provide a simple web interface. The one flaw is that the client will either have to poll the server occasionally to see if any new messages have arrived, or potentially to remain connected until the new messages come in. The interface is slightly awkward, but essentially there is a selection next to the submit button which chooses whether the client wants to check for news and return immediately or just stay connected and wait for the answer.

## 3.2 Data Structures and Messages

In this section, we will give a brief introduction to the data structures and messages used in the meshFind system. In the meshFind system the messages are represented as objects in a hierarchy; many of the different message objects have parts in common. This is just a stylistic convenience, and need have little impact on the physical transport of information. Apart from the message object hierarchy itself, there are some additional data structures which form the types of some of the fields of the message objects. We will first turn our attention to these underlying data structures, and then go on to cover the message hierarchy itself.

### 3.2.1 Underlying Data Structures

The most common of these data structures is the Address structure. This structure stores address information, including a hint which may give a location, the OID[1] of the addressee, an "attention" OID, and an addressee information block.

[1] OID is a common abbreviation for Object IDentifier, and two important properties of an OID are traditionally uniqueness and immutability. In these respects they are similar to URNs, as described in [29]. OIDs usually have the additional property of telling nothing about the location or content of the object which they identify. The MESH system uses OIDs to identify objects, and several systems are currently being constructed which attempt to locate an object given its OID. A good paper describing many aspects of the MESH system can be found in [30].

The attention OID is an optional field which is used to identify an object local to the server receiving the message. In cases where a message is the response to one of several queries with the same Search ID, the attention OID allows the response to be easily paired up with the object to which it is related. Because it relates to an object local to a server, the attention OID is only useful for sending a response back to the server that issues a request containing it. That is, it can be placed in a return Address so that the response sent to that Address is properly interpreted. Such methods are necessary because the meshFind protocols are not stateless, and consequently at any given time a Sponsor may be awaiting responses from many different servers on behalf of a single client. Although in many cases the return address is sufficient to distinguish these responses, in the interest of simplicity the attention OID is used instead.

The information block is not completely specified, but certainly contains information such as a description or abstract describing the service and the cost of the service it describes. The specifications of cost, for example, have been left somewhat loose because it is difficult to predict the direction of the current push for commercial Internet services.

Another simple data structure is the Document List. These are lists of structures which contain either a whole document or an abstract combined with some information telling how to get the rest. These structures are returned by Index servers after a successful search.

The third data structure we will discuss is the Plan structure, the data structure underlying the Proposal object. Plans are intended to represent strategies for searching. A Plan may be one of three types: a Serial Plan, which contains a list of sub-plans to execute in order, a Parallel Plan, which contains a list of sub-plans to execute concurrently, or a Plan Element, which represents either a reference to a Referral server or a search of an Index server. By composing these types of Plan, a hierarchical structure can be formed, with the actual searches at the leaves. Typically, Referral servers try to organize their suggested searches by relevance, and the Plan they form is a series of parallel Plans, starting with the most relevant group of searches and working down. This structure is only one possibility, but having some

kind of standard structure makes the job of writing Filters considerably easier.

The Filters we mentioned before make up a fourth data structure. These are Scheme programs, represented as a list of objects. The programs can reference a certain set of pre-written standard procedures. These procedures allow the Filter to interpret and manipulate Plan objects. The goal of a Filter is to decide which parts of a Plan to execute, in what order, based on issues of cost, time, and other information. As is the case whenever foreign code is executed on a server, there are many important issues of security associated with the design and implementatation of any server that runs these Filters. These issues are discussed at length in the next chapter.

### 3.2.2 Messages

As was mentioned briefly before, the messages forming the meshFind protocol are represented for our purposes as an object hierarchy. The root type of the hierarchy is the Search object. Every Search object has a Search Identifier, which is an identifier common to all objects that are used during a client's session. All communication with the Sponsor that is related to that session is tagged with the same Search ID. Every Search object also has both a To Address and Return Address, which is one of the Address structures described in the previous Section.

In general, search objects form two basic categories: Requests, which are the objects going away from the client, and Responses, which are those coming back. "Simple" messages make up a minor exception, since for simplicity of implementation all messages conveying only a flag are in the Response category.

There are three types of Request in the hierarchy: Unsolved Query object, Solved Query object, and Local Search Request. The Unsolved Query is the object the client initially sends to the Sponsor to begin a session. These objects are also forwarded by the Sponsor to the appropriate Referral servers. The Unsolved Query stores question information, quantities of time and money available, a Filter program, and several other less important fields. When the word query is used in this document, typically it refers to an Unsolved Query object.

The Solved Query object can be interpreted as an indicator that the search is successfully completed. The Solved Query also contains both the question and the desired answer, so servers that collect these objects might implement some kind of caching scheme. Such things are beyond the scope of this document.

The Local Search Request essentially contains a string of commands intended for the Index server to which it is sent. As we have seen, these commands are determined by a Leaf Referral server, in response to a Query.

There are also three types of Response object, but they are generally simpler to describe. Most types of Response are of the "simple" variety – that is, they are simple messages such as `sponsor-accepts-query`. There are two other kinds of Response, a Proposal object and a Bundle object. A Proposal is a Response object containing a Plan. The reply sent back from a Referral server is a Proposal object containing the suggested Plan. A Bundle is a Response object containing a List of Documents, typically returned by an Index server.

## 3.3 A Brief Example

The following is a simple example of the interaction of all these parts. The diagram in Figure 3-1 should help to keep the communication straight. In the diagram, all messages passed in the example are indicated by arrows. The number next to the head of the arrow corresponds to the step number in the following series. The message passed is indicated in the oval in the middle of the arrow. The various servers involved are represented by the large boxes and ovals. In order to keep the diagram understandable, the content of the messages is referenced by the single-letter names used in the following description, and most of the steps that did not involve message-passing were not indicated.

In this example, a very simple transaction takes place. The Client formulates a Query and sends it to a Sponsor, which from the Client's perspective takes care of the whole process. After some amount of processing throughout the network, the results filter back up to the Client, who makes judgements about their relevance and quality.

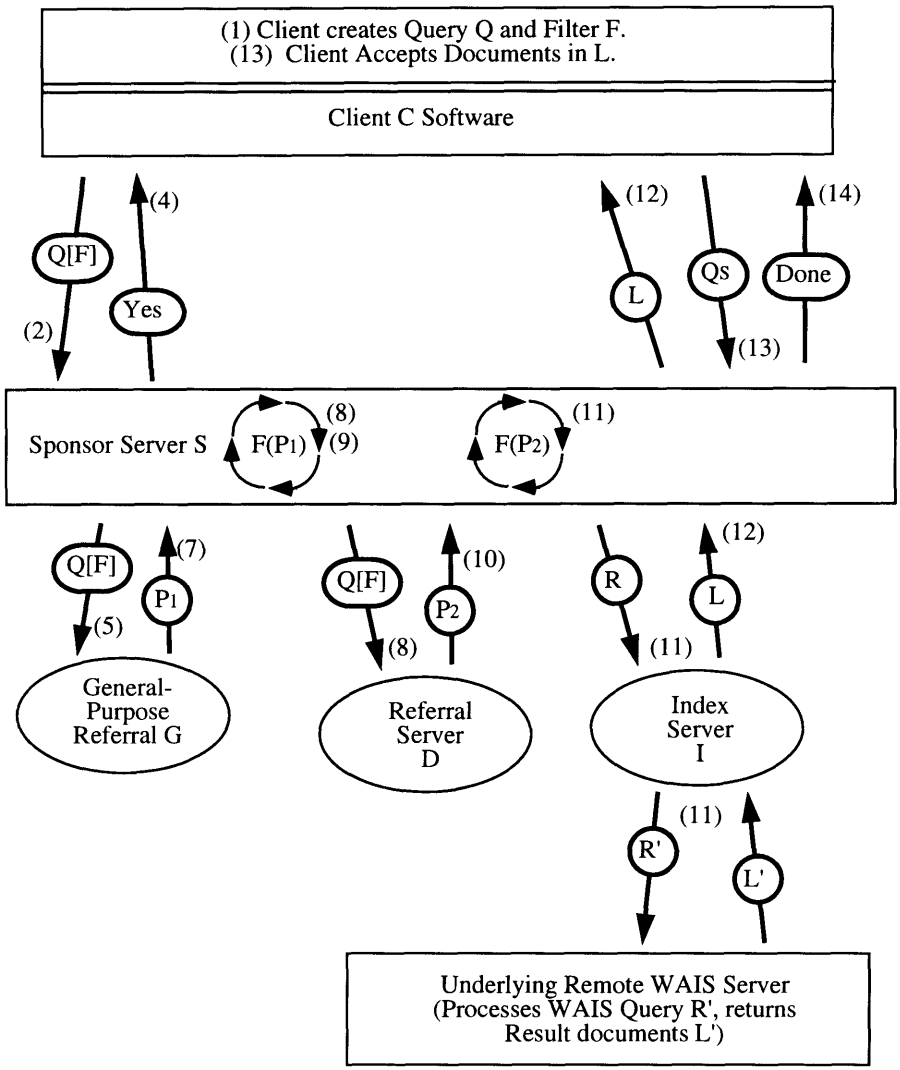


Figure 3-1: A brief example of the meshFind system in action.

The processing done by the Sponsor includes running the Client's Filter program and sending the Query and, eventually, the search Requests, out to the network of Referral and Index servers. At the very bottom of the process, actual information services search their proprietary databases and return the requested information to the Client through several layers of proxy servers.

1. Client C thinks of a question and builds an Unsolved Query object Q containing Filter F.

2. Client C sends Query Q to Sponsor S.

3. S examines Q, and decides to sponsor it:

4. S sends a simple message with the token `sponsor-accept-query-token` to C.

5. S sends Q to the general purpose Referral G.

6. G receives Q. Upon analysis of Q, G forms a two-step plan $P_1$:

   (a) Step 1 is to try Referral servers A, B, and D.

   (b) Step 2 is to try Referral server E.

7. G sends $P_1$ to S.

8. S receives $P_1$, and runs F on it. F runs through the plan and chooses to try Referral server D first because of the more relevant servers it is the cheapest. S then executes that portion of the plan, which causes S to send Q to Referral server D.

9. S continues to run F on the plans, and perhaps other parts are executed.

10. When Referral server D receives Q it responds by sending a plan $P_2$ containing a Local Search Request R back to S.

11. S runs $P_2$ through Filter F, and F chooses to execute this Local Search Request. R is sent to the specified Index server I.

12. I processes the request R by initiating a connection to an underlying WAIS server. Server I then interprets the response from the WAIS server, and constructs a list of documents L, which is returned to the sponsor S. L is in turn forwarded back to the client.

13. The client decides that those documents are good, and sends out a Solved Query object $Q_S$. The sponsor receives this Solved Query and stops the search.

14. To conclude the session, S then sends a simple message containing the token `sponsor-termination-token` back to the client.

## 3.4 Conclusion

This concludes the introduction to the design of the meshFind system. In this overview most of the details have been left out; these will be filled in by the next four chapters. In Chapter 4, the data structures shared throughout the system and the structure of the messages passed between the servers will be covered. In Chapter 5, Sponsor servers will be covered in detail. Since they are the path of communication between the Client and the Referral and Index servers, most of the meshFind protocol is in some way involved with the Sponsor. Along with the protocol, many specifics about the implementation of our Sponsor server will be explained. In Chapter 6, the details of Referral and Index servers will be given. Again, along with a discussion of the protocol, a large portion of the text will be devoted to implementation details, which in this case consists of issues involved with connecting the meshFind system to existing services. In Chapter 7, the design of user interfaces to the meshFind system is discussed. The last Chapter provides a summary of the implementation, some conclusions resulting from our efforts, and many ideas for future work.

# Chapter 4

# Data Structures and Messages

In this chapter, we will describe in greater detail the data structures and messages used in the meshFind system. The shared data structures used in a software system make a good foundation for understanding the way the parts of the system fit together. In the context of this system, they are especially important, since they are the substrate for the protocol which links the various servers in the system together.

## 4.1 Addresses and OIDs

The first structure we will describe is the Address structure. Almost every part of the meshFind system uses instances of these objects; for example, every message passed contains two Address structures, one describing the sender and one describing the recipient. The following table lists the fields of which the structure is composed and their corresponding types:

| Structure addr | |
|---|---|
| `server-last-loc` | URL |
| `server-oid` | OID |
| `attn-oid` | OID |
| `server-info-block` | Info Block |

Table 4.1: The Address structure

In general, Address structures are used to identify and locate principals in the meshFind system, much as street addresses and email addresses are used in everyday life. More specifically, the Address structure has two key functions. Its foremost function is to store the information needed to send a message to one of the principals in the meshFind protocol, in the process specifying the particular transaction to which the message relates. Its secondary function is to provide information about the service to which it points.

The network location of the addressee is determined from the first two fields of the Address structure. The first field stores a hint which may give a recent location of the server, in the form of a URL, or Uniform Resource Locator. This hint is entirely optional and can not be assumed to be present, but if the system is working properly most of the Addresses being passed around will have a valid hint. If the hint fails, the server must be found using the server object ID contained in the second field. This OID normally contains no location information, and must be resolved through a name resolution service. Since in the MESH environment OIDs are the primary means of naming objects, such services are indispensable and can be assumed to exist and to be relatively efficient, given a functioning MESH infrastructure.

Under certain conditions, a message that is sent to a server relates to a specific transaction or session on that server. For example, if a Proposal object is sent to a Sponsor server in response to a specific Request made by a specific Client, information telling which Client and which Request must be included so that the message is dealt with properly. This information is transmitted in the `attn-oid` field of the Address structure. Essentially, the field contains an OID of an object local to the recipient server which is then interpreted by that server to determine which part of its system should get the message. Since the meaning of these OIDs as session, transaction, and thread identifiers is only understood by the Server which will eventually receive and interpret them, the contents of the `attn-oid` field must generally be determined by the server which will in the end interpret that value.

This is typically accomplished by including the appropriate value in a Return Address of a Request; when the Response is sent back, that Address is used as the

"To" Address. For example, when a Sponsor sends a Query object to a Referral server, the `attn-oid` field of the Return Address is set to the OID of an object which represents the plan to send that Query object out to that Referral. When the Referral server receives this object, it processes it using its various techniques and generates a Response. The Return Address from the Query object (containing the filled-in `attn-oid` slot) is then placed in the "To" Address slot of the Response, and the Return Address of the new Response is set to the Referral server's Address. This Response message is then sent out, and after some delay it arrives back at the Sponsor. The Sponsor then interprets the `attn-oid` field of the "To" Address and internally routes the new Response to the appropriate thread of the server, where it can be processed.

The `attn-oid` field is an artifact of the stateful nature of the meshFind protocols. Many protocols avoid statefulness because of the increased complexity that it brings with it; however, as we mentioned briefly before, the meshFind protocols are not stateless, despite the costs of increased server and protocol complexity. This design choice was made after a great deal of thought about how the meshFind system might be structured. Before we go into the reasons for making the meshFind protocols stateful, let us first review the benefits and limitations of stateless and stateful protocols.

The major benefit of stateless protocols is simplicity. The stateless protocols that might be relevant to the meshFind system start out as the basic Client/Server model: a Client makes a connection to a Server, one or more transactions are made over the established channel, and the connection is closed. The key feature of the protocols which makes them *stateless* is that no state relating to a particular transaction is kept *after that transaction is completed*[1].

[1]What makes this notion of "stateless protocol" confusing is that a protocol is only stateless when looked at from a high enough layer. Clearly some state must be kept in order to provide reliable datagram transmission, for example. However, when one builds a higher protocol layer in which each transaction is independent of the others (i.e. they can be reordered arbitrarily, etc.), then that protocol is termed "stateless", regardless of the state kept during the processing of each individual transaction. In the context of this discussion, we are concentrating on the layer of individual connections from Client to Server, so a stateless protocol is one which does not keep state between connections, whereas a stateful protocol is one in which a single transaction between

This makes the implementation of Servers simpler in many ways. The biggest simplification comes from the simplification of failure modes and error handling. Usually in stateless protocols such as HTTP, this work is handled by underlying software and is hidden from the implementation of the topmost (stateless) protocol layer. In the case of a HTTP, the work involved with keeping state is handled by the underlying TCP implementation. Since in HTTP each transaction conincides exactly with a single TCP connection, the implementor of an HTTP server does not need to worry about the details of error recovery and handshaking – unless something really goes wrong and the TCP connection times out or closes unexpectedly.[4] As the extent of exposure of the underlying state increases at the Server's protocol level, both the chances that some failure occurs and the number of ways that failure *can* occur likewise increase. For example, suppose that our stateful protocol requires that the Server respond back to the Client using a separate TCP connection. The Client might go off line after the initial connection, necessitating repeated attempts to respond. The Client might also give an invalid or incorrect address, resulting in miscommunication. Finally, the Server might go down or somehow lose track of the Client's transaction, in which case the Client would be forced to try to start the process over. In a stateless protocol, all problems of this sort are handled transparently by underlying layers.

Apart from the issue of failure, stateless protocols tend to be generally simpler, making many of the issues involved with server construction simpler. A stateful protocol requires that memory or disk space be allocated to any partially completed transactions. If the transactions are active for more than a relatively brief time, this storage overhead can get to be a real problem. Stateless protocols need only store enough state to handle the fixed number of transactions that the Server is capable of processing concurrently, which is a much more manageable constraint.

Stateless protocols also have limitations, however. These limitations generally stem from constraints imposed by the underlying protocol layers. The biggest limitation involved with a stateless protocol is the length of time that can be allocated to any given transaction. For example, if the protocol is implemented above TCP, there

a Client and a Server spans several connections.

may be intrinsic limits on the length of time that a connection can remain idle, and the operating system may limit the number of connections that can be maintained at one time. Since the number of transactions that can be processed concurrently is limited to the number of connections that can be open at once, in order to achieve high throughput each transaction must be completed rapidly so that the connection can be allocated to another Client.

With stateful protocols, the time allotted to a given transaction is constrained only by the Server's storage resources, since as the average time per transaction increases, the number of concurrent transactions will also increase, but the number and duration of connections need not. Servers running stateful protocols can also have higher throughput, even if the latency of an individual transaction is increased. This is because they can do a much more efficient job of pipelining, since the number of transactions being processed concurrently is limited only by the speed at which the starting and ending connections can be made and by the amount of storage space which can be devoted to unfinished transaction. Although stateful protocols require more connections per transaction, this problem can be ameliorated to a degree by collecting any responses to the same address and batching them into one connection. This has the drawback of further increasing the complexity of the protocol, but may help in situations where a Client makes frequent requests to the same Server.

In the context of the meshFind system, we found that many of the constraints imposed by the problem indicated that despite the additional complexity stateful protocols would serve the purpose better. There are two primary factors which lead to this decision. The first of these factors has to do with the limitations that stateless protocols place on the amount of time that a single transaction may consume. The object of the meshFind system is to provide the Client with strategies for locating and searching the right information services. If we had chosen a stateless protocol, we would have to limit the amount of time devoted to serving any given Client, which would preclude a wide variety of techniques. For example, certain Queries might be better solved after making an initial connection to an Encyclopedia in order to look up lists of topics related to an unrecognized word in the Query. If such a query takes

a minute to process, that would place a great burden on a server that runs stateless protocols.

An even more compelling reason for stateful protocols is the Sponsor server. The Sponsor serves an important purpose: it provides a proxy which manages the Client's search, in the process taking care of a great many details with which the Client need not be bothered, while maintaining close contact with the Client and dispensing the Client's funds in accordance with the decisions of the Client's Filter program. Given that in a full-size meshFind system a Query might pass through ten or a hundred separate servers, the Sponsor is a necessary part of the system. While the functionality of the Sponsor could be provided by a program on the Client's local machine, by offering Sponsors as separate servers they can become a shared resource and can also potentially reduce the costs of communication by making multiple requests in the same connection. A single session between a Client and a Sponsor might last for an hour, potentially with relatively long delays between the times when the Client's search is actually progressing at the Sponsor (since Referral servers and Index servers may take time to process a Query). If a Sponsor is to serve multiple Clients, then probably stateful protocols will be necessary to maximize the throughput and average utilization of the Sponsor.

Some of the problems with stateful protocols which we have described in the preceding arguments are mitigated to a degree by the nature of the problem that meshFind is designed to solve. The meshFind system is intended to provide essentially "best-effort" service. The stringent requirements of a service such as mail delivery need not apply to the problem of searching, since when a request to one server is temporarily stalled it is difficult to quantify that as success or failure. The Plans returned by Referral servers are not guaranteed to work for the Client – they are merely suggestions for possible avenues of search. When taken individually, the response from a single Referral server should not be a bottleneck in the critical path from Query to solution; there should always be alternate routes around any particular blockage. This means that while graceful modes of failure must be designed into the system, elaborate error handling is probably unnecessary.

Because of the decision to design stateful protocols, the meshFind protocols and servers are of increased complexity, and the `attn-oid` field of the Address structure is evidence of this. As a rule, there are few places where the attention OID is really necessary, but in most cases it makes things simpler to a sufficient degree that it is worthwhile. And, since many of the objects already have OIDs[2], they make convenient handles.

The secondary function of the Address structure is to provide information about the service which it addresses. This function is accomplished by the `server-info` field of the Address structure. This field always contains a Server Info structure, which in turn contains information such as how much the service costs, who can access it, and what sorts of services it provides. The following table describes the structure of the Server Info structure. To some degree the contents of this structure is really beyond the scope of this work. Specifically, the issues of cost and security have been covered only superficially; for example, the `cost` field is specified as a simple price with no mention of rates or units. However, such issues can be addressed later with minimal changes to the overall design.

| Structure `server-info` | |
|---|---|
| `name` | OID |
| `kind` | symbol |
| `abstract` | string |
| `cost` | dollars |
| `typical-delay` | time |

Table 4.2: The Server Info Structure

The `name` field identifies the server. When a Server Info structure is included in an Address structure, the `name` field of the Server Info structure should match the `server-oid` field of the Address structure. The `kind` field of the Server Info structure is a Scheme symbol which indicates the type of the server being described. The `kind`

[2]Whether or not to assign OIDs to the various objects in the system was an issue of perennial difficulty. Most of these objects are not things that will need to be found, since they either exist only briefly, or are internal to the servers and are not intended to be accessible. However, since OIDs make referring to them easier, they have been included in many of the data structures.

field is generally taken from a set of three symbols: {`sponsor, index, referral`}. In a future revision, the `kind` field may be filled in with an OID which names a role associated with the given type of server. That role OID could then be used to look up methods for manipulating the other data stored in the Server Info structure. This could make it easier to integrate new types of servers into the system.

After the `kind` field, the most important field of the Server Info structure is probably the `abstract`. This string in intended to give some idea of what sort of service can be expected from a given server. A more sophisticated Filter program could potentially use this information to aid in the decision-making process, although because of the inherent difficulty in making such a Filter effective, the system is designed to make this unnecessary to the greatest degree possible.

The `typical-delay` field stores an estimate of how long the server usually takes to process a request. This is used to determine when a Sponsor should assume that something went wrong in a negotiation with a Referral or and Index server.

## 4.2 Document Lists

A Document List is a list of Document structures which as a rule is produced by an Index server in response to a Local Search Request. The Document structure is loosely based on the Document ID structure used in the WAIS system[17].

| Structure `document` | |
|---|---|
| `oid` | OID |
| `headline` | string |
| `relevance` | integer |
| `cost` | dollars |
| `size` | integer |
| `server-addr` | Address |
| `data` | (optional) string |

Table 4.3: The Document Structure

A Document structure can contain either a full document, if the `data` field is filled in, or all of the information necessary to determine the document's relevance and

to retrieve the document. In order to retrieve a document given such a structure, a Local Search Request must be sent to the Index server which produced the Document structure. This request contains the list of documents to retrieve in its `commands` field, and is sent to the address stored in the `server-addr` field of the Document structure. Whatever payment authorization is needed must be sent along in the `security-info` field of the request. Typically, a client would decide which documents to download since is may cost money and time; these decisions are routed through the Sponsor to the appropriate Index servers, and the retrieved documents are routed back.

## 4.3 Plans

The Plan structure provides a means of representing a search strategy. Because of the way the meshFind system works, in order for such a structure to be useful it must be designed to be mechanically assessed. Specifically, the structure of Plans must be geared towards what can be readily processed using a Filter. The following table describes the structure of a Plan.

| Structure `plan` | |
|---|---|
| `parallel?` | boolean |
| `time-allocated` | time |
| `items` | list of Plans or `plan-elts` |

| Structure `plan-elt` | |
|---|---|
| `prefetched-until` | time |
| `cost-estimate` | dollars |
| `relevance-est` | integer |
| `time-allocated` | time |
| `attn-code` | OID |
| `send-to-addr` | Address |
| `request` | (optional) string |

Table 4.4: The Plan and Plan Element Structures

Table 4.4 shows the two structures of which Plans are formed. Essentially, Plans are constructed hierarchically out of `plan` structures, and at the leaves of the tree structures of type `plan-elt` are filled in. All of the actual references are stored in the leaves, and the `plan` structures only serve to organize and categorize the references.

Structures of type `plan` contain a list of sub-plans in the `items` field. This list may be interpreted either as a list of concurrent plans or as a series of single steps.

This interpretation is determined by the value of the `parallel?` field: a true value means that `items` contains a set of concurrent plans. There is no intrinsic restriction on the depth or complexity of the tree of plans, but keeping in mind the construction of Filters it is advisable to stick to a fairly standard form. The form we use in the demonstration system is simply a series of concurrent plans, each of which contains groups of references judged to be of commensurate relevance. The series lists the most relevant group first and continues in order of declining relevance.

Let us now examine the leaves of this structure more closely. The fields of the `plan-elt` structure are a mixture of data intended to be communicated by a Referral server and data which is convenient to the internal workings of the Sponsor. Since the Plan structure forms the bulk of a message sent from Referral servers to the Sponsor, this mixture is evidence of flawed design, but due to time pressure the necessary design changes were not feasible. Consequently, the `time-allocated` and `attn-code` fields need not be filled in by a Referral server – they are reserved for use within the Sponsor. The other fields are intended to be filled in by the Referral server which creates the plan.

We will now explain the purpose of the fields that are provided by a Referral server. The first field, `prefetched-until`, stores the time at which the pre-fetched documents will be dropped from the cache, or is set to false if no documents were pre-fetched. In the present system, pre-fetching has not been implemented, so this is always false.

A thorough discussion of the `cost-estimate` field is probably beyond the scope of this work, but we will at least explain the intent of the field here. The main issue having to do with the sale of networked services in this system is the fact that the client's decisions are made by proxy through the action of the Filter. As a result, when services are offered for sale it is the Filter which must decide which to buy, and it is the Filter which must ensure that the purchased services are in fact received. Since placing all of this responsibility on a small piece of code is dangerous, it makes sense to think of ways of building some kind of checking into the system. The cost estimate is one method of redistributing the responsibility: when a Referral server

makes a reference to a service which is not free, the Referral server is in fact re-selling that service, at the price quoted in the cost estimate. The reference itself would be a coupon issued by the Referral server which can be redeemed for the promised service; the service provider privately bills the Referral server, and the client pays the Referral server according to the cost estimate associated with each reference. In this way, it is the Referral server's responsibility to ensure that the cost estimate is no less than the actual cost, and competition among the Referral servers should cause the estimates to follow approximately the true costs. This is especially believable when one considers that the Referral servers would probably get some kind of bulk contract with their "suppliers", probably meaning lower prices than would be available directly. However, the details and protocols associated with these binding estimates are beyond the scope of this work.

The third field is an integer describing the estimated relevance of the reference. These values only have meaning with respect to the other relevance estimates in the same plan, so in order to make comparisons of elements of different plans, the relevance estimates may need to be normalized. It is difficult to estimate the importance of relevance estimates in the construction of effective Filters.

The `send-to-addr` field contains the address of the suggested server. This server may be either a Referral server or an Index server. If the address is that of a Referral server, then when this plan element is executed the existing query will be sent to that address for processing. We considered the possibility that a sufficiently clever Referral might be capable of forming a Plan which included modified Query objects, customized for specific servers. The customizations would not constitute revisions in the normal sense, but would be more akin to highlighting certain parts of the query. This would certainly increase the potential power of Referral servers, and the capability would not be difficult to add, but it is presently not available. If the `send-to-addr` field refers to an Index server, then when the plan element is executed, a Local Search Request is created and sent to the given address. The `request` field, which is otherwise set to false, must in this case contain a string that will be stored in the `commands` field of the Local Search Request.

## 4.4 Filters

In the previous chapter the structure of the Filter object was described completely, but the operation of a Filter was not explained. As was stated before, a Filter is a regular list of Scheme symbols, and it is interpreted as a program. In the context of the Sponsor, there are certain requirements which must be met in order for the Filter to work. These issues will be discussed in this section. But first, we will discuss the way in which Filters are processed in the meshFind system.

Whenever pieces of code are to be imported and executed, a number of important security issues crop up. Extreme care must be taken to ensure that the foreign code neither crashes nor hacks the system. Since code cannot in general be screened for infinite loops, the system must also be prepared to terminate processes which run too long. These problems are solved in the meshFind system through the use of a simplified Scheme interpreter. The interpreter yields reduced performance, but provides many positive features which will be discussed below.

Several groups are developing systems for portable code. Since in Scheme it is not too difficult to implement such a system, and since existing systems such as Olin Shivers' Scheme-based Webserver[28] were not readily available in the MESH environment, it turned out to be more expedient in the short term to implement a simple package to do what we needed.

### 4.4.1 The Filter Interpreter

The interpreter used in meshFind is designed to implement a reduced version of the Scheme language. In order to fulfill the requirements of security, several additional features have been included. First, Filters are allowed access only to the limited set of functions needed by the Filters that are to be run. This means, for example, that no functions are provided that allow any kind of file I/O or access to memory outside the scope of the Filter process itself. Control over the memory and functions that are available to a Filter is maintained by limiting the Filters to reference only the procedures and variables that are contained in their associated "environment", and

by placing in that environment only safe references. Second, in order to cope with the possibility of Filters which either contain an infinite loop or simply need to run for longer periods of time than is normally allotted, the interpreter is designed to allow a program to be interrupted during execution and later continued from the point of the interrupt.

### A Brief Introduction to Scheme

Before beginning an explanation of our particular implementation, we should first give a very brief introduction to the Scheme language. For a very good and thorough introduction to Scheme, see [1]. Scheme is a lexically scoped variant of LISP in which procedures are first-class objects. A calculation is made in Scheme by evaluating an *expression* in the context of a given *environment.* Here an expression is a list containing symbols and other lists, and an environment is an object which can be used to map symbols to their bound values. For example, the list `(+ 1 2 x)`, when evaluated in the context of an environment which maps `x` to 5 and maps `+` to a procedure which performs addition, would result in a value of 8.

Going into slightly greater detail, when a list such as `(+ 1 2 x)` is evaluated, a two step process ensues. First, each element of the list is evaluated, in an unspecified order. When the symbols `1` and `2` are evaluated, the returned value is simply the corresponding numerical value. When the symbols `+` or `x` is evaluated, the environment is searched for entry binding that symbol to a value, and if a value is found it is returned. If a value is not immediately found, it may be the case that the symbol was bound at a previous level of scoping. In order that such bindings might be found, every environment has a pointer back to its parent environment. For example, an expression within a procedure will be evaluated in an environment which is removed from the global environment; hence any references to global variables from within the procedure will be found only by following the pointer to the parent environment. If the search reaches the global environment and still fails, an error has occurred. In all of this, procedures and other objects are treated in the same way, although naturally they return different values. A symbol that refers to a procedure will return a special

kind of object, a procedure object, which has the property that it may be *applied.*

This brings us to the second step in evaluating an expression. Recall that in the first step, each item in the original expression was evaluated and the resulting values have been collected into a new list. We now try to apply the first value in the list to the rest of the list of values. In order for this to work, the value of the first item in the list must be a procedure object, since applying a procedure to a list of values essentially means running the procedure with those values as arguments. When a procedure is applied, the formal arguments specified in the text of the procedure are bound to the values being applied in a new environment whose parent is the environment that was active when the procedure was defined. There is one additional special case regarding procedure invocation: Scheme features *tail-recursive* procedure calls. That means that when the last form in the body of a procedure is a recursive call (that is, a call to itself), rather than building up a new stack frame to keep track of the new invocation, the new values are inserted into the old stack frame and the procedure body is re-executed. This allows such tail-recursive procedures to loop without using up additional memory, answering the traditional objection to recursive programs. Thus the basic process of interpreting a Scheme program is summarized.

Data structures in Scheme are composed of two basic types, *pairs* and *vectors.* Pairs are objects composed of two parts, traditionally called the `car` and the `cdr`. They are created using the procedure `cons`, which takes the car and the cdr as arguments and return a newly constructed pair. Vectors are similar to the arrays used in many high level languages, and have associated procedures for creating new vectors, and for setting and referencing elements of a vector. Scheme uses a memory manager based on *garbage collection*, which means that chunks of memory are allocated for use until no more memory is available, at which point all unreferenced blocks of memory are located and made available again. This memory management technique allows Scheme programs to be conveniently written in a "functional" programming style. In functional programming, procedures do not return their results by *mutating*, or modifying, existing data structures. Rather, the values to be returned are stored in newly constructed structures and those structures are returned to the caller as a

return value. The primary compound data structure used in Scheme is the *list*, one which is well suited to functional processing. Lists are formed by a chain of pairs in which the values in the list are stored in the car and the next pair in the chain is stored in the cdr.

### The Implementation of the Filter Interpreter

We will now begin to discuss some of the details of our particular implementation, focusing on those that provide the special features of interruptability and robustness. The Closure, Environment, and Procedure structures shown in Table 4.5 are central to the implementation of the Filter interpreter.

| Structure `closure` | |
|---|---|
| `exp` | list |
| `env` | `envir` |
| `continue` | false or list |
| `result` | value |
| `error` | boolean |

| Structure `envir` | |
|---|---|
| `bindings` | list |
| `back` | `envir` |

| Structure `proc` | |
|---|---|
| `formals` | list |
| `env` | `envir` |
| `sequence` | list |

Table 4.5: The Closure, Environment, and Procedure Structures

In the explanation of Scheme given in the earlier part of this section, we saw how procedures, environments, and expressions interact within an evaluator. The `envir` and `proc` structures specified above represent in a fairly straightforward way an environment and a procedure object respectively. The `closure` structure represents the state of a Filter program during evaluation.[3]

[3]Readers who are familiar with the standard concept of a closure may be somewhat confused by the names of the fields in the Closure structure. A closure in the most general sense is a mechanism for associating a context containing names with the context in which the names may be interpreted. In this case, the `env` field contains the "environment", which lists a collection of variable bindings. The `exp` field points to the expression which is being evaluated. This expression may be a procedure object or may be an expression taken from the body of a procedure, depending on the level of evaluation that is being processed. This implementation does not conform strictly to the standard form, mainly because of limitations on the time that was available for implementation.

The `envir` structure represents an environment. The `bindings` field stores an *association list* which maps the variables to their respective values, while the `back` field stores the `envir` structure which describes the parent environment. An association list is a list of key-value pairs which can conveniently represent a mapping. Each element of the list is a pair whose car is a key and whose cdr is a value. In this case, the keys are the symbols which name the variables, and the values are the objects stored in the variables.

The `proc` structure represents a procedure object. The `formals` field stores a list of symbols that name the arguments which the procedure expects upon invocation. The `env` field stores the environment that was active when the procedure object was created. The `sequence` field stores the body of the procedure itself. When a procedure is invoked on a list of argument values, first a new environment is constructed. A value for the `bindings` field is formed by pairing each element of `formals` with its corresponding element of the argument list, while the value of the `back` field is set to the contents of the procedure object's `env` field. This new environment will contain all the necessary local bindings, as well as inheriting any other references accessible within its lexical scope. Then, the body of the procedure is evaluated in the context of the new environment. As was mentioned earlier in this section, a special case occurs when a tail-recursive call is performed. In such a case, rather than forming a new environment, the existing environment is modified and the body is re-evaluated.

The `closure` structure is intended to represent the state of a Filter program at a given point in time. During the evaluation of a Filter, the complete state of the Filter process is contained within the combination of the associated `closure` structure and the state of `filter-eval`, the *evaluator* program itself. The `filter-eval` procedure implements the following rules governing its inputs and outputs:

1. The `filter-eval` procedure takes as an input a `closure` structure representing a Filter process at some point in time. To represent a Filter process that has not yet been run, we define a `closure` with the following field values: `exp` contains the text of the Filter to be run; `env` is set to an initial environment containing all of the standard procedures, usually the result of calling the

`make-initial-environment` procedure; `continue`, `result`, and `error` are all set to false.

2. The `filter-eval` procedure continues the evaluation of the Filter process represented by its input for a limited number of evaluations. This number is controlled by a global variable `*steps*`, which is initialized prior to calling `filter-eval` and is decremented once every time an atom (i.e. number), variable, or quoted expression is evaluated. When such an evaluation is attempted and `*steps*` is zero, the Filter process is suspended and a closure is returned that represents the state of the Filter process directly before that evaluation would be done. In such a case, the `continue` field of the returned `closure` will contain the state information necessary to continue the process at a later time; the `result` and `error` fields will be false.

3. In the event that a Filter process returns a value and terminates, a `closure` is returned containing the returned value in the `result` field and all other fields set to false.

4. In the event that a Filter cannot be evaluated because of an error in the text of the Filter, a `closure` containing a value of true in the `error` field will be returned. Other helpful information, including error messages and a stack trace, can be interpreted from the contents of the other fields.

The rules enumerated above help to define the Closure structure by providing an invariant standard for the inputs and outputs of the `filter-eval` procedure. Since the `filter-eval` procedure is the only procedure which accesses the internal parts of a Closure, any Closure structure can be expected to behave according to the rules when it is submitted to `filter-eval`.

### Stack and Memory Limits

Along with limiting the number of evaluations performed by a Filter before suspension, similar limitations must be placed on the memory usage and stack depth

achieved by a running Filter. These limitations are not fully implemented in the current version of the meshFind system, but the method for limiting these resources is not difficult to implement.

In order to limit stack depth, the meshFind Filter evaluator flags an error if a continued process has a depth greater than a given limit. Since any sort of infinite loop will quickly use up the available evaluations and be continued, if that loop is causing the stack to build it will result in an error and the termination of the process when the stack limit is reached. The stack limit may also be reached when inappropriate programming styles are used in Filter construction; in fact, some amount of care is required to ensure that the stack does not build up too much.

Limiting memory consumption is quite a bit trickier, and has been left out of the meshFind implementation for the time being. One way to implement it would be to count the number of calls allocating memory, and when that number reaches a limit, perform a reference search to determine how much memory is still referenced. If that count is over a set limit, flag an error. This reference search would start from all environments accessible from the stack and work back to the global environment, checking the values of all symbols. In these limits, it is not as important to have a definite limit which cannot be exceeded as much as it is important to prevent the memory or stack usage from getting out of control.

By now the way in which Filters are interpreted should be somewhat clarified. The details of the Filter interpreter are not really the matter of importance here; the explanation is more a means to explain the more relevant issues of how the Filters are written and how they are expected to be designed, which is the thrust of the next section.

### 4.4.2 What is Idiomatic Filter Design?

The design of a Filter is to a great degree determined by the way in which the Sponsor executes it. For a Filter to work at all, it must present the proper interface so that it and the Sponsor can communicate. A secondary but similarly important factor in Filter construction is the use of *idioms*. Idioms are standard forms or approaches of

which effective Filters are built. We cannot write an inclusive list of these idioms, but the development of sample Filters and toolkits for Filter construction will provide a large base of these hints on which to build. Writing Filters the "wrong" way is likely to produce unfortunate results, because the restrictions on the number of evaluations and on memory usage mean that inefficiently designed Filters will fail to work at all.

A detailed description of the Filter/Sponsor interface, as well as an introduction to Filter design, is to be found in Chapter 5, which covers Sponsors. However, as a concluding remark to this section, we will give a very general overview. A Filter, as passed to the Sponsor in an Unsolved Query, is a single form. That is, it is a list which comprises an evaluatable Scheme expression. That form must define two top-level procedures, one called `init`, and one called `filter`. In essence, the `init` procedure is responsible for initializing the Filter's state information, if any, and the `filter` procedure is responsible for deciding what to do with a given Plan. The `filter` procedure returns its decision in the form of one of several symbols. As a rule, all but the simplest Filters will need to keep some amount of state in the form of global variables which retain their values between calls to `filter`.

We now move on to discuss in greater detail the various message objects used in the meshFind system.

## 4.5 Search Objects

A more thorough description of the Search object hierarchy is shown in Figure 4-1. The inheritance structures shown in the diagram is primarily a convenient method for the internal representation of messages in our implementation. Because of support for the transport of Scheme objects between processes within the MESH system, these objects can be transported in a relatively transparent way. In this discussion we concentrate on a layer in which these objects can be considered to be sent and received, without worrying about the details of the actual wire representation. While such details are fairly straightforward, they are also quite lengthy, making them a less worthwhile topic of discussion. For the most part, then, these details will be left

out of the explanation to make room for a clearer and more thorough coverage of the higher-level concepts.

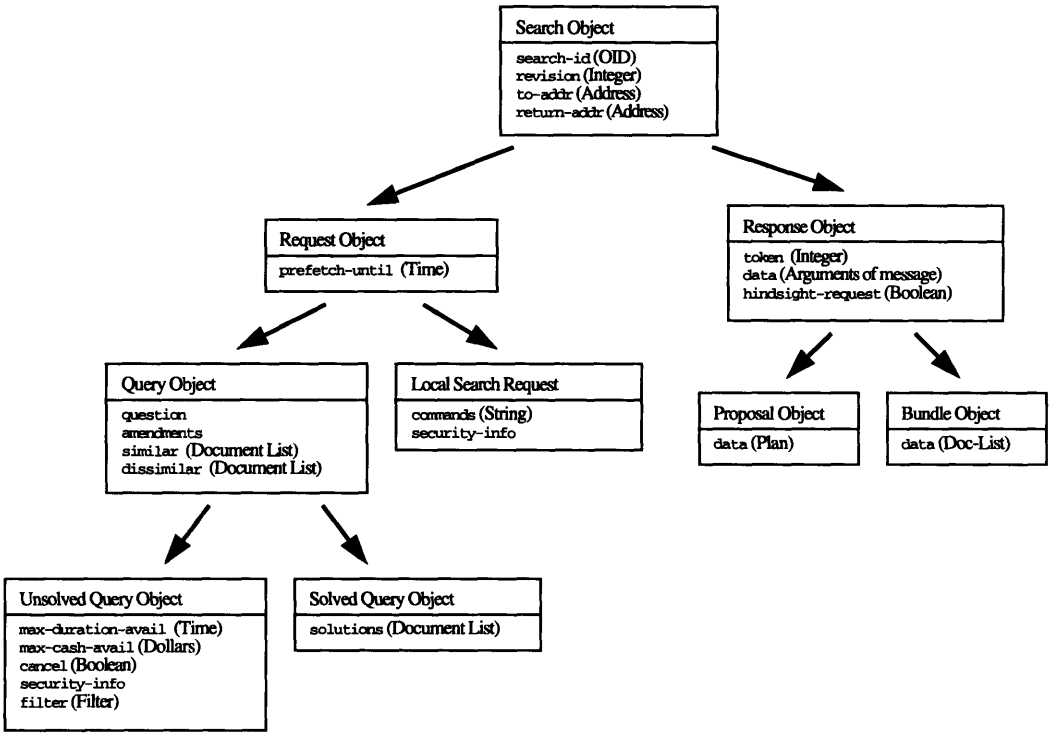


Figure 4-1: The Search object Hierarchy. Arrows indicate the child's inheritance of the parent's fields.

In general, the messages used in the hierarchy are the leaves of the tree; the other parts of the tree show the fields that the leaves have in common, but are not used to represent messages themselves. Consequently, plain Search Objects are never used as messages, although the Search object accessors, for example `search/search-id` or `search/to-addr`, can be used to reference those parts of any object in the tree. In the following sections, we will cover each of the messages in detail. The rest of the hierarchy will be covered along the way, since essentially they are the intersections of the function of the messages, and it is often clearer and much more succinct to explain only once the functionality that is common to many objects. We begin this process by discussing the fields that are common to all Search Objects.

| search | |
|---|---|
| search-id | OID |
| revision | integer |
| to-addr | addr |
| return-addr | addr |
| . | |
| . | |
| . | |

Table 4.6: The Search Object

These common fields, listed in Table 4.6, help to identify and distinguish messages. The `search-id` field identifies a message as relating to a specific search in progress, and hence a specific client. The `revision` field identifies the message as pertaining to a specific revision of the search. Using this field, it is possible to distinguish old and irrelevant messages from new ones. Finally, the `to-addr` and `return-addr` specify the recipient and the sender of the message.

### 4.5.1 Request Objects

The subtree headed by Request objects contains several objects used to communicate requests to a meshFind server. Table 4.7 shows the one field common to all Request objects.

| request | |
|---|---|
| search-id | OID |
| revision | integer |
| to-addr | addr |
| return-addr | addr |
| prefetch-until | time |
| . | |
| . | |
| . | |

Table 4.7: The Request Object

The `prefetch-until` field of a Request object is used to indicate to a server

that the Request should be prefetched. Prefetching is a method of speeding the Referral process. Normally, a Sponsor sends a request to a Referral server, and the Referral sends back a Proposal. Some time later, after the Client's Filter has processed the new Plans, some of the parts of the Plan may be executed. The Sponsor will then send new Request objects out to the servers listed in the Plan, and will then await responses. Prefetching is a way of taking advantage of the delay caused by the Sponsor's involvement in the process. The idea is that when the Referral sends the Proposal back to the Sponsor, it simultaneously sends a prefetch Request to the servers referenced in the Proposal. This prefetch Request asks the servers to service the Request and cache the result, in the hope that the Request will be officially made by the Sponsor in a short amount of time. If the servers receiving prefetch Requests are underutilized and if they have space in their cache, it makes sense for them to do the prefetching.

In a Request object, the `prefetch-until` field can be either false, true, or a time value. When the Client or the Sponsor send a Request, the `prefetch-until` field is set to false, meaning send the result back directly. When a Referral decides to make a prefetch Request to another server, the Request object it sends has the `prefetch-until` field set to either true or a time value. If the field is a time value, that time value is the requested time until which the result will be kept in the cache. The recipient of a request to prefetch is neither bound to do so, nor to do so in any set amount of time. However there may be economic advantages to supporting and implementing prefetching, since the customers perhaps can be served more efficiently.

We will now cover the different types of Request object. They are the Query objects and the Local Search Requests.

### Query Objects

Query Objects are used to represent a question that a client wants answered. There are two types of query object, the Solved and the Unsolved variety. Solved Query objects store a list of solutions in addition to the question information. Unsolved Query Objects store some additional information which specifies how the client

| solved-query | |
|---|---|
| search-id | OID |
| revision | integer |
| to-addr | addr |
| return-addr | addr |
| question | string |
| amendments | list |
| similar | Document List |
| dissimilar | Document List |
| solutions | Document List |

| unsolved-query | |
|---|---|
| search-id | OID |
| revision | integer |
| to-addr | addr |
| return-addr | addr |
| question | string |
| amendments | list |
| similar | Document List |
| dissimilar | Document List |
| max-duration-avail | time |
| max-cash-avail | dollars |
| cancel | boolean |
| security-stuff | security info |
| filter | Filter |

Table 4.8: The Solved and Unsolved Query Objects

wants the search to proceed and some which gives the client direct control over the progress of the search. We will now explain the purpose of the fields which are common to all Query Objects, and leave the explanation of the additional fields to the following two subsections.

The format of the question is specified to be a string. The syntax of the question is effectively determined by the Referral servers to which the query will be sent. At the moment, the current Referral servers expect this string to be English text, without any specific formatting restrictions or special keywords, although words such as "and" and "or" are sometimes recognized. However, future servers may specify more restrictive syntax, or may have more clever processing which makes a deeper analysis of the English text. As the number of Referral servers grows, it is likely that a number of different syntaxes will become commonplace; for the system to work well, however, there will need to be ways of translating between these syntaxes. Hopefully this will be a strong enough influence to encourage the development of mutually convertible syntaxes.

The `amendments` field stores a list of amendments to the question. Since the `question` field always contains the latest version, the `amendments` field stores a record listing the revision during which the amendment was made, and the text that was

replaced. In this way, any previous revision can be built by tracing backwards, undoing each amendment. It is not clear exactly how useful the `amendments` field is, but since it is not too expensive and fairly easy to include, it has been included.

The `similar` field stores a Document List containing documents related to the query. The idea behind this field is that Referral and Index servers might then be built which operate using document-comparison techniques. Such techniques have been studied in [24]. Typically these techniques rely on the use of *word-frequency vectors* to determine document similarity. Word-frequency vectors are vectors in which there is one dimension for every word in a selected vocabulary. For each of these words, the frequency of its occurrence in a given document is recorded to form a word-frequency vector for that document. In order to assess the similarity of two documents, a word-frequency vector is computed for each of the documents and the vectors are compared using standard vector-arithmetic techniques. Despite the simplicity of this technique, it has been used in a number of applications with relative success. Although we have covered the basics here, in order to get this technique to work well there are many issues to be resolved associated with normalization of the vectors and the selection of the proper vocabulary.

The `dissimilar` field stores a Document List containing documents which appeared to be good but did not turn out to be relevant. Not unlike the `similar` field, this field could be used to help determine the relevance of potential matches. Support for determining values for both the `similar` and `dissimilar` fields would be provided by the client software.

Having now covered all the fields which are common to the two types of Query object, we now discuss the fields which are specific to them.

### Unsolved Query Objects

The Unsolved Query object is used by the client not only to communicate the query to be answered, but also to communicate constraints on the search. These constraints control the amount of time and money spent, and, through the use of the Filter, provide for much more subtle control over the process of the search. Refer to Table

4.8 for a listing of the fields and types.

The first field specific to an Unsolved Query object is the `max-duration-avail` field. This field stores the maximum amount of time that the Sponsor is allowed to spend in the search. Similarly, the second field,`max-cash-avail`, stores the maximum amount of money that is available for the search. When the Sponsor forwards an Unsolved Query object to Referral servers, it is usual for these fields to be updated with the latest values (i.e. the starting values less accrued deductions).

The third field is the `cancel` field. This is set to true by the Client in the case that the search is to be immediately stopped. When the Sponsor receives an Unsolved Query with the `cancel` field set to true, it immediately forwards that Query to any Referral servers which are currently processing that Search. It then terminates the Client's session. In only one other instance is the `cancel` field used. In the event that the Sponsor receives a new revision of a Search currently being processed, the Sponsor sends to any Referral servers currently in use copies of the old revision with the `cancel` field set to true. After the out-of-date searches have been terminated, the new ones can be begun.

The fourth field is reserved for security information. Such features as the ability to purchase information and services, the ability to restrict the dissemination of information or the use of services to specific people or groups of people, and many other security-related possibilities might need to send some kind of authentication along with the requests. Issues of privacy may also be an issue, but these are much the same as the privacy issues relating to electronic communication in general. In any case, the `security-stuff` field can probably accommodate most of the security schemes that might become necessary, but since secure protocols in the Internet are currently a major issue of debate, the design of these schemes was for the most part not taken into consideration beyond the inclusion of the `security-stuff` field. For a look the many ideas currently being discussed in the realm of security and authentication in the Web, see the W3C's pages on the topic.[32][31]

The last field stores the Filter which helps the Sponsor determine the path taken by the search. Filters are a key part of the operation of the meshFind system. Filters

and their use is detailed elsewhere; see the section on Filters earlier in this chapter, and also chapter 5 which describes the Sponsor.

### Solved Query Objects

A Solved Query represents a question and answer pair which can be stored, cached, and analyzed. Typically, the conclusion of a search is signalled when the Client sends a Solved Query to the Sponsor. Provisions have been made for these Solved Query objects to be automatically forwarded to any servers that helped in the search and which request to be notified of the answers. Collections of these Solved Queries might be used to better understand new Queries, or to personalize the services provided by a Sponsor. If the Client does not wish to release this information, the search can also be terminated by sending an Unsolved Query with the `cancel` field set to true.

The Solved Query object adds the `solutions` field to the standard Query object (refer to Table 4.8 for a list of fields and associated data types.) This field stores in a Document List any documents which the Client found to be acceptable answers to the Query.

### Local Search Requests

| local-search-request | |
|---|---|
| search-id | OID |
| revision | integer |
| to-addr | addr |
| return-addr | addr |
| commands | string |
| security-stuff | security info |

Table 4.9: The Local Search Request

The Local Search Request, or LSR, is used to perform a search on a specific index. The contents of an LSR are usually determined by a Referral server and returned to a Sponsor as part of a plan. The job of the Referral server in such a case is to translate the Unsolved Query into a selection of requests which will answer the Query. The

Sponsor then packages each request in an LSR and sends it off to the appropriate Index server. In order to perform the local search, an Index server needs two pieces of information, and these are stored in the two additional fields of an LSR.

First, the Index server needs to know what query to send to the underlying search engine. This is given in the form of a string and is stored in the `commands` field of an LSR. These query strings are of different formats depending on the architecture of the underlying search engine. While each format is slightly different, for the most part the differences are not drastic; most of these formats are variations on the theme of lists of keywords joined by conjunctions.

The other field which is included in an LSR is the `security-stuff` field. This field parallels the `security-stuff` field contained in the Unsolved Query object. Once again, the details of the design of the various security schemes which may be necessary are beyond the scope of this work. This field does provide enough design space to implement many of these schemes should that become desirable.

### 4.5.2 Response Objects

The Response object is an exception to the rule that all of the messages used in the meshFind system are leaves of the Search object hierarchy. In this case, the Response object itself is used to transport simple messages, while its child objects, Proposals and Bundles, are used to transport more complex responses.

#### Simple Messages

Simple messages are messages which convey only an acknowledgement or flag. Essentially, they are stored in two parts, the `token` field and the `data` field. The `token` is an integer which indicates the essential form of the message. A list of some of the possible token values is shown in Table 4.11. The `data` field contains a list of arguments which fit into the form specified by the `token`. For example, if a Sponsor rejects a Client's Query, the Sponsor sends a simple message containing a token value of 5, and an argument list containing a string explaining why the Query was rejected. There is no real need to send an argument list rather than a single

| response | |
|---|---|
| search-id | OID |
| revision | integer |
| to-addr | addr |
| return-addr | addr |
| token | integer |
| data | list |
| hindsight-request | boolean |

Table 4.10: The Response Object

string; however since our implementation is written in Scheme the argument list was chosen for simplicity of implementation.

| Token Symbol | Value | Args |
|---|---|---|
| sponsor-accept-query-token | 4 | – |
| sponsor-reject-query-token | 5 | reason |
| sponsor-termination-token | 7 | time, money, evals used |
| sponsor-termination-threat | 8 | time until termination |
| insufficient-funds-token | 9 | cost |
| filter-error-token | 10 | error message string |
| renewal-from-client | 11 | time, money, evals |

Table 4.11: Partial Listing of Token Values

The third field specific to the Response object is the `hindsight-request` field. Whenever a Response is sent back to a Sponsor, this flag can be set, indicating that the Sponsor should send back a copy of the Solved Query object if and when it is determined.

### Proposal and Bundle Objects

Table 4.12 shows the structure of the Proposal and Bundle objects. These objects are in fact implemented on top of the simple message structure; in each the `head` field overlies the `data` field, and the `token` field has a reserved value which indicates one of four types of message. These four types are listed in Table 4.13 below.

In order to distinguish between Proposals and Bundles being sent "away" from

| proposal | |
|---|---|
| search-id | OID |
| revision | integer |
| to-addr | addr |
| return-addr | addr |
| head | Plan |
| hindsight-request | boolean |

| bundle | |
|---|---|
| search-id | OID |
| revision | integer |
| to-addr | addr |
| return-addr | addr |
| head | Document List |
| hindsight-request | boolean |

Table 4.12: The Proposal and Bundle Objects

| Token Symbol | Value | Args |
|---|---|---|
| proposal-to-client-token | 0 | Plan |
| proposal-from-client-token | 1 | Plan |
| bundle-to-client-token | 2 | Document List |
| bundle-from-client-token | 3 | Document List |

Table 4.13: Token Values Reserved for Proposals and Bundles

the Client (termed *sponsor-directed* Proposals and Bundles) and those being sent "towards" the Client (termed *client-directed*), two different token values can be used to describe a given Proposal or Bundle. Essentially, objects being sent to the Sponsor from a Referral server and objects sent to the Client from the Sponsor are considered to be client-directed and are marked with either the `proposal-to-client-token` or `bundle-to-client-token`, depending on the type of data being sent. Proposals and Bundles sent from the Client to the Sponsor are considered to be sponsor-directed and are consequently marked with the `proposal-from-client-token` and the `bundle-from-client-token` respectively.

## 4.6 Conclusion

We have now covered in detail all of the data structures that are involved with inter-server communication. In the next chapters, we will cover the design of the servers themselves in more detail. Each server will be covered individually, giving a much more specific description of its behavior and of its internal data structures.

# Chapter 5

# The Sponsor

As we discussed in Chapter 3, the part the Sponsor plays in the meshFind system is central to its operation. In this chapter, we will examine the Sponsor server in greater detail, starting with an overview of its function and its interaction with the other parts of the system. Then, we will delve into the details of our Sponsor implementation, including a discussion of several internal data structures. Finally, we will cover the interfaces and protocols once more, in a more terse and detailed form.

## 5.1 Functional Overview

The Sponsor provides a proxy layer between the Client and a network of Referral and Index services. By doing the work which requires high connectivity, the Sponsor can make a wide range of services available to Clients who cannot afford a high-bandwidth network connection. By taking care of much of the book-keeping, the Sponsor, together with the network of Referral servers, help the Client sort through a complex and confusing array of services.

The Sponsor is designed to provide this service to many Clients concurrently. Although it is by no means a necessity, by servicing a community of Clients the Sponsor gains the opportunity to become a layer through which Clients can share the benefits of knowledge pooled from solving the queries submitted by the group. The details of implementing an effective communal knowledge-base are a thesis in

themselves, but the meshFind system was designed to support the development of such meta-tools.

### 5.1.1 Initiation of a Search

Before a search can be initiated, the Client must create an initial Unsolved Query object. As is specified in Chapter 4, the Unsolved Query object, as well as all other Search objects, contains a field called `search-id`. When the Client creates an Unsolved Query object, a new unique OID is stored in its `search-id` field. This OID will be used to identify all messages related to the proposed search. The other fields of the Unsolved Query allow the Client to specify the question to be answered, as well as certain details about how to manage the search. These fields are described in detail in Chapter 4.

A search is initiated when a Client sends the initial Unsolved Query object to a Sponsor. When the Sponsor receives the Query, it must decide whether or not to accept it. There may be many factors influencing this decision. For example, the current load on the Sponsor may make it impossible to provide adequate service, the format of the question may be inappropriate, or the topic of the question may be one for which the Sponsor is poorly prepared. Whatever the reasons, the Sponsor makes a decision and sends an acknowledgement back to the Client which either acknowledges acceptance of the Query or gives a reason for rejection.

This exchange is a fairly typical example of the communication between the Sponsor and the Client. The connections between the Client and the Sponsor may be implemented on top of a variety of underlying protocols. For example, one way to implement this would be to have the Client open a TCP connection to the Sponsor, send the Unsolved Query, and wait for the acknowledgement. While this technique is robust it is also expensive, since the Sponsor must keep the connection open while making the decision, and in the event of excessive load this might force the Sponsor to refuse many of the attempted connections.

A less robust implementation (or at least one which requires more complex error handling in order to achieve equivalent robustness) might be built on the UDP pro-

tocols. Using UDP the Sponsor and the Client send the parts of the protocol to each other as separate messages. That is, the Client sends the Query to the Sponsor, but rather than establishing a connection in the TCP sense, the packets are sent more or less blindly – the Client will not know for sure whether they were received. On the other hand, there is much less overhead involved with the UDP protocols, and the connection will not need to be kept open while the Sponsor makes its decision. The drawback is that the packets are not guaranteed to get to the Sponsor, and the acknowledgement is not guaranteed to make it back to the Client. This means that both sides have to be prepared to resend and must be prepared to receive duplicate packets and packets out of order.

This error handling, while it is eminently feasible, makes the protocols significantly more complex. As a result, for our implementation we chose to use reliable inter-process communication as the basis of communication. This way, the task of error handling was greatly simplified, and the implementation could be converted to a TCP-based protocol without much difficulty. Using inter-process communication also has the positive side-effect of making it easy to simulate the system using one machine, which was desirable when we wanted to simulate a system with many servers. Although it would not be difficult to replace the inter-process communication with TCP-based reliable datagram transmission, given favorable network conditions this implementation would probably be slower than a UDP-based system. It would be an interesting extension to this work to perform some simulations of meshFind, implemented using the various protocols, and to compare the relative efficiency of the different implementations. With a system such as this one, it is difficult to predict what the important factors are, and only through simulation can these factors be discovered and properly assessed. One factor which would become important is congestion control. TCP builds in some clever techniques for controlling network congestion, and in order to construct a working system running over UDP, similar features must be implemented in the module which handles resending and acknowledgement.

### 5.1.2 Choosing and Implementing an Initial Plan

After the initial Query is sent and a positive acknowledgement from the Sponsor has been received, the Client is considered to have initiated a search. The Sponsor now registers the Client in its internal database and goes to work on the Client's behalf. In order to start the search, the Sponsor forms an initial Plan for solving the Query. This Plan typically contains references to one or more Referral servers which it regularly uses as starting points. These are most likely the "general" type Referral servers that were briefly described in Chapter 3. This initial Plan is then passed through the Filter contained in the Query to decide which element to execute first. When the Filter selects an element of the Plan to execute, the Unsolved Query object will be forwarded to the Referral server referenced by that element of the Plan. Again, the Unsolved Query object will be sent using some kind of message-passing protocol; in our implementation inter-process communication is used.

When a Query object is forwarded in this way, typically certain fields of the Query are modified to reflect its new context. Clearly the `to-addr` and `return-addr` fields must be updated, placing the Address of the Referral server in the `to-addr` field and the Address of the Sponsor in the `return-addr` field. The `max-cash-avail` and `max-time-avail` fields are also updated to reflect the current situation; as money is spent and time passes these fields will need to diminish appropriately. The new value for the `return-addr` field must also contain an appropriate value in the `attn-oid` field of the Address. When the Referral server sends back a response to the Sponsor, this OID will identify the element of the Plan to which that response relates. This way, the multitude of responses coming back to the Sponsor which relate to a given search (i.e. all having the same value in the `search-id` field) can be quickly matched to the `plan-elt` which spawned them.

This is important because it allows the Sponsor to keep track of which requests have been completed and which have not. Requests that are not fulfilled in a timely way may need to be resent, especially if the underlying protocol is unreliable. Even with a reliable protocol, it is possible that a Referral server might go down or somehow lose a request, so a resending protocol is required regardless of the underlying

message-passing protocol. As we saw in Chapter 4, the `typical-duration` field of the `server-info` block relating to a server gives an estimate of the length of time the server needs to process a request. In our implementation, this estimated response time is used to calculate a time after which the Sponsor must assume that something went wrong. In the event that the `typical-delay` field is not filled in, a fixed length of time is used. If a response has not been received after the specified time interval has elapsed, either the request is sent again and the process repeats, or the Sponsor decides to terminate this particular thread.

### 5.1.3 Using Filters to Direct the Search

The Sponsor continues the search process by running the Client's Filter on any available Plans. Because many Clients are being served concurrently, the Filters are run on a time-shared basis. Each Filter is allowed to run for a fixed number of evaluations before being suspended so that other processes may run. The Plans awaiting processing by the Filter are stored in a queue containing structures of type `plan-process`. The `plan-process` structure is detailed in Table 5.1, but essentially it contains a series-type Plan and an integer representing that Plan's priority in the queue. The queue is not always kept in order of priority; for reasons which will become clear later it is sometimes out of order, but it is re-ordered at fairly regular intervals. Taken from a wide perspective, the Sponsor runs the Filter on each Plan in the queue, starting from the front and working back. By giving responses to the Plans provided, the Filter can control its progress through the queue.

| Structure `plan-process` | |
|---|---|
| `the-plan` | Series-type Plan |
| `priority` | integer |

Table 5.1: The `plan-process` structure

In order for a Filter to work it must adhere to the interface specifications imposed by the Sponsor. This means that the Filter must define a procedure called `filter`

which takes three arguments: a `plan-process` which contains a Plan to filter and the current priority assigned to that Plan, a duration argument which gives the time left in the search, and third argument telling how much money is available to spend.

Given these arguments, the `filter` procedure must determine what to do with the given Plan. After making this determination, the `filter` procedure must communicate the decision back to the meshFind system by returning one of a set of acceptable symbols as its result. As we have seen, the input Plan is provided in the `plan-process` structure. The Plan stored in the `the-plan` field is always a series-type Plan containing at least one element in its list. Although there is nothing preventing a Filter from considering the latter elements in the list, the responses that it can make refer only to the topmost item in the list. These responses are detailed in Table 5.2.

| Filter Responses | |
|---|---|
| `execute` | Execute topmost item in current Plan |
| `punt` | Throw out topmost item in current Plan |
| `push` | Push topmost item in current Plan down one step |
| `rewind` | Sort Plan queue and start from the beginning |
| Positive Integer X | Set current Plan's priority to X, move to next Plan |
| Negative Integer X | Sleep for X Ticks, rewind |
| Other Values | Sleep until next turn, rewind |

Table 5.2: Filter responses and their associated meanings.

### Requirements Placed on Plans

These responses may at first seem to be a strange collection, but taken in the context of the way Plans are formed and collected, they make sense. In the course of a search, a Client may be presented with a number of competing Plans, collected from a variety of sources. In our discussion of Plans in Chapter 4, we saw that Plans can be serial or parallel combinations of subplans, eventually terminating in `plan-elt` structures which contain references to specific Referral and Index servers. However, in order for a relatively simple Filter to have any chance of interpreting them, this recursive structure must be formatted in some kind of standard way. In our implementation,

we chose a standard which provides a basis for interpretation without being overly restrictive in its format. We specify that any Plan returned by a Referral server must be a Series-type plan, with at least one element. The list contained in the Plan must be sorted in order of decreasing relevance, judged to whatever accuracy the Referral server can provide. Each item in this Plan may be an element or a subplan, and any subplans may be either parallel or series-type Plans; however, any series-type subplans must in turn satisfy the ordering constraint specified for the outermost Plan.

By ordering the Plans in this way, the queue of Plans has some interesting properties. To a certain degree, the topmost item in each of the Plans in the queue is at the same level of relevance. By setting the priority specifier attached to each Plan in the queue, any differences in relevance may be offset by varying the position of Plans in the queue. A typical Filter process would then run through the Plans in the queue, examining the topmost element, doing something with it, setting the priority of the remainder of the Plan, and moving on to the next item in the queue. Referring to Table 5.2, we can see the responses a Filter must make to complete this process. When the `filter` procedure is called with a given Plan and it examines the topmost item in the series, it may do something with that item by responding with either `punt`, `push`, or `execute`, or it may assign a priority to the current Plan and go on to the next Plan in the queue by returning a positive integer (or it may start from the first Plan in the queue by responding otherwise).

### Filter Responses

The first three responses all cause modifications to be made to the current Plan. In these cases, if the modified Plan is non-empty it will be passed to the next call of `filter`. The `push` response exchanges the first and second items in the current Plan. If only one item exists in the current Plan, an error condition is flagged. The `push` response is useful for situations where a Filter needs to examine the top two items without deciding on the top item first.

The `punt` response causes the topmost item to be thrown away. Items that are thrown away are not retrievable, although there is nothing preventing a Filter from

saving a copy of an item before throwing it away. The next call to the Filter will be given the remainder of the Plan as its input, unless the top item was the only element in the Plan, in which case the next Plan in the queue would be submitted.

The `execute` response is more complex. When `execute` is returned, a variety of actions can occur depending on the type of item at the top of the current Plan. If that item is a `plan-elt` structure, the Sponsor will send out the appropriate messages to cause that part of the Plan to be executed; if the element refers to a Referral server, the Sponsor will forward the Client's Query to that server, while if the element refers to an Index server, the Sponsor will send a Local Search Request. See Section 5.1.4 for more detail on this message passing. If the topmost item is a serial-type Plan, that Plan will be inserted into the queue of Plans directly after the current Plan. This way, when the Filter eventually moves on to the next Plan in the queue, the newly executed subplan will be next in line to be processed. Similarly, if the topmost item is a parallel-type Plan, each of its parallel items will be inserted into the queue as a separate series-type Plan, again directly after the current Plan. In all cases the topmost item is removed from the current Plan after it is executed.

The other responses cause the next call of `filter` to be given a different Plan to work on. By responding with a positive integer, the current Plan's priority will be set to that number and the next Plan in the queue will be passed to `filter` the next time it is called. Filters which have a good scheme for assigning priority have the potential to be very effective – but even if a Filter never wants to change the order of the Plans this response is necessary merely in order to step through the queue.

When a Filter responds with `rewind` or a negative integer, the queue will be reordered and the first Plan in the queue will be passed to the next call to `filter`. When `rewind` is returned, `filter` is called immediately with the newly ordered queue.

By returning a negative integer the Filter can indicate that it is done processing for the moment and will await further developments. This is especially useful if the Sponsor charges money for Filter evaluations. The Filter will be reawakened when the specified time is up or when a new Plan comes back from a Referral server, whichever is sooner. When new Plans arrive they are always inserted so that they are the next

Plans sent to the Filter.

### Housekeeping Headaches: Errors and Evaluation-Counting

The last issues to cover in this section on Filters are practical ones. The first is the problem of handling errors in a Filter. In the event of an error, the evaluator will return an error message to the Sponsor program. This message is sent back to the Client in a simple message with a token value of `filter-error`. Rather than trying to recover from where the error occurred, the state of the Filter is returned to the state directly after the completion of the last successful run, and the Filter is run on the next plan in the queue. Hopefully this will avoid an immediate repeat of the error and will give the Filter a chance to operate on the other Plans in the queue. If the Filter returns an error several times consecutively, the Filter is re-initialized by evaluating the original Filter in a fresh environment. If this still yields an error, the search is terminated.

The second issue is that of charging extra for inefficiency in Filter design. While the number of evaluations used is strongly coupled to the amount of time spent using the server (and hence can usually be considered to be a fixed cost per unit time), certainly Filters can be designed in more or less efficient ways. It may be the case that in order to encourage efficient use of the available resources, a charge should be applied to every evaluation made in excess of an initial allowance which is proportional to the amount of server time allotted. This charge would be drawn against a separate fund to which money can be added with the permission of the Client. This authorization would be made through the use of a simple message of the type `renewal-from-client`, in a process which is explained in Section 5.1.6. As with most cost-related issues, this facility was not implemented in our prototype system, but room was left in the design to accommodate such a scheme.

### Simple Examples of Filters

The simplest possible Filter is probably the following:

```
(begin
 (define init
   (lambda (time-left cash-left)
     #f))
 (define filter
   (lambda (plan-process time-left cash-left)
     'execute)))
```

Of course, that is not a very useful Filter, since it passes all Plans submitted to it. A more sophisticated example of a Filter is described in Appendix A. Understanding how it works will require a more in depth understanding of Scheme

### 5.1.4 Sponsor/Referral Communication

As the Client's Filter is run on the Plans constructed for the Client, the Filter will occasionally choose to execute some elements of the Plans. When that happens, the Sponsor must send the appropriate message to the server contained in the Plan element. We described much of the process of forwarding the Query to a Referral server in Section 5.1.2.

Whenever a Client is presented with a Plan, the Sponsor "prunes" the Plan before handing it to the Client's Filter. This pruning process involves checking each element of the plan to see if it has appeared before in some other Plan given to this Client. Any elements that already exist in some other Plan are dropped from the new Plan, and the newly suggested elements are entered into a master list of elements suggested to this Client. When an element is entered into the master list it is assigned an OID. If at a later time an element is actually executed, that OID will be used to identify the response coming back from the Referral or Index server. This is done by storing the element's OID in the `attn-oid` field of the return Address when the request is sent out. This way when the response comes back, the `attn-oid` field of the "To" Address will match exactly the OID of the element which spawned the request.

In order to keep track of which requests have been sent out, a list is kept called `pending-requests`. This list keeps track of any requests that have been sent and are awaiting responses, along with the time each request was made. Whenever a

response relating to the Client's search (i.e. with the right Search ID) is received by the Sponsor, the value in the `attn-oid` field of the "To" Address is matched against the `pending-requests` list. If no match is found, the response is thrown away. If a match is found, then the matching element is moved to another list containing places which have already been searched, and the response is processed.

When the Filter chooses to execute a part of a Plan, the process takes one of two different forms depending on whether the element is referring to an Index server or to a Referral server. If the element describes a Referral server, a copy of the Client's Unsolved Query object is updated to reflect the latest limitations on money and time, and new Address information is filled in. The Address of the Referral server is placed in the `to-addr` field, and the Address of the Sponsor is placed in the `return-addr` field. As we mentioned before, the OID associated with the element being executed is placed in the `attn-oid` field of the return Address. This Query is then sent to the Referral server, and the element being executed is placed in the `pending-requests` list. When a response comes back with the matching OID, we expect that response to be a Proposal object containing a valid Plan. This Plan is first checked for proper syntax, next pruned, and finally placed in the Client's queue of Plans in a position where it will be the next Plan processed.

If the element refers to an Index server, a Local Search Request must be constructed from the information given in the element. Similar to the case above, the "To" and return Addresses are set with the Index server's and the Sponsor's Addresses respectively. Again, the `attn-oid` field is set to match the OID of the element so that the response can be matched back up with the element that spawned it. The Local Search Request is then sent to the proper Index server, and the element is placed in the `pending-requests` list. When a matching response arrives at the Sponsor, we expect the response to be a Bundle object of the client-directed variety containing a list of Documents for the Client. This Bundle object is then forwarded to the Client by simply modifying the "To" and return Addresses and sending it back out.

The `pending-requests` list must be checked occasionally for requests that have timed out. When a request is not answered in a specified interval of time, the request

must be resent to the appropriate server. This resending will occur intermittently until either a response is received, the Client stops the search, or the time allotted for the search elapses and the search is terminated by the Sponsor. The period of this resubmission may be a fixed duration or may depend on the known or typical behavior of servers on an individual basis.

### 5.1.5 Client/Sponsor Communication

The communication between the Client and the Sponsor was from the beginning designed to be very light-weight. So far the only communication we have discussed has been the initiation of the search in the very beginning and the forwarding of Bundle objects returned from Index servers. However, there are a number of other cases in which the Client communicates with the Sponsor.

At any time during the Client's session with the Sponsor, the Client may update the Query (perhaps based on new insight or on information from result documents). First, the Client must construct a new Unsolved Query object with updated question information, a larger number in the `revision` field, and the same `search-id` as the original Query. Then the updated Query can be sent to the Sponsor. Upon receipt, the Sponsor will terminate the old search (see Section 5.1.6) and immediately start the search over from the beginning in an effort to solve the new Query.

As we saw in the previous section, the results of the search of an Index server are automatically forwarded to the Client. When the Client receives a client-directed Bundle object, the `document` structures contained within it may not all contain the complete text of the documents. Many Index servers send only the headlines and OIDs associated with the documents, requiring the Client to select those documents for which the full text is required and to request them separately. These requests are made by creating a sponsor-directed Bundle containing the `document` structures relating to the requested documents and by then sending that Bundle back to the Sponsor. When the Sponsor receives this Bundle, it will formulate a Local Search Request which lists the desired documents in the `commands` field and send that LSR to the Index server which holds the documents. This can be done using the usual

process used to handle Local Search Requests, including placing an entry in the `pending-requests` list. Similar to other LSRs, when a client-directed Bundle is received by the Sponsor in response to this LSR, it is immediately forwarded to the Client. This response should contain all of the documents requested by the Client.

There are many issues surrounding the design of the Client-side software which impinge upon the current system of document retrieval. For example, might it be useful to retrieve only part of a document, say a only a single chapter of a whole book? If some of the data objects being retrieved are pictures or music or other types of "document", might there be even more complex ways to describe which parts to retrieve? Clearly the design of the Client software, the types of data available for retrieval, and the design of the retrieval protocol are all interrelated. Given that presently most of the documents available are of manageable length and given that our prototype system is restricted to text data, we chose to implement a relatively simple document retrieval protocol.

### 5.1.6 Termination of a Search

There are two ways in which a search may be terminated by the Client. If the Client sends the Sponsor an Unsolved Query with the `cancel` flag set to true, the Sponsor will cancel the search. When the Sponsor cancels the search, any pending requests are cancelled by sending an Unsolved Query with the `cancel` flag true to the Referral servers currently processing the Client's request. Then the Sponsor removes the Client from its internal structures and schedules. Finally, the Sponsor sends a simple message back to the Client, acknowledging the termination of the search. Any messages received by the Sponsor that are related to a terminated search may be either thrown out or forwarded to the Client at the option of the Sponsor.

If the Client has found a satisfactory answer to the Query, the Client may instead choose to terminate the search by sending a Solved Query object to the Sponsor. When the Sponsor receives a Solved Query, it cancels the search in the same way that it does when it receives an Unsolved Query with the `cancel` field set; however in addition it also passes the solutions on to any of the servers which were involved in the

search and which requested to be given solutions. These requests are made by setting the `hindsight-request` flag of a Response object to true. It is the Sponsor's job to keep a list of those servers which request notification and to send out the solutions if possible.

Under certain conditions the Sponsor will terminate the search without the Client's request. If the time allotted to the search runs out, or if the evaluations allotted to a Client's Filter run out, the search will be terminated automatically. In all cases, the termination is signalled by a simple message with `sponsor-termination-token` as its token value. Since the termination means that in order to continue the search would have to begin again, it is advisable to give the Client warning of impending termination. This can be done with a simple message with a token value of `sponsor-termination-threat` which gives a time at which the termination will occur. The Client can postpone termination by sending a simple message to the Sponsor of type `renewal-from-client`, which extends the time, money, and evaluation limits that had been previously set.

This concludes the overview of the design of the Sponsor server. In the next section, we will discuss in detail the internal data structures and modules which support our implementation of the Sponsor. Then, a final section will give a much more specific description of the protocols and processes that form the Sponsor server, along with a description of our implementation, its modularity, and possible extensions.

## 5.2 The Implementation of the Sponsor

Our implementation of the Sponsor server is broken into several modules, each of which handles its own collection of tasks and manages any associated data. In this section, we will describe each of these modules. Our purpose here is not to give a complete description of the code. However, in our effort to give a thorough explanation of the protocols used in our implementation, we need to describe in some detail the structure of the code. This section will provide this by describing the interfaces and data structures used in our implementation. The final section in this chapter

will then build upon this by giving a precise specification of the protocols, in doing so making reference to the procedures and structures described here.

We will begin this explanation at the top level, with an explanation of the `sponsor` data structure. However, we will hold off on the description of the procedures in the Sponsor module until after we have described the other modules. This ordering is necessary because, while the Sponsor module's procedures rely on procedures in the other modules, the other modules rely on the data stored in the `sponsor` structure.

| Structure `sponsor` | |
|---|---|
| `sponsor-id` | OID |
| `address` | Address |
| `name` | string |
| `client-files` | hashtable of `client-file` structures |
| `schedule` | `schedule` structure |
| `reject-client-hook` | procedure |
| `formulate-initial-strategy-hook` | procedure |

Table 5.3: The `sponsor` structure

The `sponsor` structure holds all the data needed to run our implementation of a Sponsor. In constructing our prototype, we tried to separate the required parts of a Sponsor implementation from the parts which would vary among different Sponsors. These required parts form a kernel of code which relies on the hooks stored in the `sponsor` structure to do the implementation-specific work. This arrangement made it convenient for us to simulate multiple Sponsors on the same machine, since the kernel code could be shared; to run a given Sponsor we needed only to pass the appropriate `sponsor` structure to a procedure in the kernel.

The Sponsor represented by a `sponsor` structure is identified by the data in the first three fields. The `sponsor-id` field stores the OID associated with the Sponsor server. The `name` field stores a string naming the server. This is mainly a convenience and has no significance with respect to the protocol. The `address` field contains an Address which can be used to send messages to the server. This Address contains information about the server in the `server-info` field, and contains the server's OID

in the `server-oid` field.

The current state of the Sponsor is stored in the next two fields. The data relating to the Clients being served by the Sponsor is kept in the `client-files` field. Associated with each search being performed is a `client-file` structure which stores information about the state of the Client's search. The `client-files` field of a `sponsor` structure stores a `client-file` record for each of the Sponsor's current Clients in a hashtable keyed by Search ID, allowing convenient and efficient retrieval of Client information. The `schedule` field of the `sponsor` structure is used to organize and plan the processing done by a Sponsor server. The field contains a `schedule` structure, which packages a set of independent queues into one object. Schedules are maintained and manipulated using the procedures in the Schedule Module, which is described in the next section.

The last group of fields in the `sponsor` structure contain hooks. These hooks are procedures that are used to pass control out of the meshFind kernel, allowing code in the external implementation to run. This makes it possible for Sponsor implementations to be written which use the kernel code to implement the standard protocols but which also have their own private strategies and which implement their own extensions to the protocol. For example, when a prospective Client sends an initial Query to the Sponsor, the `reject-client-hook` is called with two arguments. The first is the `sponsor` structure itself, and the second is the initial Query object sent by the Client. Based on this information, the hook procedure must decide whether to accept or reject the Client, and must return either false, meaning that the Client is to be accepted, or a string containing a reason for rejecting the Client.

The Sponsor kernel roughly forms five modules. At the top level of control there is a main Sponsor module, which handles all of the communication with the outside world and implements the protocol described herein. The other modules take care of smaller tasks which center around the data structures they maintain. The `schedule` structures mentioned before are maintained by the procedures in the Schedules module. Similarly, there is a module Plans which contains procedures for manipulating Plan structures. A module Client-Files is used to maintain the database of Clients

and also to provide many of the services to them, including that of running their Filters. Finally, the Filter-Sim module contains the code for the Filter evaluator. Each module in turn will be discussed in the following sections.

### 5.2.1 The Schedule Module

| Structure `schedule` | |
|---|---|
| `size` | number of queues in Schedule |
| `priority-frames` | stack of "frames" |
| `heads` | vector of the heads of the queues of entries |
| `holds` | vector of lists of held entries |

| Structure `entry` | |
|---|---|
| `time` | the time at which to perform the action |
| `tag` | symbol indicating the action to perform |
| `args` | list of arguments containing any necessary data |

Table 5.4: Structures maintained by the Schedule module

As was briefly mentioned before, a Schedule can contain several queues. The overall idea behind the Schedule structure is that the Sponsor needs to do many different types of processing, and external conditions may mean that one type will need to be postponed while other processing gets done. For example, if a large backlog builds up in one of the queues, it might be a good idea to put a higher priority on emptying that queue. On the other hand, if that backlog exists because no progress can be made, for example if the network connection is down for a few minutes and there are many messages waiting to be sent, priority should be placed on servicing the other queues, despite the apparent backlog. These issues of priority and scheduling are difficult, and the Schedules module is designed to provide some useful tools for solving some of these problems.

At any given time, a queue can store a number of entries, arranged in some order. These entries are structures of type `entry`, and as such have a `time` field which is usually used to order the queue. In general, when an entry is entered into the queue it is inserted in a position which will maintain the desired time ordering.

Entries are removed and processed in order from the front of a queue. Entries may also be removed from the queue by "filtering" the queue. This refers to the process of examining each entry in the queue and deciding whether to keep it or to throw it away. All of these features are fairly standard queue-processing features. The Schedule module also contains procedures for determining which queue should be processed next, taking into account indicators of priority.

Schedule queues have one somewhat unusual feature: holding. When an entry is about to be processed, there is the option to "hold" the entry rather than remove it for processing. When an entry is held, it is moved to a separate queue of held entries, making the next element in the queue available for processing. At a later date, the entries stacked up in the hold queue can be reinstated in their original queue, formed into a new queue, or just thrown out. Hold queues are neither conceptually substantial nor hard to implement, but they do come in handy sometimes. For example, messages to a server that is temporarily inaccessible can be conveniently "put on hold" and resubmitted at a later time. There is a facility for determining the *"hold" ratio*, which is the ratio of held entries to regular entries for a given queue. Depending on the rationale governing the choice to hold an entry, this ratio might be useful for deciding when to process the held items, or perhaps for determining the priority to assign to the queues.

Each queue in a Schedule is made up of three parts. These parts are stored in three separate vectors of length `size`. Each queue in the Schedule has an index which is used to reference its parts in the three vectors. The head of a queue comes from the vector stored in the `heads` field of a Schedule. The head is a list of entries, usually in order of increasing `time` value. The first element in the list is the next entry to be processed. The hold queue associated with a given queue is an element of the vector stored in the `holds` field of a Schedule. The hold queue is a list of entries in which the first item in the list is the last entry held.

The `priority-frames` field of a Schedule contains a list of "frames" in which the first item in the list is the current frame. A frame is a vector of length `size` in which the elements are numbers indicating the relative priority of the queue of

corresponding index. When a Schedule is created, the `priority-frames` field is initialized as a list containing a frame in which all the values are set to 1, thus initially assigning equal weight to each of the queues. The list of frames implements a stack, allowing temporary adjustments in priority to be made by pushing a new frame on the stack. The previous conditions can be reinstated by popping the stack. Procedures for pushing new frames and popping old frames have been provided in the interface to the module. The weights in the current frame affect the ordering of the Schedule by influencing the decision of the procedure which chooses the next queue to service.

| schedule/... | Arguments expected; return type |
|---|---|
| create | size:integer; schedule |
| add | self:schedule, index:integer, item:entry; – |
| add-to-end | self:schedule, index:integer, item:entry; – |
| next | self:schedule, time:integer; entry |
| hold-item | self:schedule, index:integer, item:entry; – |
| reinstate-held-items | self:schedule; – |
| push-priorities | self:schedule, frame:vector; – |
| pop-priorities | self:schedule; vector |
| time-of-next | self:schedule; integer |
| hold-ratio | self:schedule, index:integer; ratio |

Table 5.5: Procedures Implemented in the Schedule Module

The procedures implemented in the Schedule module are for the most part self-explanatory; we will conclude this section by briefly describing each. Note that the names of the procedures all begin with the "`schedule/`" prefix, which is left off for brevity. The `create` procedure returns a new Schedule of the given size. The `add` procedure inserts, in increasing time order, a new entry to the queue of `self` specified by `index`. The `add-to-end` procedure adds the new entry to the end of the specified queue, and adjusts the `time` field of `item` if necessary. If the `time` field of `item` is less than the `time` field of the last entry in the queue, the `time` field of `item` is increased to match.

The `next` procedure determines the next queue to process taking into account

the current priority frame, removes the first element in the queue, and returns it. The `hold-item` procedure places a given entry into the specified hold queue. The `reinstate-held-items` procedure moves all held items in the given Schedule back into their original queues. In our implementation we did not encounter a need to reinstate only selected held items, but such a procedure would not be difficult to implement should the need arise. The procedures `push-priorities` and `pop-priorities` provide access to the stack of frames. The `time-of-next` procedure returns the earliest time at which a queued entry will come due. The `hold-ratio` procedure returns the ratio of held entries to queued entries for a given queue.

### 5.2.2 The Plans Module

In this section we will build upon the foundation laid by the discussion of the Plan structure in Section 4.3. The discussion of the structure of Plans will not be reiterated here; instead we will describe the procedures which provide support for the construction and interpretation of Plans. Because our goal is to describe the implementation of the Sponsor, the main emphasis is placed on interpretation of Plans, since, except for the construction of an initial Plan, it is the job of a Referral server to construct plans.

| Procedure | Arguments expected; return type |
|---|---|
| `make-plan` | `parallel?`:boolean, `time-allocated`:integer, `items`:list; `plan` |
| `make-plan-elt` | parts of `plan-elt`...; `plan-elt` |
| `plan/encapsulate` | `self`:Plan; Plan |

Table 5.6: Constructors for Plans

Table 5.6 lists the constructor procedures required to construct a Plan object. The parameters to the constructors `make-plan` and `make-plan-elt` follow the list of fields in the structures (see Table 4.4). Plans are constructed recursively, beginning with the construction of the `plan-elt` structures which make up the actual references. These references are then formed into lists which can be included in newly constructed `plan`

structures. By composing these constructions, any desired Plan can be manufactured. The `plan/encapsulate` procedure is used to ensure that a given Plan is a series-type Plan. The Sponsor uses this procedure to convert input Plans which are either single elements or parallel-type plans into the series-type plans that are expected. This is done by constructing a series plan whose only step consists of the single element or parallel-type plan respectively. Using this procedure allows the Sponsor to accept the results of Referrals which do not submit a series Plan as requested by the protocol.

The other procedures in the Plans module are more relevant to a Sponsor, since they deal with the interpretation of Plans. These procedures are listed in Table 5.7.

| `plan/...` | Arguments expected; return type |
|---|---|
| `compute-scheduled-sites` | `self`:Plan; list |
| `check-plan-syntax` | `self`:Plan; boolean |
| `prune-and-record` | `self`:Plan, `client`:`client-file`; Plan |
| `push!` | p:series-type Plan; – |
| `pop!` | p:series-type Plan; Plan |
| **`plan-elt/...`** | **Arguments expected; return type** |
| `eqv?` | `p1`,`p2`:`plan-elt`; boolean |
| `execute` | `self`:`plan-elt`, `client`:`client-file`, `sponsor`:Sponsor; – |

Table 5.7: Procedures for Manipulating Plans

The `compute-scheduled-sites` procedure takes a Plan and returns a list of all elements included in the input Plan. The `check-plan-syntax` procedure checks the syntax of an input Plan and returns true if the syntax is correct. In order to be robust, the Sponsor must use this procedure to check all Plans received from external sources. The `prune-and-record` procedure takes as parameters a Plan and a Client File (see Section 5.2.3 for details about the `client-file` structure). A new Plan is constructed containing only the elements which have not been submitted to the given client, and this new Plan is returned as the result of the procedure. All new elements are duly recorded in the master list of submissions stored in the Client File.

Plans may be manipulated with the `plan/push!` and `plan/pop!` procedures. These procedures are used by the Sponsor to manipulate Plans in accordance with

the responses of a Filter, which are described in Section 5.1.3. In order to provide procedures that are useful for designing Filters, these procedures do not provide the functionality that might be expected from their names (perhaps they should be named differently). These procedures take a series-type Plan as an input, and manipulate the item list. The `pop!` procedure behaves as one might expect, removing the topmost item and returning it as a result. However, the definition of the `push!` procedure is somewhat unusual.

In order to reduce the necessity for Filters to maintain state between executions, the `push!` procedure allows a Filter to peek at the next item in a Plan without losing the topmost item. This is accomplished by swapping the two topmost items in the Plan, in that way bringing the second element to the top and deferring to a later time consideration of the element that was originally on top. This feature has limited utility, since it only effects the top two items, but it can ease the construction of simple Filters. Since the number of cycles used by Filters is often at a premium, simpler Filters may be desirable, and in such cases the `push!` procedure is often useful.

The remaining two procedures operate on the elements of a Plan. The procedure `plan-elt/execute` takes as inputs a `plan-elt` structure describing a service to investigate, the Client File associated with the Client authorizing the search, and the Sponsor which is processing the given Client. It then queues up the appropriate actions in the given Sponsor's Schedule and deducts any costs associated with the service from the amount listed in the Client File. If the Client cannot authorize the expenditure, a message indicating insufficient funds is queued to be sent. In our implementation, the Sponsor is a single-threaded process, so entries that are placed in the queues at this time will actually be processed at a later time. However, the design allows for the construction of multi-threaded Sponsors in which the processes that add entries to the queues operate concurrently with processes that remove entries from the queues. This sort of design would have a significant impact on the performance of a high-volume system.

The `plan-elt/eqv?` procedure is used to determine whether or not two `plan-elt`

structures refer to the same service. This determination is made based on the Addresses stored in the structures, as well as on the contents of the `request` fields. The `eqv?` procedure is used to remove duplicate suggestions when a Plan is pruned. Our implementation does not worry about the contents of the `cost-estimate` field, but in a large system it would make sense to ensure that if two equivalent `plan-elt` structures are submitted, the one with the lower value in the `cost-estimate` field is kept.

### 5.2.3 The Client-Files Module

The purpose of the Client-Files module is to store and make use of the information related to a client. Much of the work of a Sponsor is accomplished using the procedures in this module.

| Structure `client-file` | |
|---|---|
| `oid` | the OID associated with the Client by the Sponsor |
| `query` | the Query object submitted by the Client |
| `stop-time` | time at which the search is scheduled to end |
| `money-left` | amount of money remaining |
| `money-used` | amount of money used |
| `evals-left` | number of evaluations allowed |
| `evals-used` | number of evaluations used |
| `servers-requesting` | list of servers requesting solutions |
| `requests-pending` | list of times and requests in process |
| `places-looked` | list of completed requests |
| `places-scheduled` | list of requests not yet executed |
| `plan-threads` | queue of `plan-process` structures |
| `last-filter-loc` | pointer to the last Plan filtered in `plan-threads` |
| `filter-state` | `closure` structure representing current filter state |
| `processed` | boolean indicating whether the client has been recently processed |

Table 5.8: Structures maintained by the Client-Files module

Much of this structure is self-explanatory, but there are a few points which should be elaborated. First of all, we are now seeing the implementation of those lists of `plan-elt` structures which have been mentioned so many times before. The "master

list" referenced in the last section is actually the concatenation of three lists of plan elements. These three lists are stored in the `requests-pending`, `places-looked`, and `places-scheduled` fields of a `client-file` structure. When a new Plan is received, any elements of the Plan which do not appear in either of the three are put into the `places-scheduled` list. When one of these elements is chosen for execution by the Filter, that element is time-stamped and moved into the `requests-pending` list. When a response is received, the element is moved into the `places-looked` list. As we mentioned before, elements that get lost or fail will be retried intermittently.

The Plan queue is stored in the `plan-threads` field of the Client File. The `filter-loc` field stores a pointer into the queue which references the last Plan processed by the Filter. In general, new Plans are inserted into the queue after this point, so that they will be the next to be processed. The state of the Filter is stored in the `filter-state` field. Between executions of the Filter, this field contains a closure which may represent either a stopped process (that is, one which has returned a result and stopped) or a suspended process (that is, one that was interrupted). When the Client is next processed, the Filter will either be run on the next Plan or will be continued, depending on the contents of the `filter-state` field.

The last field in the structure is the `processed` field. This field stores a boolean value which indicates whether the Client File has been processed in the last "cycle". This flag is part of a system for fairly distributing the available processing resources. The constraints are two-fold: first, scenarios in which Clients go for a long period without being processed are to be avoided, and second, when new Plans arrive for a Client, that Client should be processed sooner than was originally scheduled, since action may need to be taken based on possible new information in those Plans.

One of the queues in the Sponsor's schedule is devoted to servicing Clients. This queue, called the `service-client` queue, generally holds one entry for each Client currently being served by the Sponsor. These entries are stored in the queue in the order in which they are to be processed. Processing a Client means that the Client's Filter will be run for the allowed number of evaluations, and any responses returned during that time will be acted upon. After a Client is processed, the `processed` field

of its associated Client File is set to true. When a Client is processed its entry is removed from the queue and re-inserted at a time later than the present time. The duration of this delay determines the period with which the Client's Filter is re-run, and may be determined in a number of ways. For example, the delay can be a fixed amount of time, or it can be dependent upon the average length of time taken by the various Filters involved.

In order to protect against certain types of unfair scheduling, a Client whose File is marked "processed" will not generally be processed again during the present cycle. If such a File appears at the front of the queue, it will be placed in the "hold" queue. When all the Clients are in the hold queue, (i.e. they are all marked "processed") a new cycle begins. All Client Files in the database will have their `processed` fields set to false, and the held entries will be reinstated in the regular queue. As we mentioned above, when a new Plan arrives for a Client, the entry associated with that Client gets pushed towards the front of the queue. The interrupting entry is inserted with a time value corresponding to the current time. This occurs without regard to whether the Client has already been processed in the present cycle; if it has, the `processed` flag of that Client's File is set to false. In this way, fair cyclic service of the Clients is balanced with a Client's need to quickly respond to new information.

This system has some interesting properties. In order to see how this works, let us examine a version of this system in which we set the delay before which clients are rescheduled to a fixed value. Suppose that in such a system the queue begins to run behind schedule. In such a case, any preemptions due to new inputs will be queued for the time at which the inputs are received, after any of the entries in the queue which are already late. For example, if the remaining Clients in the cycle were intended to run before the present time, those Clients will be serviced before any Clients who preempt now. By making this and other simple analyses, we conclude that poor scheduling and preemption should cause only negligible interference with the scheduled servicing of Clients. While Clients can be pushed ahead in the queue, during periods of congestion there is a decent chance that the Client will instead be further delayed.

Let us now look at the procedures implemented in the Client Files module. New Client Files are created using the `create` procedure, which takes an Unsolved Query object and returns a new Client File based on the information in the Query. This procedure and a number of others are listed in Table 5.9.

| client-file/... | Arguments expected; return type |
|---|---|
| create | query:Query Object; client-file |
| add-to-scheduled | self:client-file, item:plan-elt; – |
| search-redundant? | self:client-file, item:plan-elt; boolean |
| transfer-scheduled->pending | self:client-file, attn:OID; – |
| transfer-pending->looked | self:client-file, attn:OID; – |
| is-scheduled? | self:client-file, attn:OID; boolean |
| is-pending? | self:client-file, attn:OID; boolean |
| spend! | self:client-file, cost:integer; boolean |
| update-query! | self:client-file; – |
| import-plan | self:client-file, plan:Plan; – |
| cancel-pending-requests | self:client-file, sponsor:Sponsor; – |
| service-client | self:client-file, sponsor:Sponsor; – |

Table 5.9: Procedures for Manipulating Client Files

In order to provide the necessary access to the three lists of Plan elements, several procedures are provided. These procedures are not necessary from a top-level view, since there are other higher-level procedures which provide this function indirectly. Regardless, we do include a quick description of them because they form part of the interface between the Client Files module and the other modules, for example the Plans module.

These six procedures summarize the behavior of elements in the request lists. New elements are placed in the `places-scheduled` list using the `add-to-scheduled` procedure. A prospective element can be tested for redundancy (i.e. presence of an equivalent element in one of the lists) using the `search-redundant?` predicate. When a request is being executed, it can be moved from the `places-scheduled` list into the `pending-requests` list by invoking the `transfer-scheduled->pending` procedure. Similarly, when an answer does come back, the completed request can be moved into the `places-looked` list with the `transfer-pending->looked` procedure. The last

two procedures, `is-scheduled?` and `is-pending?`, are predicates used to determine membership in the `places-scheduled` and `pending-requests` lists respectively.

The `spend!` procedure directly modifies the `money-left` and `money-used` fields of the Client File structure. If an insufficient amount of money is left, a value of false is returned. Otherwise, the amount spent is deducted properly and a value of true is returned.

The `update-query!` procedure makes a direct modification to the Query stored in the `query` field of the `client-file` structure. The Query is updated to reflect current constraints on time and money (that is, the `max-cash-avail` and `max-time-avail` fields are set). In general, this should be done before the Query is sent to a Referral server so that the Referral servers can make judgements based on the most recent information.

The remaining three procedures in the Client Files module are high-level procedures called by the top-level Sponsor code. The `import-plan` procedure does any work associated with incorporating a newly submitted Plan into a Client's existing information. The Plan passed to `import-plan` is assumed to be syntactically correct. The procedure first ensures that the Plan is a series-type Plan, and then calls the `plan/prune-and-record` procedure to add the elements to the `places-scheduled` list and to remove any redundancies. The resulting pruned Plan is then added to the `plan-threads` list at the place in the list which will be next processed by the Filter.

The `cancel-pending-requests` procedure queues up a cancellation request to each server in the `pending-requests` list. These cancellation requests take the form of Unsolved Query objects with the `cancel` field set to true. This procedure is used by the Sponsor to stop any processing being done on Referral servers after a search is cancelled.

The final procedure, `service-client`, is a comparatively complex one. This procedure sets the `processed` bit to true and runs the Client's Filter on the next Plan in the queue, or continues an existing Filter process, for the maximum number of evaluations. Any responses returned by the Filter are handled; if the Filter replies with `execute`, the current element is executed, etc. If the Filter makes a request to

sleep, or if the Filter exceeds the evaluations allotted to a single run of a Filter, the amounts of resources used are recorded, and the Client's next servicing is scheduled. In the event that the Filter requests `rewind` or sleeps, the queue of Plans is sorted by priority.

### 5.2.4 The Filter Simulator Module

A great deal has already been written about the Filter simulator and what it does. See Section 4.4 on Filters for a description of how Filters are represented and interpreted, specifically Table 4.5 describing the `closure` structure. Also see Section 5.1.3 for information about the interface between Filters and Sponsors and about how Filters are written in general. Since the The Filter Simulator Module is built around the `closure` structure, and has only a few procedures in its interface, the interface is fairly easy to describe. The functionality is complex, but has been described at length elsewhere.

| `closure/...` | Arguments expected; return type |
|---|---|
| `start-thread` | `exp`:list, `env`:Environment, `steps`:integer; `closure` |
| `continue-thread` | `clos`:`closure`, `steps`:integer; `closure` |
| `stopped?` | `clos`:`closure`; boolean |
| `error?` | `clos`:`closure`; boolean |
| `env` | `clos`:`closure`; `envir` |
| `result` | `clos`:`closure`; value |
| `explain` | `clos`:`closure`; – |
| `*steps*` | Global variable storing number of evals left |

Table 5.10: Procedures for Evaluating Filters

There is no explicit creation method for `closure` objects. A valid `closure` object can be formed using the procedures `start-thread` or `continue-thread`. Since `start-thread` does not require a valid `closure` as one of its parameters, it takes on the function of a creation method. The `start-thread` procedure takes as arguments an expression `exp`, an environment `env`, and a number of steps `steps`. It evaluates `exp` in the environment `env` for at most `steps` evaluations. If an error or a result is

returned, closures representing these states are returned. In these cases, the number of steps left is stored in the global variable `*steps*`. If the expression is not completely evaluated after the specified number of evaluations, a `closure` representing a continuation is returned. Given such a `closure`, the `continue-thread` procedure can be used to continue the evaluation process for a specified number of steps.

When a `closure` structure is returned from one of these procedures, there are several procedures which can be used to determine the outcome of the evaluation. The `stopped?` procedure is a predicate which returns true if the input `closure` terminated and returned a result. If so, this result can be retrieved by passing the `closure` to the `closure/result` procedure. In the process of evaluating the Filter, changes may have been made to the environment which should be maintained during the next run. In order to make this possible, the latest version of the environment that was used can be retrieved with the `closure/env` procedure. This environment should then be used for the evaluation of the next `filter` invocation, so that any state that was saved on the last run can be used during the next run.

In the event of an error, the `error?` predicate will be true for the resulting `closure`. The most convenient way to generate an error message is to use the `explain` procedure, which returns a string describing the input `closure`. If the `closure` represents an error condition, this string includes a partial listing of the stack leading to the error.

### 5.2.5 The Sponsor Module

The top level of the Sponsor implementation includes a number of procedures which control the operation of the Sponsor server from a high level.

A new Sponsor object is created using the `sponsor/create` procedure. At the time of creation, the `reject-client-hook` and `formulate-initial-strategy-hook` fields are initialized with Scheme procedures, partially defining the behavior of the Sponsor server. It is planned to add additional hooks to this list so that the range of Sponsor implementations can be expanded, but at the moment these are the only hooks.

| sponsor/... | Arguments expected; return type |
|---|---|
| create | name:String, reject-client-hook, formulate-initial-strategy-hook: Procedure; Sponsor |
| receive-input | self:Sponsor, object:object; – |
| process-queues | self:Sponsor; – |
| cancel-search | self:Sponsor, client:Client File; – |
| complete-search | self:Sponsor, query:Query object; – |
| process-new-client | self:Sponsor, query:Query object, reject-reason:String; – |
| query-update | self:Sponsor, update:Unsolved Query; – |
| deliver | item:object; boolean |
| schedule-now | self:Sponsor, tag:symbol, item:object; – |
| schedule-after | self:Sponsor, tag:symbol, item:object; – |
| schedule-at-time | self:Sponsor, tag:symbol, item:object, time:Time; – |

Table 5.11: Top level procedures in the Sponsor

The next two procedures neatly split up the Sponsor's work. The first procedure, `receive-input`, takes as input a Sponsor and an object which has been sent to the Sponsor, and does whatever is required to respond to the object. Whenever the Sponsor receives a message, the object it contains is passed to the `receive-input` procedure. The second procedure, `process-queues`, does all the work associated with acting on the entries in the queues. The queues are processed up to the point at which all entries in the queues are scheduled for times after the present time. The `process-queues` procedure must be called on a fairly regular schedule in order for the server to function well. The next section contains a detailed description of the protocol implemented by a Sponsor. This description is divided into two parts, exactly along the division specified by these two procedures.

There are two procedures devoted to cancelling a search. The `cancel-search` procedure is used to cancel the search in cases where the Client or the Sponsor requests its termination. When the search is cancelled, all unprocessed entries in the Sponsor's queues that are related to the search are removed, excepting messages to the Client. Then, using the procedure `client-file/cancel-pending-requests`, cancellation

messages are forwarded to any servers currently processing requests related to the search. Finally, the Client is sent a termination message and the Client's File is removed from the database.

The second cancellation procedure, `complete-search`, is used to cancel a search when a Solved Query object has been sent by the Client. As we have seen before, when the Client terminates the search, the Client may optionally provide a set of documents which form a solution to the question. A Solved Query object is used to communicate this information. When such an object is received by the Sponsor, `complete-search` is called. The work of cancellation is still done by the `cancel-search` procedure, but after the search is cancelled the Solved Query is sent to any servers which requested the solutions determined in the search.

The next two procedures are involved with the receipt of Unsolved Query objects. When a Query object arrives with an unfamiliar `search-id`, the Sponsor must decide whether or not to Sponsor the new Client and must then form an appropriate reply. This is done by the procedure `process-new-client`. If the Client is rejected, a rejection notice is sent as a simple message containing a reason. If the Client is accepted, an acknowledgement is sent, a record is entered into the Sponsor's database, and the search is started.

Beginning the search amounts to first forming an initial plan and then entering a `service-client` request and a `kill-client` request in the appropriate queue. This initial plan is formed by calling the `formulate-initial-strategy-hook` procedure which is stored in the Sponsor structure. By shifting this strategy into a hook procedure, the Sponsor kernel allows different Sponsor implementations to use different strategies for starting the search. An initial plan can be chosen based both on the text of the Query and on the characteristics and methods of the individual Sponsor. This plan is then imported in the usual way using the `client-file/import-plan` procedure.

At the initiation of a search, a service request and a kill request for the new Client are entered into the `service-client-queue`. This queue, the mechanics of which appear in Section 5.2.3, essentially contains an ordered list of Clients to service. In

fact, the queue contains at all times exactly one service request and one kill request for each Client. When a Client is serviced, a new service request is entered at a later point in time. The kill request for a given Client is entered when the Client is accepted, and is set to be acted upon after the pre-arranged time limit has expired. This time limit is specified by the `max-time-avail` field of the Unsolved Query object, but may be extended with a `renewal-from-client` message from the Client.

A Client may also communicate revisions of the original Query by sending the Sponsor an Unsolved Query relating to a search in progress. Any Unsolved Queries that do not have a higher `revision` value than that of the Client's current Query are ignored. Those Queries that do represent a new revision essentially cause the Sponsor to start the search over with the new information. This is done by calling the `query-update` procedure, which essentially cancels the old search and resubmits the new Query. It may be the case that the Sponsor does not accept the revised Query, in which case the Client will have to attempt to restart the search. It would be a fairly trivial extension of the protocol to allow the Client to select from several revision policies. For example, one choice the Client could make might determine whether to continue the search with the revised Query or to restart the search with the revised Query.

The final four procedures are more primitive. Three of these procedures are designed to manage the addition of new entries to the queues. The fourth procedure handles the sending of messages. Before we discuss these procedures, we should first give a more thorough description of these queues that are being manipulated. In Section 5.2.1, the mechanics of Schedules was described, but we have yet to make clear exactly how queues are used in the Sponsor.

A Sponsor's Schedule is composed of four queues. Each of these queues maintains an ordered list of entries which request that various actions be performed at various times. Earlier in this discussion we mentioned the `service-client-queue`. The other three queues are the `send-client-message-queue`, the `send-lsr` queue, and the `send-referral` queue. The fact that these queues are kept separate makes it easy to halt the processing on one of them, or to speed up the processing on another,

based on time requirements or on environmental factors such as network loading.

Entries are added to these queues with the `schedule-now`, `schedule-after`, and `schedule-at-time` procedures. These procedures take as parameters a `tag` and an `item`, and place an entry containing that `tag` and `item` in the appropriate queue, scheduled for the appropriate time. The choice of queue is made based on the specified `tag`; for example, a `tag` of `send-lsr` would cause an entry to be added to the `send-lsr-queue`. The time is determined in different ways by each of the procedures: the `schedule-now` procedure sets the `time` field of the entry to the current time, while the `schedule-after` procedure sets it to the `time` of the last entry in the queue, and the `schedule-at-time` procedure sets it based on the value of a `time` parameter that is given as an argument.

The last procedure to describe is the `deliver` procedure. This procedure is responsible for delivering all messages sent by the Sponsor. It takes a message `item` as an argument, and returns true if it successfully delivered the message. These messages are all Search objects, and because of the inheritance in the Search object hierarchy, all messages will have the addressee stored in the `to-addr` field. In our implementation, and any that implement a reliable datagram transmission technique, this procedure can send the message and return after any necessary acknowledgement has been received (or in the case of TCP, open a connection, send the message, and close the connection). However in order to increase the efficiency of the server it may become necessary to do other useful processing while waiting for acknowledgement or while waiting for a connection to be established, etc. In the case of TCP, this delay might be as long as 30 seconds or so, or the connection might never be established after waiting for the software to time out. In the event of such a failure the unsent message would probably be temporarily shelved in a hold queue while other messages are sent.

## 5.3 The Protocol

In this section, we will describe the protocol implemented by the Sponsor in detail, but more tersely. We will make many references to the structures and procedures discussed in this chapter, so it may be difficult to comprehend without reading the previous sections. We will begin by enumerating the Sponsor's response to the possible incoming messages.

*Upon receipt of...*

**Unsolved Query object** (*un*familiar `search-id`):

Call the `reject-client-hook`. *On result...*

**Yes** Send a simple message to return address of Query, with a string argument listing a reason and with a token value of `sponsor-reject-client-token`.

**No**
1. Create a new Client File and add it to the database.
2. Place an immediate service request in the `service-client-queue`.
3. Place a future kill request in the `service-client-queue`.
4. Send a simple message to the Client's address, with a token value of `sponsor-accept-query-token`.

**Unsolved Query object** (familiar `search-id`):

1. Look up the Client File associated with the `search-id` of the Query.
2. If the `revision` field of the new Query is greater than that of the Query stored in the Client File, or if the `cancel` field of the new Query is true, *then...*
   (a) Send a copy of the Client's Query, with the `cancel` bit set, to each server in the `pending-requests` list.
   (b) Send a simple message to the Client's address, with a token value of `sponsor-termination-token`, and arguments set to time, money, and evals used.

(c) Remove the Client's File from the database.

3. *Otherwise,* ignore the Query.

**Solved Query object** (familiar `search-id`):

1. Send a copy of the Client's Query, with the `cancel` bit set, to each server in the `pending-requests` list. This is accomplished by using the procedure `client-file/cancel-pending-requests`.
2. Send a copy of the Solved Query to all servers in the `servers-requesting` field of the Client File.
3. Send a simple message to the Client's address, with a token value of `sponsor-termination-token`, and arguments set to time, money, and evals used.
4. Remove the Client's File from the database.

**Client-directed Proposal** (familiar `search-id`):

1. Use the procedure `plan/check-syntax` to verify that the new Plan is syntactically correct.
2. Add any new Plan Elements to the Client's `places-scheduled` list; assign each of them an OID; form a new Plan which leaves out any redundant Elements. This step is accomplished with the `plan/prune-and-record` procedure.
3. Add the new Plan to the Client's Plan queue so that it is the next to be processed, using procedure `client-file/import-plan`.
4. Reschedule the Client to be serviced at the present time (i.e. after any entries that are already behind schedule).

**Sponsor-directed Proposal** (familiar `search-id`):

Sponsor-directed Proposals are an extension to the protocol. The idea is that the Client may be given more control over the search, at the expense of requiring

additional bandwidth in the Client-Server connection. Essentially, the Sponsor can be put into a mode in which the Filter can request Client participation in the decision about a given Plan. The Filter indicates that a given Plan should be turned over to the Client for a decision; the Plan in question is sent to the Client in a Proposal object; the Client replies with a revised Plan in a Sponsor-directed Proposal; this Plan is checked for proper syntax and imported using the `client-file/import-plan` procedure.

**Client-directed Bundle** (familiar `search-id`):

1. Look up the Client File associated with the Bundle's `search-id`.
2. Set the Bundle's `to-addr` field to the Client's Address.
3. Set the Bundle's `return-addr` field to the Sponsor's Address.
4. Send the modified Bundle.

**Sponsor-directed Bundle** (familiar `search-id`):

1. Look up the Client File associated with the Bundle's `search-id`.
2. Organize the items in the Bundle's Document List into collections of Documents with equivalent `server-addr` field.
3. Form each collection into a separate Document List.
4. For each of these lists, formulate a Local Search Request, with the `to-addr` field set to the server holding the documents, and with the commands field listing the OIDs of the documents in the list following the `:retrieve` keyword.
5. Send the Request objects.

**Simple Message** (familiar `search-id`):

The only simple message accepted by a Sponsor is a message from a Client with a token value of `renewal-from-client`; all others are ignored. If such a message is received, then:

1. The specified amounts of money and evals are added to the totals listed in the Client's File.
2. The `kill-client` entry is dequeued from the `service-client-queue`.
3. A new `kill-client` entry is queued, offset by the specified additional amount of time.

We have now completed a description of the protocol of responses to the various communication objects used in the meshFind system. Any messages which do not fit into this protocol are erroneous and must be ignored. Erroneous messages can be expected to occur on a regular basis as a result of delayed or retransmitted messages, errors in the implementation of the various servers, or other failures in the system. In order for the system to work at all the component parts must be robust and the protocol must be clear about what does and what does not constitute a valid message.

The other half of the protocol is implemented as part of the system of queues. Various parts of the system insert queue entries which are then then acted upon at a later time. This part of the protocol can be described in terms of responses to elements of the queues.

*When processing an entry with a tag of...*

**kill-client** Call the procedure `sponsor/cancel-search`.

This procedure will send the Client a termination message as part of the cancellation process. This message is a simple message with a token of type `sponsor-termination-token`, and includes in its arguments statistics about the search.

**service-client** Call the procedure `client-file/service-client`:

- This procedure runs the Client's Filter program on submitted Plans, for a certain number of evaluations. The Filter may decide to "execute" certain parts of the submitted plans.
- When the leaf of a Plan is executed, the Sponsor must send a request out on the Client's behalf. This request may be either a Local Search Request

or an Unsolved Query object, depending on the contents of the executed Plan element. The `to-addr` field will be the Address of the desired server, and the `return-addr` field will be the Sponsor's address. This request is then queued to be sent.

- When the Filter has been run for the maximum number of evaluations, it is suspended until the next time the Client is serviced. A Filter may also choose to "sleep" for an arbitrary amount of time. If new Plans arrive during this time, the Client will be awakened and the Filter will be run on the new Plans. In any case, when the Filter stops, a new service request for the Client will be queued for some time in the future. In this way the Clients will be serviced on a regular basis, with preemption in the event of newly submitted Plans.

**Send Message** (`tag` $\in$ {`send-client-message`, `send-lsr`, `send-referral`}):

The given message is sent over whatever protocols are in place. These protocols may vary depending on the recipient; for instance, the Client may be linked by modem or serial line to the Sponsor, while the Referral and Index Servers communicate using TCP.

This concludes the description of the Sponsor's protocol. The next chapter goes through a similar process to describe the protocols used by the Index and Referral servers. The chapter also covers the facilities we developed for our implementation to allow our servers to access a variety of information services.

# Chapter 6

# The Referral and Index Servers

In a full-scale meshFind system, the bulk of the infrastructure lies in the network of Referral servers and Index servers. It is here that the knowledge of the topology of the network is built and maintained. The Referral servers are also responsible for the interpretation of questions submitted by Clients. Essentially, the problem of offering a resource discovery service has been broken into three parts: the Sponsor manages the services provided to the Client and provides the Client with a single local contact, the Referral servers provide hints for finding the desired information, and the Index servers serve as archives and front ends to actual services.

In this chapter, we will examine the Referral and Index servers in greater detail. We will begin with two sections which describe the general protocols implemented by Referral servers and Index servers. The next sections will discuss some interesting details of the implementation. Among these are the techniques used to connect meshFind to actual services in the Internet and a set of procedures used to interpret the text of submitted questions.

## 6.1 The Design of the Referral Server

### 6.1.1 The Standard Referral Protocol

The standard protocol followed by a Referral server is fairly simple. As we have seen before, a Referral server's primary job is to interpret a Client's query and return a Plan to the Client which hopefully will help answer the question. There are a variety of extensions to the standard protocol which can be used to provide additional features and potentially to improve the quality of service.

A Client or a Sponsor can request help from a Referral server by sending an Unsolved Query object to the Referral server's address. When the Referral receives the Query, it begins processing it. Usually the Query is matched against a cache simultaneously with the commencement of processing, and if an equivalent Query is found in the cache the known answer is immediately returned. Although a cache is not necessary, it makes a number of enhancements possible. Support for prefetching can be easily built on top of a cache by processing prefetch requests as normal requests, caching the results, but just neglecting to send the results back to the server requesting the prefetch. Another benefit of caching recent results is an increased tolerance of requests to resend results, which may occur when servers are overloaded and communication is delayed. Finally, servers which answer a few questions very frequently will find a cache extremely beneficial.

The exact nature of the processing that is done on a Query in order to resolve it into a Plan may vary from server to server. It is here that diversity manifests itself in the Referral network; no two servers necessarily use the same technique for determining their response. This is a powerful feature of the Referral network, since more specific Referral servers can use techniques which are more specific to the subject matter they cover. The set of query languages that a specific Referral server understands is another factor that varies among servers. The subject of query languages is discussed in detail in Section 6.1.2, but the essential idea is to have most Referral servers understand the meshFind language (which we will see is actually just English) and to allow them to also understand any other query languages that Clients

might want to use. To the degree that translation is possible between languages, translation services may exist which provide translations to a Sponsor or to Referral servers. The issues involved with translation are also covered in Section 6.1.2.

When a Referral determines a Plan in response to a Query, it packages that Plan in a Proposal object and sends that object to the return Address specified in the Query. Referral servers are permitted to respond after arbitrary delays, although if the delay is going to be particularly long, this should be reflected in the `typical-delay` field of the server's `server-info` block so that the Sponsor does not needlessly resend its requests. The intention of the `typical-delay` field is to give a reasonable upper bound on the response time. The Referral server should make an effort to complete its processing within this time interval in most of the cases, and should try to balance its load in an effort to maintain that level of service. One easy way to balance load is to have multiple mirrored copies of a popular Referral server. Whenever one of the servers is overloaded, all new requests are answered immediately with a Plan which refers the sender to a less busy mirror server. The identification of an underutilized server can be done if the servers occasionally broadcast to each other their present loading condition.

Referrals may be implemented as either multi-threaded or single-threaded systems. Probably, the distributed nature of the Referral network will reduce the load on individual servers and make it possible for many of them to be of the simpler single-threaded variety. However, it is clear that a heavily-used server will need to be multi-threaded; that is, it will need to process requests in a pipeline so that time is not wasted waiting through communication delays. This is especially true if processing a request requires communication over the network to remote information sources, such as dictionaries and encyclopedias which dispense general information useful for interpreting a query. Depending on the system underlying an implementation, multiple threads may be easy or hard to implement. Our prototype system is implemented in a Scheme environment that has threading but does not support interrupts, which makes it hard to get much benefit out of a multi-threaded Referral server.

This concludes the description of the standard protocol implemented by Referral

servers. As we see, this protocol is fairly simple and should not be difficult to implement. Many extensions can be made to this protocol, and some of these are discussed in Section 6.1.5. The next section will discuss issues connected with query languages, and will specify the details of meshFind's standard query language.

### 6.1.2 The Query Language

Many languages have been invented for the expression of queries. Typically every new system comes with its own query language, perhaps only slightly different, but generally somewhat specialized to the new system's particular needs. For the purposes of this system we chose a minimalist approach in designing a new language. Our intention was to encourage the implementation of servers which understood more natural query languages, while at the same time making the system as backwards-compatible as possible. This language we developed defines the form of the `question` field of a Query object. In this section and the next we will give a more precise definition of the syntax of our Queries.

The syntax of the language we chose takes the form of a list of lists. Parentheses are used to delimit the lists, and the standard LISP keyword syntax is used. The first item in each list within a query is a keyword identifying the syntax of the contents of that list. Let us consider a simple example:

```
((:meshfind "space probes launched by the usa in 1972"))
```

This example gives the query using only one syntax, in this case the standard meshFind syntax. The use of this particular syntax is indicated by the `:meshfind` keyword preceding the string. Because it is given in the meshFind format, the string will be interpreted essentially as English text. In general, a query may be specified in a number of different formats, using different keywords to indicate which format is being used. These keywords and syntaxes would need to be well-known so that clients and servers can agree on their meaning. It is unclear what kinds of explicit mechanisms can be used to make this work, since the production of formal descriptions of a syntaxes and query languages is a very difficult problem to solve in the general

case. For that reason, we prefer to think of it for the present as a kind of culture that builds over time; certain servers accept various languages with generally agreed-upon names, and they advertise which languages they accept. Clients can then see the collection of culturally accepted languages, and use them to express their questions. For example, in one possible culture, we might have specified the query shown above using multiple syntaxes in the following way:

```
((:meshfind "space probes launched by the usa in 1972")
 (:harvest "space AND probes AND USA AND 1972"))
```

In any case where a query is specified using multiple formats, the recipient of the query may use any of the versions of the query that can be processed. A Referral server might understand several of the different syntaxes, and would then have the option of using either version or both in the resolution of the query. By giving the same query in several forms, the Client has a better chance of having the query successfully resolved. Because translating queries can be a bit tricky, often this can result in a more effective search when the query finally gets to the underlying service. This is especially true if the Client is well versed in the syntax of the underlying services and can make a more intelligent translation than the one that will be done mechanically. However, in a completed system it is hoped that these translations will become fairly good, and that the natural language used in the standard meshFind syntax will become increasingly sufficient.

What types of processing can be done to interpret a query specified using the meshFind syntax? In our prototype, we do some fairly routine text-processing to come up with what we hope is a reasonable first approximation. In the case of a General Referral server, there is not much more that can be done reasonably. Although text-skimming approaches which do shallow parsing of full text documents have been shown to have some degree of utility in [21], full-blown natural language processing is presently beyond the reach of our technology, and even a text-skimming system would be difficult and time-consuming to implement, since it needs to have an extensive vocabulary and frame database to be useful. However, as we get into

increasingly closed domains, natural language processing becomes a possibility, and the English text in a meshFind query could conceivably be interpreted with some success. We suspect that many overlapping narrow-range expert systems would be more effective than a global approach such as a universal text-skimming system, and furthermore that the expert systems would not be that much more work to construct. Unfortunately, the implementation of such systems, though feasible, is well beyond the scope of this work.

### 6.1.3 Text Processing by Referral Servers

The text processing done by the Referral servers in our prototype is fairly simple, and is based primarily on keyword search. When a query comes in with the `:meshfind` syntax indicator, the string that makes up the next item in the list is removed, and the words are separated out. Words are delimited by whitespace or parentheses in this step. If the parentheses fail to match up they are thrown out. Otherwise, the words are put into a list structure based on the placement of parentheses. Then the structure is filtered with a stop list, causing really popular and meaningless words to be eliminated. At the same time, the words "and", "or", and "not" are parsed and converted into symbols.

After this processing, the query has been reduced to what is essentially a list of keywords. The list structure formed by the parentheses will cause some of the items to be conjunctions and disjunctions of keywords. Some of the keywords may be marred by punctuation, so if a keyword is not recognized at first, it should be stripped of punctuation and retried. The Referral must then make use of the keywords forming the query, using its own particular methods. The servers we constructed usually did something fairly simple, keying on words that might indicate a more specific area to try, and making a list of these more specific servers. This list of servers is then ordered by estimated relevance and grouped into classes of similar relevance, which is then formed into a Plan which can be returned to the issuer of the query. Keyword stemming is required in order to make this system work, but because of the difficulty involved with implementing stemming, our prototype includes redundant keywords

on the matching end in order to get around the problem (i.e. it recognizes "launched", "launches", and "launch" rather than only needing "launch").

In the case of the example shown above, our system associates the words "space" and "launched" with the general area of space research. Since two space-related words appear, the relevance of a Referral specific to space research would then be higher than the relevance of other servers which might key off only the "space" keyword, such as servers relating to interior design. It is interesting to note that although as a rule systems based on keyword search include "space" in their stop list because it has so many meanings, our system still uses such words to provide guesses about the overall topic of the query. As the query is submitted to Referral servers of increasing specificity, wrong turns will be weeded out much more easily.

### 6.1.4 Topology of the Referral Network

The question of topology is an interesting one, and it is intertwined with other issues about the possibility of mechanical construction of the Referral network. Although it is often easiest to think about as a neatly ordered hierarchy, certainly the Referral network is not constrained to be purely hierarchical. In fact, it will be more powerful if it is not, and the construction will be easier as well. One of the largest problems involved with constructing the network is that the General Referral may make a wrong turn – that is, it may direct the Client's search down the wrong path, into more specific servers that cannot answer the query. In order to counter this, the more specific Referrals should have cross links to domains that are in parallel parts of the hierarchy, thus increasing the chances that a misdirected query can get back on the right track. This technique cannot solve all the problems of misdirection at the higher levels, but it is certainly an argument against a strict hierarchy.

The construction and maintenance of the Referral network is an enormous job. As the amount of data in the system increases, this job will become increasingly difficult. Certainly any plan for the future will need to include some plans for the mechanical construction and maintenance of the Referral network. In the production of our prototype there was neither sufficient time nor resources available to really attack this

problem, so the network in our prototype was constructed manually. However, a reasonable amount of thought was put into this problem, and a few possible plans of attack have been determined. One idea which seemed to have some promise was that of using the abstracts contained in each server's `server-info` structure to mechanically organize the network. The structure thus produced could be arranged hierarchically using a document clustering technique similar to Gloor's hypertext system [11], or could be arranged in a mesh of linked lists similar to the Ingrid system.[9]

Each Referral server would be identified by a machine-readable abstract summarizing the scope of queries it could answer effectively. Part of this abstract might even be a piece of code that acts as a filter on incoming queries, choosing which should be directed to that Referral server. These abstracts could then be clustered into groups of similar servers using statistical techniques such as word-frequency vector analysis. This analysis is described in [24] and [25], and is applied to the similar problem of sensibly organizing hypertext documents in [11]. For each group of documents for which the associated vectors form a tight cluster, a *new* Referral server could be automatically generated. The information in the abstracts of the included documents could be used to implement the new server – essentially by implementing a filter which directs incoming queries to the appropriate Referral in the cluster. An abstract for the new server could be generated by composing the abstracts of the Referrals in the cluster. This process could conceivably be done on a routine basis, leading to a constantly changing but hopefully consistently useful hierarchy.

Certainly the details of this system are far from worked out at the present time. It is not clear how these ideas might be tried out, without first having a fairly large number of specific Referral servers. However, after the first few were complete, it might not actually be that hard to make a fairly large collection of specific Referral servers, even servers that support reasonably powerful natural language processing. Much of the code could probably be reused, in a similar way to way companies which sell customized expert systems reuse their code.

### 6.1.5 Extensions to the Referral Protocol

The protocol followed by Referral servers may be extended in many ways. One of the simplest extensions is a method for requesting the solutions that will be the eventual result of a submitted Query object. Recall from Chapter 4 that the Solved Query object is used to represent and to transmit those result documents indicated by the user to closely approximate the desired answer. Whenever a Referral server issues a Proposal object, it can set the `hindsight-request` bit of the Proposal to true. This instructs the Sponsor which receives the Proposal to send the Solved Query object related to the search back to that Referral server, if and when such a Solved Query is released by the Client. The other end of this process is detailed in Chapter 5, but essentially the Sponsor which made the request to the Referral keeps a list of servers to whom to send solutions, and if the Client indeed sends out a Solved Query object, it is automatically forwarded to the servers in the list.

There are a number of possible uses for collections of Solved Query objects. One possibility is caching, although this is difficult since as a rule the same question can be asked in a large variety of ways, and even if the question matches the answer is not guaranteed to be the same (for example, "What is the weather like?"). Furthermore, Solved Query objects store the solution that a *particular* Client accepted as a good answer, and different Clients may accept different things. In most cases the answers will not vary greatly, but there is always the potential for variation. Much of this variation may be trivially removable, perhaps by parsing and reformulating the text of the query in a canonical form. Regardless of the details of implementation, it is clear that some kind of caching must be included in a full-scale system, and Solved Queries may aid in cache construction and maintenance.

Solved Queries can be used for other purposes as well. Perhaps the most useful thing about collecting Solved Query objects might be that they provide a certain form of advertising. Referral servers may be able to learn about new resources when they appear in Solved Query objects. Approaches which make statistical analysis of the Solved Queries might yield hints about how to answer queries better. A Referral server might notice that a certain Referral seems to be helping a number of Clients, and

perhaps would begin applying a new analysis of incoming queries which would cause a reference to the newly discovered server. This is essentially another issue having to do with the mechanical construction and maintenance of the Referral network. Solved Queries provide another tool for maintenance and incremental expansion, but there are many unsolved problems, such as how these cross-links that have developed over time might be maintained in the event of a major reorganization?

Another extension to the standard Referral protocol is the capability for prefetching. The prefetching protocol has two sides, the Referral side and the Archive side. A server may at its option implement either side of prefetching, both sides, or neither side. Prefetching is a method of speeding up the service provided by a Referral server, and has been briefly described in Chapter 4. Suppose Referral server A is about to return to its Client a suggestion to try server B. If both servers A and B support prefetching, server A returns its response to the Client with the `prefetched-until` field set to some reasonable time. Server A simultaneously sends a request to server B which specifies what action server B should take and also specifies that the result should be saved in a cache of prefetched documents. This is done by formulating exactly the Request that the Client would make in response to server A's Plan, only varying from the usual routine by setting the `prefetch-until` field to a reasonable time. Server B should then compute the result and hold it in a cache, waiting for the Client to ask for it. The time specified in the `prefetch-until` specifies the minimum length of time the object should be kept in the cache.

Let us now look at each side of the protocol in more detail. On the Referral side, a server selects an element of a Plan which is about to be returned to a Client and which has a high probability of being requested by the Client, thus making it a good candidate for prefetching. The server constructs a request based on this element which is identical to the request the Client would construct in response to that element. When constructing the request, the server also sets the `prefetch-until` field, which is reserved for the purpose of making prefetch requests. This field contains the time after which the result will be removed from the cache. Typically this time should be at least ten minutes in the future, but less popular servers could probably keep

prefetches around for a few hours. If the Request is sent by a process which requires establishing a two-way connection, for example TCP, it would be possible to negotiate the prefetch-until time. Negotiation is not really that necessary, however, because if the answer is dropped out of the cache before the Client makes a Request that only means that the request will need to be processed again.

The server implementing the Archive end may be either a Referral server or an Index server. Essentially, the server must watch the incoming Requests, and whenever one arrives with the `prefetch-until` field not set to false, process that request as a prefetch request. At that point, any negotiation over the time value in the `prefetch-until` field would occur. When that issue is settled, the request is processed normally, except that the answer is not sent back to the return Address. Rather the answer is stored in the cache, and is kept there for at least the time agreed to during the negotiation. Note that typically all answers are stored in the cache for a length of time in order to protect against resends caused by confusion, lost or delayed packets, etc. The prefetched results are typically allowed to stay in the cache for a longer period of time (if negotiation is implemented, this time is agreed upon). When a Request is received by the server which has a familiar `search-id` value (i.e. something in the cache has the same `search-id`), the request is checked against the request that spawned the answer in the cache, and if the Requests match the answer is sent out immediately. This can be conveniently implemented by making the cache a hash table keyed on `search-id`.

The process of negotiation can be made fairly simple. For example, one way to implement it would be to do a simple bidding process, with a limit on the number of bids submitted. The Referral side begins with the submission of the prefetch Request, containing an initial bid in the `prefetch-until` field. The Archive side responds, either with that value or with a lower value. Responding with the same value as the last bid indicates an agreement at that value. Thus the two sides bid; the Referral side's bids must be monotonically decreasing, while the Archive side's bids must be monotonically increasing. After a specified odd number of bids the negotiation ends regardless of the outcome.

This concludes the description of the protocol implemented by Referral servers, both the standard protocol required by all Referral servers and the extensions to that protocol. The next section describes in a parallel way the protocol implemented by Index servers.

## 6.2 The Design of the Index Server

### 6.2.1 Standard Protocol of the Index Server

The standard protocol of the Index server is of similar complexity to that of the Referral servers. The Index server serves a dual purpose. First it must accept search queries and dispense document headers. Second, it must accept document queries and dispense the rest of the documents. In these capacities, it serves as an interface or wrap around an Indexing service and an Archive. It provides query (i.e. indexed) access and access by identifier to a group of documents. Both of these types of access occur using the Local Search Request object, and in both cases the response returned is a Bundle object containing a Document List.

The Local Search Request is a versatile tool which contains a single string, in the `commands` field, which specifies the nature of the request. This string may be a list of keywords to pass to the search engine, or a list of commands more specific to the protocol of the underlying information service, or a list of OIDs of documents to retrieve. It is important to understand that the Index server interface is not intended to be completely standardized across the network. Instead, the Index server is intended to provide some amount of standardization, for example in the protocol of Requests and Responses, and in general make it easier to submit queries to the underlying service. However two Index servers wrapping around different underlying service types (for example, a WAIS server and a Harvest server) might accept slightly different query formats in the `commands` string. For this reason, for any given Index server there is generally a special Referral server which provides any remaining translation and essentially suggests what questions to ask of that Index server. However, most Index

servers accept a list of keywords as a valid input to the `commands` field.

So an Index server waits for Local Search Requests to be sent to its address. Upon receipt of a LSR, the Index server processes it, resulting in the creation of a Document List structure. This Document List is packaged into a Bundle object and is sent back to the address listed in the return address field of the LSR. The processing done to generate this Document List can take two forms. If the `commands` field contains the `:retrieve` keyword followed by a list of OIDs, then the documents related to the OIDs must be retrieved from the Archive and stored in a Document List. If the `commands` field contains recognizable commands, those commands must be submitted to the underlying information service and the resulting document headers (or in same cases, whole documents) must be stored in a Document List.

In general, the system will work best if the Index is fairly tightly connected to the underlying service holding the data. For example, a server that is closely linked to its underlying search engine can queue up requests and send them in blocks, significantly speeding up the average communication time and at the same time reducing the load on the underlying server. In our prototype, we were not tightly connected to our underlying servers, but we did connect to a bunch of different families of servers, which will be discussed in Section 6.3.

### 6.2.2 Extensions to the Index Server Protocol

The main extension that we have considered is the additional protocols required to support prefetching. These protocols are detailed in Section 6.1.5, and that information holds for Index servers as well. Other than that, there are no particularly pressing extensions planned.

One issue that came up during the initial design of this project was whether or not the Index servers were really necessary. The argument against Index servers essentially said that they are an unnecessary layer, and that the functionality they provide could easily be provided by the other servers in the system. However, there are several reasons why this extra layer significantly improves the design. First, the Index servers provide a uniform interface for document retrieval, which in other

implementations would need to be replaced by a much more complex system in the Client's software, together with additional interfaces in the Referral's software. In order to retrieve documents from the many different underlying technologies, the Client would need to know all the protocols. This can get to be a mess.

Second, the Index servers provide a systematic set of gateways to the outside world. In many cases bulk transfers or other efficiency-improving techniques might be arranged (although in many other cases it will be impossible) which is easiest done if there is a layer masking it from the rest of the system. Third, the Index servers are intended to instill a certain degree of standardization on the query process by inventing more standard query languages that are easy to mechanically convert into the language that is required by the underlying service. While this standardization is not complete, it is a start and it makes the job of building Referral servers somewhat easier.

We will now move on to describe the implementation of the gateways between the meshFind system and several Internet services. These gateways are implemented within Index servers in the Referral server network.

## 6.3 Gateways to the Internet

In our prototype system we have implemented a number of gateways which allow the Index servers and Referral servers to access a variety of Internet services available through the Web and through the standard TCP interface. Existing information services such as WAIS and Harvest have been made available to users of our prototype system through several Index servers.

### 6.3.1 The WAIS Gateway

WAIS, which stands for Wide Area Information Server, is a product originally developed by Thinking Machines and currently supported and marketed by WAIS, Inc.[16] WAIS provides efficient text query access to collections of documents. Large reverse indices are generated from the full text of the documents, and queries are applied

to these indices after filtering the queries through elaborate stop lists and stemming software. The WAIS product is a very good one, and is highly efficient. Its approach is somewhat brute-force (the index is typically about as large as the collection of documents, for example), but because of its portability and good design it is the underlying search engine for a variety of Internet services. For example, many Harvest servers run WAIS as the underlying Indexing Subsystem, and *Britannica Online* also uses WAIS as an indexing engine. For more about these systems, see the following sections.

Connecting to a WAIS server is made more difficult by the complexity of the protocol. Similar to the meshFind system, WAIS servers pass around fairly complex blocks of handshaking information, including a powerful query language. The protocol is an extension of the ANSI Z39.50 protocol, which makes for a difficult implementation, or at the very least a complex one. However, WAIS also provides a very portable package of C code which can be used to implement clients and servers.

By making very minor modifications to this C library we were able to incorporate a WAIS interface into the standard MESH kernel, with a simple top level package for implementing WAIS clients in Scheme. This package allows a Scheme program to open a connection to a given WAIS server and to send a query. The response comes back in the form of a list of headlines which can be easily converted into a Document List. The full text of each document may then be retrieved using another part of the package. In general the connection to the server is closed when it is not in use and reopened if necessary; in order to save time, the most apparently relevant documents may be retrieved during the initial connection. This determination is made easier by the relevance feedback feature of the WAIS system.

Table 6.1 lists the procedures which make up the Scheme interface to WAIS. These procedures are implemented in C, and are called from Scheme using the foreign function interface provided by the Scheme→C system in which the MESH is implemented. The interface to WAIS must be initialized before it can be used, using the procedure `wais-init`. This procedure takes a string containing the pathname of the "sources" directory as an argument. This sources directory is part of the WAIS interface. For

| Procedure | Arguments expected; return type |
|---|---|
| `wais-init` | `SDirPath`:string; int |
| `wais-close` | – ; int |
| `wais-q` | `sourceFN`, `queryText`:string, `maxResults`:long, `verbose`, `closeOnExit`:int; Question |
| `wais-free-q` | q:Question; int |
| `wais-get-kth-score` | q:Question, k:int; int |
| `wais-get-kth-headline` | q:Question, k:int; string |
| `wais-assign-oid` | q:Question, k:int, `OID`:string; int |
| `wais-retrieve-by-oid` | `OID`:string; string |

Table 6.1: The Scheme Interface to WAIS

each accessible WAIS server, it contains a file ending in ".src" which lists information describing the server. The name of the server, the machine through which is may be contacted, any costs associated with the use of the server, and an abstract describing the service it provides are contained in the source file. When the WAIS interface is no longer going to be used, the `wais-close` procedure can be used to free up any memory allocated during initialization. Since the `wais-init` procedure takes some time to run, usually the `wais-close` procedure is not used until the system is being shut down.

Queries can be sent to a given WAIS server using the `wais-q` procedure. The server to which to connect is given by the `sourceFN` parameter in the form of a filename of a source file. The query is specified in the WAIS query language and is given as the `queryText` parameter. The maximum number of results to retrieve is specified by the `maxResults` parameter. The `verbose` and `closeOnExit` parameters are boolean values, which by C convention must be given as the integer values 1 and 0. The `verbose` parameter specifies the wordiness of the output to the log file, and the `closeOnExit` parameter specifies whether the connection to the server should be closed immediately or left open for future transactions. If an Index server needs to submit many queries to a single server it makes sense to leave the connection open.

When the `wais-q` procedure retrieves data from a WAIS server, it is stored in a "Question" structure, which is a fairly complex C object. As such it is not easily

transferred into the Scheme domain. In order to get around this problem, we pass the C pointer to the structure back to Scheme, and we use several C procedures to access parts of the structure given this pointer. This works reasonably well, since the pointer makes a good unique identifier, and since it can be directly used to operate on the structure. In order to guard against Scheme giving an incorrect or outdated pointer, the pointer is checked against a list of currently active question structures before it is used. When the Scheme program determines that the results of a query are no longer needed, the procedure `wais-free-q` may be used to free the memory taken by the Question structure.

There are two procedures which provide ways of accessing the Question structure. These procedures, `wais-get-kth-score` and `wais-get-kth-headline`, provide access to the summary of the documents retrieved by WAIS. By giving the Question pointer and the index of the result document, the Scheme program can retrieve the score or the headline of any of the result documents. The Scheme program may also assign an OID to a given result document using the `wais-assign-oid` procedure and then retrieve that document by passing that OID to the `wais-retrieve-by-oid` procedure. This way, any documents which are not immediately retrieved may be retrieved much later than the time of the original query by giving their OID. This is only necessary when the documents are not already identified by OIDs.

The construction of an Index server is fairly straightforward from this point. The `commands` field of the Local Search Request in this case contains only two items: a string giving the query in a format WAIS understands (usually boolean combinations of keywords) and an integer specifying the maximum number of results to return. These objects can be discriminated on the basis of type, so no other syntax is necessary. The Index server then uses the WAIS interface to submit a query.

After the results come back from the WAIS server, the headlines and scores are assembled into a Document List. Each document returned by WAIS is assigned an OID, and these OIDs are placed in the Document structures in the list. This list is then sent back to the return address specified in the Local Search Request. If at some later time the Client decides to download one of the documents, the OID stored in

the Document structure can be used to retrieve it.

### 6.3.2 Gateways to the Web

There are many good things about the World Wide Web. Certainly it is easy to use both from the naive end-user's perspective and from the perspective of a person who wants to write new client software. There is already an abundance of user friendly client software available on every platform, and part of the reason for this lies in the simplicity of the HTTP protocol.[2] Writing a program in Scheme to retrieve a web page from a server turns out to be a very simple process, assuming a support package for TCP connections is available. Another good feature of the Web is that a great number of gateways have been implemented which connect to other types of service. Because of the Web's high availability, organizations offering new services are encouraged to make their demos and products available via the Web. As a result of these factors, the Web can be a very powerful tool for connecting to a diverse set of services without having to know a bunch of new protocols.

The Web has its drawbacks as well. The fact that it is all based on a very simple-minded yet loosely specified protocol means that the web is much more wasteful of bandwidth than it ought to be. The fact that the names used in the Web are based on locations means that if a service moves a forwarding pointer may need to be followed to locate the service properly.

But the most immediately powerful aspect of the Web is also its biggest flaw as a universal infrastructure. Diverse services may be offered over the Web by using an interface specified using the HTML rich text format. This format makes it easy for a Web browser to produce attractive graphical pages representing the service, which is the primary reason for the Web's sudden success. But in order for a software agent to use the Web to access this diverse collection of services, these HTML pages must be mechanically interpreted. Since the pages are intended for display and subsequent interpretation by humans, it is a very frustrating task to write software to do the same job. For example, when a service such as Harvest is offered over the Web, the Harvest page will contain a "fill-in form" which allows the user to set certain

parameters, which are then packaged up into a query when the "Submit" button is pressed. Interpreting such a "form" is not a particularly feasible job for a computer, so when a gateway is constructed the format of the query is usually hard-coded into the gateway program.

The interpretation of the results of a search suffer from the same problems. Rather than getting a set of results encoded in some reasonable way, the results are returned in the form of another Web page, again using the HTML language. This HTML must the be mechanically interpreted and converted into data structures which can conveniently and efficiently store the information. Since the page was produced mechanically by the information server, this task is possible, but again practicality dictates that the gateway have hard-coded procedures for deciphering this output. There are a number of problems with a system that works in this way.

First, since the pages are designed and intended for human interpretation, the format could be changed drastically without causing the breakdown of the system from the perspective of the intended audience. However, any change in the format could easily cause the hard-coded interpreter programs to completely fail. In fact, some providers might even deliberately alter their pages in order to foil such mechanical translation. But even if the interpreter program could be made smart enough to withstand major format changes, the system is poor in terms of efficiency, and smarter interpreters will be even less efficient (and would probably still fail in some cases.)

Second, an objection can be made on the grounds of redundant functionality. By encoding the information in a mire of irrelevant data which must be stripped off at a later time, the system requires what are essentially equivalent (in this case, inverse) processes to be implemented on both ends of the communication. This would not be a problem, except that the implementations on either end are not officially related; they are independently coded and do not follow any agreed-upon protocol. The whole process is a guessing game to determine the specification of the protocol.

During the development of any protocol there are many issues to discuss – but putting them all aside the first issue here is simply that if a protocol actually existed

things would be much more workable. A number of models have been proposed to address this gap currently filled by HTML. These proposals, including Dexter[13], Aquanet[20], and the roles-based object system used in the MESH[30], define object models with associated means of generating graphical representations. By separating the mechanisms for displaying and using objects from the interface to the data itself, the display mechanisms can be modified and expanded without disrupting the interface to the data on which other mechanisms and users may depend. If HTML were replaced with such an object model, excercises such as those described in the next few sections would become largely unnecessary. A discussion of how the MESH roles model might be used to overcome these problems is given at the end of Section 6.3.4. There are many ways in which the development of such systems can be encouraged, but the Web as it currently stands tends to discourage all attempts in that direction. Given this state of affairs, we will now leave this topic and proceed with a description of the (somewhat unpleasant) techniques which we used with considerable success to interface to a number of services offered on the Web.

### A Quick Look at HTML

In order to retrieve and make sense of the Web pages which come back, we constructed a set of Scheme procedures which retrieve Web pages and refine them through a series of passes. Before we go into the details of how this processing is done, we will give a quick explanation of the syntax of HTML.

HTML, which stands for HyperText Markup Language, is based on the SGML standard. Version 2.0 of HTML is specified in [3], and version 3.0 is in draft form and should be coming out soon. Essentially an HTML document is raw text with interspersed "markers" which affect the way the text should be displayed, cause buttons and links to be drawn, or request that pictures be downloaded and included in the document. The markers in HTML are for the most part placed between matched pairs of angle brackets (< and >). There are many markers which we do not have to worry about because they are only important for getting the display right, but there are a few markers that we do need to know about.

The most important marker from our perspective is the anchor marker. The anchor marker causes a distinguishable link button to appear in the page. The anchor has two major parts: a URL which locates the data to which the link points, and a bit of unrestricted text which forms the distinguishable "button" which the user will press. The format of an anchor is:

<A HREF="[URL]"> [Button Text] </A>

Unfortunately, what makes this more difficult is that Web clients and Web servers must be "tolerant" of deviations from the specification, which makes parsing much less pleasant. Whenever a server starts giving out pages which have some minor peculiarity, the makers of client software tend to try to support it, because they want their software to work on as many servers as possible. For this reason, many servers deviate from the form shown above in various minor ways. For instance, it seems to be OK to leave off one or more of the quotation marks surrounding the URL.

| HTML Token | Function |
|---|---|
| <B> | Begin boldface text |
| </B> | End boldface text |
| <I> | Begin italic text |
| </I> | End italic text |
| <UL> | Underline begin |
| </UL> | Underline end |
| <LI> | Outline item begin |
| </LI> | Outline item end |
| <OL> | Sub-outline begin |
| </OL> | Sub-outline end |
| <P> | New Paragraph |

Table 6.2: HTML Formatting Markers

There are a number of other features of HTML which we found helpful in parsing pages. There are a number of markers for formatting which are easy to spot and which in a mechanically generated page will be used pretty consistently. These markers are shown in Table 6.2. Most of them come in matched pairs, with a starting and ending marker. The use of these markers is especially reliable because they all drastically

affect the look of the pages. An HTML programmer might leave off the </BODY> marker which is supposed to appear at the end of the "body" of the page, but no HTML programmer could leave off the </B> marker which stops printing boldface text!

### Preprocessing HTML

After the raw page is retrieved, some preprocessing must be done in order to make the job of parsing easier. First, the original string is broken up into words delimited by whitespace and the characters in the set {<, >, "}. After breaking the page into words, these words are put into a tree which has a structure defined by the placement of brackets and quotation marks. Any words bracketed between < and > or between quotation marks are stored in a new list which is then inserted as a single word. Lists that were delimited by brackets are tagged with the `HTML` symbol, while those that were delimited by quotation marks are tagged with the symbol `quot`.

During the next pass, each list tagged `HTML` is passed through a recognizer. If the tokens contained within can be interpreted, they are replaced by either a symbol representing the HTML token or by a link structure representing an HTML anchor. Any unrecognized tokens are expunged from the page. The Scheme symbols which are used to represent the HTML tokens are typically named the same as the token; for example, the symbol `/il` is used to represent the token </IL>. The link structure used to represent anchors is a list containing three elements: the symbol `link`, a URL string, and a title represented as a list of words.

After this pass, the page has been converted into a convenient intermediate form. The extraneous HTML junk has been removed, leaving only the raw text and the tokens which are specifically desirable for parsing. Typically a parser will filter the resulting list and remove all HTML symbols but the ones that are specifically necessary for its particular parsing method. For example, most of the parsers remove the header information from the beginning of the page. After the page has been put into this form, the process of parsing can start in earnest. The next section discusses the procedures we have implemented in order to support the quick construction of

parsers.

### Tools for Building Parsers

Table 6.3 lists several procedures which are useful for parsing Web pages once they have been preprocessed into a list of tokens. These tokens are either strings containing a single word, Scheme symbols representing HTML tokens, or link structures.

| tokens/... | Arguments expected; return type |
|---|---|
| filter | tokens, includes:list, include- blanks, include-links:boolean; list |
| strip-html | tokens:list; list |
| strip-blanks | tokens:list; list |
| find-first | tokens:list, is-a?:procedure; – |
| grab-from | tokens:list, until-proc, and-then:procedure; – |
| grab-from-multi | tokens:list, until-proc, and-then:procedure; – |

Table 6.3: Tools for Parser Construction

The first three procedures listed are used to filter the list of tokens. The last three procedures are used to step through the Web page in a very convenient way.

The `tokens/filter` procedure allows the caller to specify which types of tokens to keep and throws away everything else. The resulting expurgated list is returned to the caller. The caller specifies the set of token symbols to keep by putting them in a list and passing it in as the `includes` argument. The `include-blanks` and `include-links` arguments allow the caller to specify whether or not to keep blank lines and link structures respectively. During the preprocessing step, the end of every line is represented by an empty string in the list of tokens. Often these "blanks" need to be removed in order to make the parsing easier.

The `strip-blanks` and `strip-html` procedures take a list of tokens and filter out the specified tokens. The `strip-html` procedure removes all elements in the list of tokens which are not strings (i.e. the regular text). The `strip-blanks` procedure removes all blank strings from the input token list. Both procedures return new token lists containing all the elements that were not removed.

The next three procedures are used to actually parse the token lists. The procedure `find-first` takes a token list and a predicate. The predicate is applied to the elements in the list, starting at the front, and stops on the first element to which the predicate returns true. The `find-first` procedure then returns the sublist of the input list beginning at that point. In order to make it easier to provide the proper predicates, the `token/make-is-a` procedure will produce a predicate for recognizing any of the tokens that might be contained in the token list.

Essentially, the `find-first` procedure does a pretty simple job. Given a whole Web page or a portion of one, and given a predicate which recognizes a certain landmark, the `find-first` procedure will return the portion of the input tokens beginning with the landmark and continuing on from there. For example, the expression

```
(tokens/find-first the-page (token/make-is-a 'P))
```

will return the portion of `the-page` after and including the first `<P>` marker. This sort of processing turns out to be exactly what is needed to interpret Web pages. The interpretation process tends to work by spotting landmarks and pulling the information out of the right places. For example, part of the interpretation of the output of a Yahoo server depends on the fact that the number of matches to a query appears directly after the first `<P>` marker, except in the case that the number of matches exceeds the specified upper limit, in which case the total number of matches is the $13^{th}$ token after the first `<P>` and the actual number returned is the $17^{th}$ token after the first `<P>`. In any case, much of the work of the parsers we constructed is made simpler through the use of the `find-first` procedure.

The last two procedures are likewise very important to the easy construction of parsers. They essentially solve the same problem, and differ in a minute way which we will explain at the end of this section. The `grab-from` procedure takes three arguments. The `tokens` argument takes a list of tokens representing a Web page or a portion thereof. The until-proc procedure is a predicate which is passed a token, and is required to answer true or false. The procedure begins at the first element of the list of tokens and executes the `until-proc` predicate with the first token. If

`until-proc` returns true, the token list is split directly before the current token. We call the piece that contains everything before the current point the "grabbed" sublist, and we call the piece that contains everything after and including the current point the "rest". If `until-proc` returns false, the process repeats with the next element in the list of tokens.

When `until-proc` finally returns true, or if the entire list is scanned, the `and-then` procedure is called. This is an example of what is known as "continuation passing style". The term *continuation* refers to the process that accepts the result of an evaluation and continues the computation from that point. For example, the Scheme expression `(pp (+ 1 2))` calls the `+` procedure and passes the result to the `pp` procedure. In this case, when `+` is invoked, the continuation would be the process of invoking `pp` with the returned value. Scheme allows us to express that process as a procedure waiting for an argument, which in this case would be

```
(lambda (the-return-value)
  (pp the-return-value))
```

In the continuation passing style, we write a procedure which takes a continuation procedure as an argument. On completion of its task, the procedure invokes the continuation with the value that it would have returned. The main reason to use this style arises when we need a procedure to return multiple values. Continuation passing style provides an elegant solution to this problem because the values will be conveniently bound at the start of the continuation procedure.

In the case of the `grab-from` procedure, our need to return two lists of tokens suggested that the continuation passing style would be elegant. The `and-then` argument must therefore be a procedure which takes two arguments and continues the computation, utilizing the return values. When `grab-from` completes its task, it will call `and-then` with the "grabbed" sublist first and the "rest" second. The `and-then` procedure must then continue processing the page.

One time the `grab-from` procedure comes in handy is when text needs to be pulled out from between a matched pair of markers. For example, if text is boldfaced, it will

be set off by a preceding <B> and a following </B>. To find the first such boldfaced block in a given page, we use `find-first` to locate the first occurrence of <B>. Next we use `grab-from` to pull out the text until the first occurrence of </B>. The following code implements this:

```
(tokens/grab-from
  (tokens/find-first the-page (token/make-is-a 'B))
  (token/make-is-a '/B)
  (lambda (grabbed rest)
    ;At this point, grabbed is bound to the sublist
    ;beginning with the first 'B and ending right before
    ;the next '/B, and rest is bound to the remaining
    ;tokens...
    ))
```

The last procedure we will cover here is the `grab-from-multi` procedure. This procedure performs exactly the same function as `grab-from`, but takes a slightly different predicate procedure. Rather than the predicate acting on a single token, the predicate passed to `grab-from-multi` will be given a pointer to the rest of the list of tokens, starting with the current token. This allows the predicate to stop before a multi-token form rather than a particular single token. For example, using `grab-from-multi` one could grab tokens until the two word sequence `("hi" "mom")` is encountered, and so on.

Now that we have covered the basic strategy we apply in order to link up to Web sites, we will briefly describe several Web gateways which we implemented during the construction of our prototype system.

### 6.3.3 A "Yahoo" Gateway

Yahoo is an index of Web pages. The Web pages are selected by humans; that is, presumably only the more important pages are included in the index. The index is searchable over several domains, including the URL string, the title of the page, and a set of human-generated comments describing the content of the page. The interface is fairly standard, with a very primitive query language. A query consists of one or

more keywords separated by spaces, and interpreted one of three ways. The keyword list may be interpreted as a disjunction, a conjunction, or as a single string. This interpretation applies to the whole list, and cannot be restricted to any particular portion of the keyword list.[33]

Our gateway is a very simple Index server which calls a backend procedure to do the searching. This procedure is called `search-yahoo`. It takes eight arguments: a list of words, three boolean values selecting which portions of the index to search: the titles, the URLs, and the comments respectively, a boolean value specifying whether or not the words are case sensitive, a value `interp` which specifies the interpretation of the word list and must take as a value a symbol from the set {`string`, `and`, `or`}, a boolean value specifying whether or not the words are whole words, and an integer specifying the maximum number of matches to return.

When designing an Index server there is a difficult question of how much cleverness to install inside the Index server and how much to push out into the Referral network. Our policy was essentially based on an estimation of how generally necessary a given bit of processing was, combined with an estimate of the change in interface complexity. More universally necessary features which don't require excessive amounts of additional interface are good candidates for inclusion inside an Index server.

In the case of Yahoo, the simplicity of the query language is clearly an unpleasant aspect of the system, and some way of covering it up with the wrap of the Index server at first seems worthwhile. For example, the Index server could be implemented to accept an arbitrary boolean query, which would then be converted to a composition and sequence of actual Yahoo queries and run, completely transparent to the user. Although this at first looks to be a good idea, a problem arises when an error condition is detected in the Index server. This could be solved in a number of ways which essentially require a slightly more complex protocol between the Referral servers and the Index servers. Currently, time constraints have meant that nothing along these lines has been implemented, but there is a discussion of some of these possibilities in Chapter 8.

If an error condition is detected within the Index server, there is at the present

time no way for that message to be communicated. In order for such a system to work, the feasibility of the Search Request must be determined when it is created at the Referral, perhaps by sending a request to the Index server requesting that the given search be tested. In any case, in our implementation we skirted the issue by making the interface to the Index server closely follow the underlying interface, with only cosmetic differences.

The `commands` field for the Yahoo Index server is a list containing the standard LISP keyword-value pairs. There are several keywords which if present connote a true value for a boolean setting: `:title-index`, `:URL-index`, `:comments-index`, `:case-sens`, and `:whole-words`. The remaining three arguments do not need to be associated with keywords since they are of different types. These are arguments consist of a list of strings which describe the words for which to search, an integer which tells the maximum number of results to retrieve, and a symbol which tells how to interpret the list of words.

These values are passed directly to the `search-yahoo` procedure, which makes the HTTP request, gets the response back, and parses the results page into a list of links to matching Web pages. These links are then formed into a Document List which can be returned to the Client. Since the Web pages do not have OIDs associated with them, the Index server must invent a mapping of OIDs to URLs so that it can return OIDs to the Client and then later retrieve the right page if the Client asks for the document associated with a given OID. The easiest way to do this is to assign a new OID to each new URL that is matched.

The details of the parser implemented by the `search-yahoo` procedure are tedious and, worse yet, quickly outdated. The parser works using the procedures described in Section 6.3.2, so the essential structure of the parser can be deduced. In general, it works by locating landmarks and extracting nearby bits of useful information.

### 6.3.4 A Gateway to the Encyclopedia Britannica

Of all the services we connected to meshFind, the Encyclopedia is one of the most potentially useful. MIT has purchased a site license which allows members of the

MIT community to have electronic access to the Encyclopedia Britannica[1] and its index, provided through a Web forms interface. The Encyclopedia is a large store of general information and cultural knowledge, carefully indexed and organized by a large and experienced staff of humans. This gives it a great deal of potential as a tool for resolving queries. For example, unknown words can be looked up and the resulting report will contain references to Encyclopedia articles related to the topic as well as lists of synonyms and related topics.

The major factor which makes this difficult is once again the HTML report format. The service uses a WAIS back-end to do the searching, but unfortunately we do not have direct access to that. Because of these barriers, we have no option but to try to parse the HTML using the same techniques as we have used for other services. Because of the complexity of the reports, this requires several hundred lines of Scheme, but the design of the parser is essentially identical to the others.

After a page of the Encyclopedia index is parsed, the results need to be stored in a convenient representation. In order to do this, we devised a structure that holds an object which the Encyclopedists call a "heading". A heading is essentially an entry in the index, but due to the hierarchical design of the index the more general headings often contain subheadings. A given search will typically match several headings of varying levels of specificity. An upper limit on the number of headings to retrieve may be specified as part of the query submitted to the server.[7] Table 6.4 lists the fields of a heading.

The contents of the `eb-index-heading` structure are determined by the structure of the entries in the Encyclopedia. Figure 6-1 shows the Web interface used by the Encyclopedia's report generator. A typical query elicits a response which exhibits most of the features which our parsing software picks out.

The response to a query consists of a series of Headings. A Heading has some

[1] *Britannica Online* is a World Wide Web information service provided by Britannica Advanced Publishing, Inc., an Encyclopaedia Britannica, Inc. company. *Britannica Online* version 1.2 is Copyright ©1995 by Encyclopaedia Britannica, Inc. *Encyclopaedia Britannica*, Fifteenth Edition, is Copyright ©1974-1995 by Encyclopaedia Britannica, Inc. All rights reserved. "*Britannica Online*" is a service mark, and "Britannica", "Britannica Online", "Macropaedia", and "Micropaedia" are trademarks of Encyclopaedia Britannica, Inc.

| Structure eb-index-heading | |
|---|---|
| heading | token list |
| synonyms | list of token lists |
| notation | token list |
| see | optional X-Ref link |
| heading-link | optional link for whole heading |
| entries | list of single entries |
| subheadings | list of subheadings |
| major-refs | list of links to major references |
| micropaedia | optional link to "micropaedia" |
| macropaedia | optional link to "macropaedia" |
| list-related | list of links to related subjects |
| see-also | list of links to "see also" X-refs |

Table 6.4: The Structure of an Index Heading

text which describes it, and often includes a link to an article which covers the whole heading. The text describing the Heading is stored as a list of tokens in the `heading` field of our Heading structure, and if a link is present it is stored in the `heading-link` field. If no heading link is present, the `heading-link` field is set to false. Often the text describing the Heading includes the word "or", followed by a comma separated list of synonyms. Any synonyms appearing there are collected into a list of token lists and are placed in the `synonyms` field. The text describing the Heading often includes a notation describing the type or category of the heading. This notation is set off by parentheses, and is stored as a list of tokens in the `notation` field of the structure.

For example, the screen shot in Figure 6-1 shows one example of a Heading. Note that the underlined portions of text are hyperlinks to portions of articles. This Heading describes the "central processing unit" of a computer, and this title is displayed directly after the round bullet. The italicized text "or" sets off the synonym "CPU". Our parser places this synonym in the `synonyms` field. If a notation had appeared as part of this heading, it would appear at the end of the title line, set off by parentheses.

The `see` field of the Heading structure is only used when the Heading is a simple cross-reference. In such cases, no additional information is stored in the Heading apart from the text describing the Heading and the link representing the cross-reference. In

these cases the cross-reference link is stored in the `see` field and the remaining fields are set to false. If the Heading is not a simple cross-reference then the `see` field is set to false and the remaining fields store the data associated with the Heading.

A Heading typically contains several entries which are displayed indented and bulleted. Some of these entries are "single entries" which contain only one reference, while others are "subheadings" which signal the beginning of a set of related entries indented one level deeper. All entries contain a list of words describing the heading, and most contain a link to a related article. In the case of single entries, the whole entry is packed into a link structure, and a list containing all single entries within a Heading is stored in the `entries` field of the Heading structure. Subheadings are processed recursively in much the same way as the original Heading. The resulting `eb-index-heading` structures are collected into a list and stored in the `subheadings` field of the Heading which contains them.

For example, the heading shown in Figure 6-1 contains three single entries and a subheading following the title line of the Heading. In this case, the first two lines are single entries; each of these includes a link to an article. The five lines after that comprise a subheading titled "use in". This subheading contains five references, two on the first line and one on each of the following four lines. Each of these is a single entry of the "use in" subheading, and each links to part of an article. Following the "use in" subheading there is one more single entry. Some of the more complex Headings have many subheading, and some subheadings contain further subheadings. Typically, however, Headings tend to be fairly simple.

The remaining five fields in the Heading structure collect a number of other types of information which appear in the list of entries which form the body of a Heading. This information is typically set off by a certain keyword which is set in italics. Information such as "major references", optional cross-references set off by "See also", and lists of related topics are a few of these additional links.

Some entries contain a reference set off by the italicized text "Major Ref.". These references link to related topics about which there is significant coverage. Entries following that form are collected and placed in a list stored in the `major-refs` field.

The electronic version of the Encyclopedia Britannica is organized in several parallel volumes. Along with the vast body of indexed articles, there are parallel sets of articles which discuss the topics in varying levels of detail, and which are organized along different lines. These volumes are entitled the *Micropaedia* and the *Macropaedia.* The index contains links to discussion in these volumes which are separate from the links to the articles in the body of the Encyclopedia, and these links are set off by the italicized notations "Micropaedia" and "Macropaedia" respectively. Links of these sorts are collected into lists and are stored in the `macropaedia` and `micropaedia` fields of the Heading structure.

Some Headings contain an entry set off with the italicized phrase "For a list of related subjects, see". These entries are parsed and the links following the introductory phrase are collected and stored in the `list-related` field of the Heading structure. Similarly, links preceded by the italics "See also" are collected and stored in the `see-also` field.

For example, in Figure 6-1, the last line in the "central processing unit" Heading is a cross-reference to "arithmetic/logic unit". Our parser places this cross-reference in the `see-also` field.

Once the report has been parsed, the much more difficult process of actually making use of the data can begin. The form of the report does not necessarily fit well into the model of an Index server, since the report includes a great deal of semantics beyond that of a plain list of documents. Certainly an Index server can be built around the Encyclopedia fairly trivially by making the query and blindly reporting all the links that come back. But this seems to be a shame because in doing this we are throwing away so much useful information. When we implemented the Index server which served Encyclopedia information, we decided to make the result a Document List in which the first document is the HTML page directly from the Encyclopedia Web server, and the rest of the list enumerates the links available from that page. This puts the burden of interpretation on the Client, who should be capable of handling it, especially given a decent browser program.

Many non-human clients may also want access to the Encyclopedia. For example,

a Referral server could make very good use of the information in the Encyclopedia to improve its ability to refine and direct the search. But in order to do so effectively the relevant information must be extracted from the Encyclopedia report using techniques analogous those specified above. This problem can be solved in a number of ways, and it is difficult to choose one method over another. One way would be to extend the protocol between Index servers and Referral servers so that arbitrary data objects could be passed back as results. This extension may not be a bad idea in general, given that as time goes on the nature of the documents being retrieved may change. It is possible that a uniform system of object types and "roles" may emerge. Such a system might rely on the use of transmitted code in order to provide implementations of the roles on different platforms. Should such a system arise it could form the basis of the document transmission process we are discussing here, and it would certainly be powerful enough to handle the transmission of the Heading structures described above. It is interesting to theorize that if the Web implemented such an object system as part of its protocol, the whole parsing mess might be elegantly avoided.

There are other less elegant solutions to the problem as well. For example, the parse code might be duplicated in the Referral server so that it may parse the HTML page again and then try to make sense out of it. This idea, while it is both easy to implement and also based on an existing protocol "standard", and hence fairly consistent with the Web philosophy, maintains this consistency to the point of being very short-sighted. This unquestionably works as a quick implementation, but it runs contrary to the principle of the Index server and in doing so leads to redundancy and decreased stability in the face of changes to the network. Because the parsing code is redundantly implemented, if it becomes necessary to change the parser (which, as we have discussed previously, is not an unlikely eventuality) the job will be much less manageable. Furthermore, the perpetuation of the HTML protocol is ill-advised.

Despite all of these reasons, our implementation presently uses this second technique as a temporary measure. The MESH project is completing a stable implementation of an object system based on *roles*, and when that system is more complete perhaps an attempt can be made to implement the scheme outlined above. The in-

tention of an object model with roles is to formalize the concept of "methods" used in traditional object models in a more general fashion. Each object "plays" a certain set of roles, but the particular roles an object plays are decoupled from the definition of the object itself as much as possible. Part of a roles system is the provision of mechanisms for packaging up the implementations of each role for each object class and for disseminating these implementations to the machines which must interpret the object. An equally important part of such a system is the dissemination of the implementation code to the machines that need it, and the services which a machine can use to locate the proper implementations. A good roles system should increase the longevity of data by making it easier to maintain current implementations of interface procedures, thus making it easier to interface to the object's fixed underlying bit representation.

Once a roles system is in place, the Index server could return an object in the Document List which represents the data refined from the Web pages by our parser. When the Document List is received by the Client or by a Referral server, the objects it contain must be interpreted and displayed or used, respectively. It is then that the benefits of a roles-based system come to fruition: a set of roles could be implemented to pull useful information out of the Heading representation generated by our parser. If the recipient of the object did not have a given role implementation available, or did not have the right version, various location services could be used to locate it.

### 6.3.5 A Gateway to Harvest

#### A Harvest Primer

Harvest is a system being developed by the IRTF-RD, or Internet Research Task Force Research Group on Resource Discovery. The Harvest system is a distributed information system that aids in resource discovery by collecting and indexing references to data from many sources. There are four types of servers in a Harvest system: Gatherers, Brokers, Caches and Replicators. The Gatherers are processes running local to the archive that stores the data to be indexed. These Gatherers generate

object summaries of any interesting data objects and return these to a Broker using an efficient bulk transfer protocol.

Each Broker accepts summaries from a given set of Gatherers, and uses an underlying Index Subsystem to store and incrementally index the summaries. This Index Subsystem may be implemented using a variety of index/search engines, including WAIS, Glimpse, and Nebula. According to the design, the only requirements placed upon an index engine are that the search capability support arbitrary boolean queries and that the index can be built with incremental updates.

Brokers also provide a query interface which allows users to search their index. The language used to formulate a query is determined by the implementation of the underlying Index Subsystem. The query language is guaranteed to support boolean combinations of keywords, and may optionally support such extensions as regular expressions and approximate queries. The results of the search process are relayed to the user in the form of a report listing the summaries for the objects which most closely fit the query. When documents located through Harvest are downloaded, the download request is simultaneously sent to both the Harvest Object Caches and to the archive which stores it. As soon as one of the servers responds, that server is used to download the object and the connections to the other servers are closed.

A growing number of Harvest servers have been established around the world. A partial listing of these can be found at the Harvest Server Registry, which is accessible from the Harvest Home Page on the Web. The Web interface to a Broker typically consists of a page with a Forms interface into which the user may enter a query and select some options which control the specificity of the search, most of which are specific to the structure of the Index and the implementation of the search engine. After the Form is submitted and processed, Harvest returns a page listing links and summaries of any documents which matched the query. Following the links downloads the documents from the nearest cache or from the archive if necessary.[5][6]

### Building a Scheme Interface to Harvest

Building a gateway to Harvest turns out to be a fairly simple operation, and our code did not differ remarkably from the code we implemented to link to Yahoo. Similar to Yahoo, the results returned by a Harvest server are organized as a list of references. Unfortunately, every Harvest broker formats its reports slightly differently, so a general-purpose Harvest report parser fails to do a complete job. In order to deal with this problem, we implemented a general `search-harvest` procedure which takes as one of its arguments a procedure which performs actions specific to the broker being contacted. This procedure includes code which connects to the broker and also some code which handles certain parts of the parsing process which tend to vary from broker to broker.

The `search-harvest` procedure takes several arguments that specify the search to perform. The first argument, `server`, provides a procedure to handle the bits of the parser that are specific to the broker to be searched. In our prototype we implemented two procedures which specify brokers in this way: the `harvest/sc-tr-server` procedure specifies a Broker that serves computer science technical reports, and the `harvest/mit-web-server` procedure specifies a broker that indexes home pages at MIT. The second argument, `query-string`, specifies the text of the Harvest query. The structure of a Harvest query depends on the interface provided by the Indexing subsystem underlying the Broker, but in general most queries are structured as boolean combinations of keywords.

The next two parameters to the `search-harvest` procedure are `case-insens?` and `whole-word?`. These parameters specify whether the search engine should treat the keywords in the query as case sensitive and as whole words, respectively. The fifth

| Structure `harvest-match` | |
|---|---|
| `doc-link` | Link |
| `index-data-link` | Link |
| `matched-lines` | list |

Table 6.5: The Structure of an Harvest Match

parameter, `num-errors`, is used to specify the number of spelling errors allowed in the process of finding matches to a query. The last parameter, `max-results`, specifies the maximum number of documents to list in the response.

### Parsing the Output of Harvest

Using the `search-harvest` procedure, a Harvest server can be searched, and the resulting report is parsed and stored as a list of `harvest-match` structures. The contents of a Match structure are shown in Table 6.5.

Typically the reports generated by a Harvest server are a series of paragraphs, each corresponding to one result document and containing several related links. In most cases, there is one link to the document being referenced, another link to a SOIF (Summary Object Interchange Format) document that summarizes the document being referenced, and a number of lines of text excerpted from the summary that contain the matching keywords. Some Brokers supply all of this information, while others provide only parts of it. Often a Broker will provide additional settings used to cause more or less lengthy report styles; for example, a Broker might give an option which selects whether the summary objects are directly included in the report, included in the form of links, or left out entirely. In such a case, the setting should indicate that the objects should be put in as links.

The parser interprets these reports using techniques identical to those we have discussed before, and converts each paragraph of the report from Harvest into a Harvest Match structure. The link to the actual document is stored in the `doc-link` field. The link to the related SOIF object, if such a link exists, is stored in the `index-data-link`, which is otherwise set to false. Any lines showing matches that have been discovered are collected into a list of token lists and are stored in the `matched-lines` field.

### Constructing an Index Server for Harvest

Because the ideas behind the Broker are in many ways analogous to the ideas behind an Index server, it is natural to construct an Index server which performs Harvest

searches. The format of the `commands` field can be structured similarly to many other Index server implementations; a list with LISP-style keyword-value pairs can be used. The keywords `:case-sens` and `:whole-words` may be used to specify values which will be sent to the `search-harvest` procedure. If the query is represented as a string and the maximum number of results is given as an integer, those values may be placed in the `commands` list and can be distinguished by type.

After the query has been submitted and the results come back, much of the information in the summary can be reformatted and added to the `headline` field of the Document structures that will be returned, and a table mapping OIDs to URLs can be built so that OIDs referring to result documents can be returned in lieu of returning the whole document. Since Harvest automatically provides object caching, a further cache at the Index server is not as necessary.

## 6.4 Summary

This concludes the description of Index and Referral servers. In this Chapter, we began by explaining the protocol implemented by Referral servers. This protocol has a kernel of required behaviors and also has a number of possible extensions which may be optionally implemented. Next, we covered the protocol implemented by Index servers in a parallel way. We then began the section which forms the bulk of the chapter and describes how Index servers have been constructed to form gateways between meshFind and other Internet services. This section begins with a subsection describing the gateway to WAIS. The next subsection discusses the benefits and pitfalls associated with trying to construct gateways to Web-based services. This subsection also contains a great deal of material describing the parsing techniques we developed for constructing Web gateways. The third subsection explains the construction of the gateway to the Yahoo service. The fourth subsection describes a gateway to the encyclopedia Britannica server. The fifth subsection describes several gateways constructed to various Harvest servers.

The next chapter discusses some issues involved with constructing user interfaces

for meshFind, thus completing the presentation of the design of meshFind. The last chapter offers a summary, followed by some conclusions and some ideas about future work.

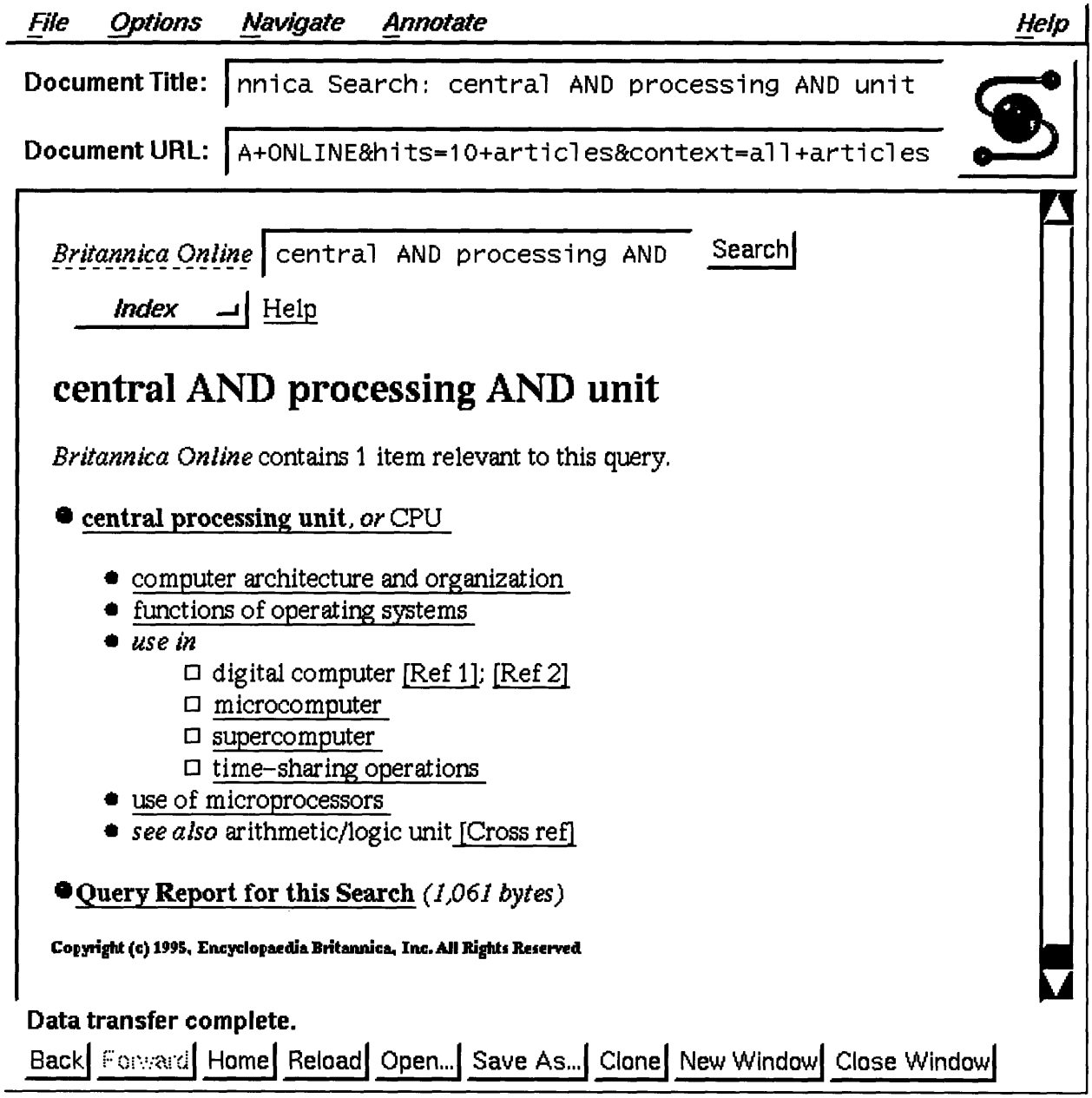


Figure 6-1: The WWW Interface of the Encyclopedia Britannica Index.

# Chapter 7

# Client Software

Client software for the meshFind system can be implemented in a number of ways. One of the initial design objectives was to produce a system that needed neither a powerful machine nor a powerful network connection to allow access to the system. This objective has been maintained to the point that a user interface to meshFind requires only a VT100 and the Client software can connect to the Sponsor using a modem. Sponsors may even offer Client software on their machine, so that users can dial in with a modem and access meshFind without running any special software.

The exact nature of the Client software is largely dependent on the nature of the existing network infrastructure. For example, the existence of a widely accepted object system and a widely accepted system for running foreign code (such as Hot Java[12] from Sun, perhaps) would affect many of the multimedia aspects of the Client software. Documents that in some sense handle their own display and decoding make browsers easier to write, whereas a document written in one of the many rich text format languages might require that a browser understand that language so that it may be displayed. Many of these issues are still largely undetermined, and different browsers choose different solutions to the problem: at the present time, the Mosaic Web browser relies on locally available software such as *XV* to display anything that is not an HTML page, while Netscape implements its own set of display tools covering most of the common object formats. Neither of these methods can cope smoothly with a new format, since in either case some new software must be manually located

and installed.

Because of the complexity and undetermined nature of the problem, we chose to simplify matters in our prototype of meshFind by dealing exclusively with text documents and text queries. Even given this limited scope, the Client software might still be fairly complex. Any number of helpful features might be implemented, such as methods of keeping track of documents which come back, methods of keeping track of revisions to the query, on-line help pages, and, most importantly, pre-written Filter programs. Although these features are not necessary to the use of the meshFind system, they would make it much easier to use.

In this chapter we will describe two implementations of Client software, one implemented for a text display terminal, and one implemented as a Web server so that several users may have access through a regular Web browser.

## 7.1 Text-Based Client Software

In our discussion of the Sponsor's protocol in Chapter 5, we saw the entirety of the Client's interface. The interface is fairly simple, despite keeping state on both sides. A Client composes a Query off-line, and then requests a session by sending that Query to a Sponsor. If the Sponsor's reply is positive then the session has been initiated. The Client must then wait for documents to be sent back.

### 7.1.1 A Question Editor

One place to start this design is with the display, entry, and editing of a Query. There needs to be a good Query editing tool that shows all the parts of a Query in summary form and allows the user to zoom in and edit any of them. This tool will be used before the connection is made, as well as being called upon whenever new information from the search leads to a revision of the query. Whenever it becomes necessary to display the Query this tool will also be used, perhaps in a read-only mode. So now we need to figure out what should appear, and how.

Fields of the Query such as `search-id`, `revision`, `to-addr`, and `return-addr`

may be displayed, but their contents must be set up by the Client software in some special way. The addresses, for example, really depend on the connection that exists between the Sponsor (sometimes there may be a choice of Sponsors) and the Client. In the case of point-to-point communications platforms such as modems there will be at best a selection of different Sponsors to connect, but if the Client's hardware can support TCP, the Client may essentially be asked for an address of a Sponsor. The Client's address should always be known.

The Revision number and the Search ID should be set by the software and should not be edited. The Search ID must be an OID, and as such must come from a server of OIDs. The current protocol requires the Client to supply a new OID in the `search-id` field of the Query that requests a search. An extension of the Client/Sponsor protocol could place the burden of inventing an OID on the Sponsor. When the Sponsor receives a Query with a value of false in the `search-id` field, it could interpret that object as a request for sponsorship, and in the event of a positive response could assign a new OID to the message that is sent back. The Revision number starts at one for any new search, and increments only when the Query is modified and sent out to the Sponsor.

The `max-duration-avail` and `max-cash-avail` fields are small fields which the user may set. The `security-stuff` field is reserved for authenticators and certificates and such. These items are not entered by the user, but perhaps the user has some involvement with setting up the contents of this field. The `filter` field will probably be set in most cases to one of a selection of pre-written Filters, or perhaps a number of Filters which have been stored on disk. In any case, writing a Filter requires testing and debugging, and is not something that is just written quickly and sent. Quickly *modifying* existing Filters would be a more believable scenario. As for setting the value of the `filter` field, selecting from pre-set choices or choosing a file is a simple interface, but designing a good Filter editor is much more difficult. For the time being, let us assume that the Filter already exists.

The Question itself must also be displayed and edited. The Question is a field which might take up a larger amount of room on the screen, and will probably need

to be summarized or truncated in some way. Related to the Question are a list of Amendments and a list of Similar and of Dissimilar documents. The latter lists can be shown as truncated lists of headlines, each of which can be expanded into a full-screen edit of any of the listed documents. The Question can be shown, truncated and wrapped into a box smaller than the screen. Since the Amendments represent a series of changes as the Query moves from one Revision to the next, the Amendments may be shown by placing previous versions of the Question "in back of" the Question on the screen. The Client can then flip through the older versions of the Query.

### 7.1.2 Displaying New Documents

The Client then indicates that a new search is desired, and a blank Query appears on the screen. The Client uses the Question editor to fill the details of the Query into the blank Query. When the Query has been filled in, the Client indicates that it is OK to send, and the Query is sent to the selected Sponsor. If it is accepted, the acceptance comes back, and the Client software goes into a waiting mode. Otherwise, the Client modifies the query and/or sends it to a different Sponsor.

Eventually, the search will result in some documents being discovered and sent to the Client. When the software receives the new documents, it makes a noise to get attention, and displays the current list of retrieved documents, with the new documents added in at the bottom. One good way to implement this would be to mimic a mail program such as Eudora. Each document might be placed on a line, with the headline and other important information taking the place of the subject lines in Eudora. In order to prevent the list from being overloaded, the documents in each bundle might be stored "inside" their bundle; that is, each of the bundles appears on a line and by selecting a bundle the documents inside can be viewed, similarly to the way subdirectories are dealt with in the Finder.

When a document in the list has been selected, a number of commands could be acted on it. The document might be marked for full-text retrieval. After a bunch of documents are marked, the request to retrieve all of the marked documents could be sent by invoking a separate command. If a document has been retrieved, the

full text can be viewed in a screen editor or printed. A document may be added to the Similar or to the Dissimilar list of the Query. A document may also be marked as being part of the solution. When the Query is officially considered solved, those documents marked as being part of the solution will be collected and included in the Solved Query object.

### 7.1.3 Commands

The Client can switch modes screens by using the function keys which are continuously active. The function keys provide a number of functions which are listed in Table 7.1. These keys are approximate, since the exact mapping should depend on the topology of the keyboard, etc. But the basic idea is there: the first six commands work at practically any time (to be accurate, they work any time between atomic operations.)

| Function Keys | |
|---|---|
| `f1` | Help |
| `f3` | New Search |
| `f4` | Complete Search |
| `f5` | Edit Query |
| `f6` | Jump to Document List |
| `f8` | Print |
| `f13` | OK/Select/Yes |
| `f14` | Cancel/No |

Table 7.1: Function Keys

The Edit Query function key and the Jump to Document List function key allow the Client to move around the program by popping into the Query editor and the Document List respectively. The New Search and Complete Search function keys allow the Client to cancel or to complete the search. In the case of a completed search, the documents marked as solutions at the time of completion will be included in the Solved Query object.

### 7.1.4 Text-Based Applications

One of the major applications for the meshFind system, and especially for text-based Client software, is the modernization of existing library systems. The computer equipment currently owned by most libraries is of the mainframe and terminal variety, and cannot run complex graphical software. Furthermore, the network bandwidth required to run several Web clients is very high and at the present time much too expensive for most libraries. Furthermore, Web terminals tend to be very popular.

These and many other arguments concerning the modernization of library systems through the use of a system such as meshFind are presented in a paper [10] I wrote as an independent research project. The meshFind system can be run using the equipment existing at many library sites, or requiring at most a small amount of additional hardware. Because it uses network resources lightly, the network connection required to connect it to the outside world need not be that expensive. Many libraries already have some kind of connection, and in those cases the implementation of meshFind might not involve a significant increase in cost.

## 7.2 Web-Based Client Software

The Client software can also be implemented through the Web. In such a case, a Web server would be set up that allows any Client to use a standard Web browser to perform a search using meshFind. Unfortunately, since the meshFind protocols are not stateless this does not work as smoothly as would an interface specifically designed to talk to meshFind such as the text-based one described above. However, using the Web the Client does not need any special software apart from the standard Web and TCP software.

A Web-based interface to meshFind works using the Forms interface. The Client downloads a Query submission Form from a well-known source, and fills in the data. The Form has slots for the question, has radio buttons for a selection of pre-written Filters, and has fill-in boxes for amounts of time and money to spend. Again, the security-related information is not yet planned, since at the present time the security

infrastructure of the Web is being developed.[32] When the submit button is clicked, the Form is submitted to the server which manages the Client interface, a Query object is constructed and sent to the appropriate Sponsor. The Sponsor treats the Web server as the Client, and the Web server relays any results to the end-user.

A problem arises because the Web server has no means of initiating communication with the end-user. This means that in order for the end-user to find out how the search is progressing, he must poll the Web server and get back a response. This can be made convenient by including a button which links to this action; however, this is significantly less elegant than notifying the end-user of new developments in the progress of the search, and will inevitably lead to delays in processing the search.

In short, interfacing via the Web has some serious defects and does not allow the user to gain the complete set of benefits which could be gained from the use of the meshFind system. This problem can be solved only by implementing local software which listens for messages from the Sponsor. If meshFind became popular, it would encourage the implementation of such software across a variety of platforms, but the lack of this software will not help to increase the popularity of meshFind.

## 7.3 Conclusion

In conclusion, the requirements for implementing a Client interface for the meshFind system are not very difficult to achieve. A Client need not have a complex graphical user interface, and if it does, a reasonable implementation could be built on top of existing Web browser implementations. We saw that the minimal requirements for a Client program can be implemented with hardware no more complex than a serial terminal and one or two modern PC's, making it more feasible to install the system into the existing public information infrastructure (i.e. the public library system).

We also saw that graphical browsers are a possibility, building on top of the existing Web browser implementations. In order to have a really effective graphical browser, it might be necessary to write an additional application to listen to the Sponsor. This program would wait for a message from the Sponsor's Webserver

that flags the arrival of new data for the Client to inspect, and would then indicate visually that the user should ping the Webserver and review the incoming data. Such a program would be relatively simple to implement, probably on the order of the *XBiff* program.

# Chapter 8

# Conclusions and Future Work

The vision of this thesis subtended a problem much too large to attack in the context of a Master's thesis. As a result, this document contains two almost orthogonal threads of discussion: one which discusses the global implications and the many difficult problems associated with the development of a global indexing and resource discovery system, and another which discusses in some detail a low-order approximation to a protocol and architecture for such a system. The two threads tend to be fairly mingled throughout the document, but in general, Chapter 5 and Chapter 6 tend to be those most concentrated on the implementation of the protocol, while Chapter 1, Chapter 2 and this Chapter are those most concentrated on the global perspective. In this Chapter, we will conclude each thread separately. We will begin by presenting a summary covering the design and implementation of the meshFind system. After the summary, the remaining sections of this Chapter will conclude the discussion of the global perspective by discussing possibilities for Referral network and for future work in general.

## 8.1 Summary of the Implementation

In Chapter 3, we saw a fairly complete overview of the structure of the meshFind system, leading up to an example of its operation. We learned that there are three types of servers in the system, Sponsor servers, Referral servers, and Index servers.

A Client initiates a session with a Sponsor server using interface software local to the Client's machine, and the Client sends a Query object to the Sponsor. The Sponsor then attempts to answer the query by forwarding it to various Referral and Index servers. The mechanics of this process is controlled by a small Filter program which the Client provides with the Query and which is run on the Sponsor's machine to determine the plan of attack to follow.

Chapter 4 begins a more specific description of the protocol. In this Chapter, a description is given of the data structures and messages that are passed among the servers in the system. In our implementation, the messages used in the protocol are arranged in a hierarchy of objects which inherit from their parents. This is not intrinsic to the structure of the protocol, but comes in handy for our implementation. The discussion of the protocol in this Chapter also details the problems and benefits of a stateful protocol, and explains the design choices made with regard to giving meshFind a stateful protocol. There is a also great deal of exposition in the Chapter covering the ideas behind the more complex structures, specifically Address structures and Filter objects. Address structures are used to describe servers in the system, and depend on the MESH infrastructure for naming and location. Filters are pieces of portable code which can be run safely on remote machines. They are based on the language Scheme and are interpreted in a safe environment. The ideas involved with Filters might be implemented using other systems for portable code, such as Hot Java[12] or Olin Shivers' Scheme-based Webserver[28], but given that none of these were convenient to the MESH implementation, we chose to implement our own simple portable code environment for meshFind.

Chapter 5 describes the structure, operation, and implementation of the Sponsor server. The Chapter begins with an overview of the operation of the Sponsor, followed by a description of the structure of our implementation. The Chapter concludes with a succinct description of the protocol followed by a Sponsor. The role of the Sponsor is to manage a Client's search through a complex network of Referral servers and Index servers. This management process is controlled by the Client's Filter program, giving the Client indirect control over how money is spent and over which services are used.

The structure of our implementation is formed of several interrelated modules which handle various parts of the job. Our Sponsor server is designed to handle many Clients concurrently, so there is a great deal of opportunity to improve efficiency through clever scheduling and task management. Although our implementation does not do so, a Sponsor server would also likely include a Web server so that Clients could use the Web to initiate and interact with sessions on the Sponsor.

Chapter 6 covers issues relating to Referral and Index servers. The Chapter begins by describing the protocols and implementation of the Referral server, then gives a parallel description of the Index server. These descriptions include both discussions of the protocols that were implemented and discussions of possible extensions to the protocol which could potentially increase the efficiency and power of the system. The rest of the Chapter then describes the gateways we implemented to connect meshFind to several information services offered on the Internet. Much of this description is devoted to services available through the Web, and includes a section which discusses the merits and flaws of the Web system and the HTML standard. The gateways implemented include the WAIS system, Harvest, Yahoo, and the Index of the Encyclopedia Britannica.

Chapter 7 discusses the requirements of Client software for accessing the meshFind system. Most of the exposition is devoted to the minimal interface required, which can be implemented using such antique hardware as serial terminals. Such an interface might be provided publicly in the Library system without too many changes to the current systems in place and without excessive spending for new hardware. The possibilities for using Web client software as an interface is also covered, although much more briefly. The reason for this is that the nature of the Web protocols is not entirely conducive to an interface to Sponsor servers, since the protocol between the Sponsor and the Client is not stateless. A possible work-around to this problem is given in the conclusion to the Chapter.

We will now finish this Chapter by concluding the other thread of discussion in this thesis, namely the overall vision of the meshFind system. We will first discuss a few of the shortcomings of the protocol which were discovered during the implementation

process, and finally conclude with a discussion of some ideas for structuring the Referral network.

## 8.2 Future Modifications to the Protocol

As is the case with most first implementations of systems, many of the lessons learned in the development of this system lead to the conclusion that the system should be redesigned. The part that needs the most redesigning are the messages sent between servers. As they are presently set up they are reasonably elegant and cover the ground pretty well, but in the course of implementation we discovered ourselves losing some of the elegance of the original design in an effort to address a number of unexpected developments. For example, the Query object contains the information about money and time available, and the `security-stuff` field to hold various authenticators and certificates, but we ended up needing an additional simple message which adds more money and time on the fly. This section presents a number of ideas for improving the protocol.

First, Queries need finer-toothed revisions. Rather than each new revision causing the search to restart, minor modifications should send a minor revision which continues from the current point with the new revision. We think of the metaphor as similar to revision `X.Y`, where `Y` is the minor revision number and `X` is the major revision number. Whenever `X` increments, the search is begun again, and whenever `Y` increments the search continues with the new information. This can be used to eliminate the simple message to add more time and money, since this can be done with a new minor revision. Also, if the time and money fields should be amounts added to the current total, so if they are left blank none is added. While we are at it, we should include a field which increases the number of evals allocated, since that was left out of the Query structure.

Second, support for the direct return of data to the Client should be provided. This can be immediately supported by hacking the return address on a Local Search Request to point to the Client rather that to the Sponsor. The difficulty with this

protocol is that the Sponsor needs to know that the search has been completed. This leads to a new idea in the construction of the Address structure: a list of sites to notify when the item has been sent.

Third, as we discussed in Chapter 6, the implementation of an object system with Roles would be a good way to improve the interface to the Index server, and would not only make it possible to return documents which were not text, but also make a much better interface to the data possible for communication between Referral servers and Index servers. Presently, the interface between Referral and Index servers is very primitive, and fails in many ways to provide a scalable and powerful interface. Apart from the question of using objects with Roles, there needs to be a way for a Referral to test out an Index server before suggesting it. Perhaps a Referral could send a Request on a trial basis and the Index server would then rate the success of the search (i.e. how many documents were located). The Referral could then see if the search would result in the retrieval of too many documents (in which case the Request must be narrowed) or too few documents.

The fourth problem we discovered was with servers that charge rates based on usage rather than a flat fee. When we designed this system we tried not to get too tangled up in the details of fees so that we could concentrate on the other details. Unfortunately, in the process of ignoring the details of fees, we made an overly restrictive assumption. We assumed that each server in the system would quote a specific price for a given service before providing the service. Using such a model, it would be impossible for a service to charge a rate per unit time, effectively varying their flat rate based on the amount of time the search actually takes, or conversely, based on how long the Client wants the search to continue. We had imagined that other cost schedules would be easy to add to the system, and had then gone ahead to other areas of the project, but it turned out not to be that simple. The trouble arises when the Filter chooses to use a given service: if the service charges a variable rate, the Filter must specify how much of the service to purchase, but this functionality has not been included in the Filter interface. Furthermore, the Filter may need to tell a service to stop processing early, which is currently another function Filters cannot

perform. By revising the Filter interface and by making some modifications to the Server Info structure, the cost schedules of services in meshFind could be diversified.

The final problem we discovered was that unless the Client has software running on his machine, the Client software might as well be a part of the Sponsor. This conclusion was drawn from the two interfaces we discussed in Chapter 7. In both the text-based interface which connected over a modem and the Web-based interface which was served by a special Web server, the Client software is not running on the Client's machine, but on a remote machine which serves a number of different end-users. This leads to the obvious simplification that makes the Sponsor have the "Client software" built in, that is, make the Sponsor provide a more direct Client interface. This makes a certain amount of sense, and greatly simplifies the picture of the system. In the case of the Web, the Sponsor would operate a Web server which serves the Clients on one end, and talks to the many Referral servers and Index servers on the other end. The difference is that the Client does not need any special software; of course, with a sufficiently powerful object system this is no longer a problem.

## 8.3 Possibilities for the Referral Network

One thing we learned in the process of doing this work was that when a really hard problem is being tackled, many attempts at solutions tend to just push the hard parts into a different part of the problem. Figuring out exactly what the problem is and really cornering it is a very difficult thing to do. In the case of meshFind, the central problem we would like to solve is a kind of knowledge representation and knowledge organization problem. We want to build some tools which know where to find resources and can determine from a question (while we're at it, say the question is in English) where to look and what to ask each of the resources. In the course of our design, the meat of this problem kept getting pushed around until it finally ended up being the responsibility of a network of Referral servers.

We stuck with this decision because it seemed to have some good qualities. First, it is a distributed and heterogeneous system. While the servers all follow similar

protocol conventions, in a large system there would not be just one "General-purpose" Referral – there would probably be many copies of several popular brands, and many more Referrals maintained for the use of specific groups of people or localities. For example, a public library might maintain its own Referral server for use by its patrons. The library would probably start with a commercial or free implementation and then modify it according to its own needs.

Second, the production and maintenance of Referral servers has positive qualities from an economic perspective. The production of ever more powerful "General" Referral servers could become a very competitive market. New servers might be able to interpret Queries with increasing effectiveness, or might give better advice, or might have more up-to-date information, or might just be cheaper. While the production cost of a good General Referral server would be high, necessitating a profit motive, the production cost of small Referral servers is really no more than the cost of the network connection. The only other cost of publishing would be getting your site listed in other referral servers.

The problem is that building the Referral network is a huge task and one which is not easily mechanized. A number of possible methods exist for mechanically building a network of Referral servers. The best idea we were able to come up with was to start with a large number of specific Referral servers that cover narrow fields fairly well, and to use the abstracts associated with each server to automatically build layers of increasingly general servers above the layer of specific servers. This might be accomplished with a document clustering technique such as that of word-frequency vectors developed by Salton,[24][25] assuming that the abstracts are written in the right way and are long enough. It is even possible that the abstracts themselves might be automatically generated from analysis of the server, or at least automatically generated and then checked over and revised by humans. We do not know how well these techniques will work, and without a large number of specific servers it is difficult to test them. This idea is also covered (but not in much more detail) in Chapter 6.

## 8.4 Conclusion

There is plenty of work which could be done in this direction. This project is one exercise in what has been and will continue to be a long design phase. While the construction of a prototype is essentially complete, there are a great many simulations which need to be done to analyze the effectiveness of the protocol, and almost certainly the protocol needs major revisions. There are also a great many issues which have not been adequately covered, such as the implementation of security and authentication software and the implementation of a powerful object system, but many people are presently working on the implementation of such systems, and we will in many cases need to wait some time until we know what systems will be in place that allow this project to be completed.

# Appendix A

# An Example Filter Program

In Chapter 5 we saw a very simple example of a Filter program. That Filter did not take any issues of cost or time into account – it just executed all plans which came back. Writing a better Filter is not difficult. For example, the following Filter takes cost and relevance into account:

```
(begin
 (define init
   (lambda (time-left cash-left)
     (top-level-define max-unit-cost 3)
     (top-level-define current-relevance-min 90)))
 (define filter
   (lambda (process time-left cash-left)
     (if (= (+ 1 current-relevance-min)
            (priority process))
         (set! current-relevance-min
               (if (< current-relevance-min 0)
                   90
                   (- current-relevance-min 10))))
     (cond
       ((and (top-is-element? process)
             (> (top-avg-cost process) max-unit-cost))
        'punt)
       ((> (top-avg-rel process) current-relevance-min)
        'execute)
       (else
        (+ 1 current-relevance-min))))))
```

Here a number of built-in procedures have been used, most of which are needed to access the process structure. Table A.1 lists the function of several of these procedures. Also note that the Filter shown above will work best if the relevance values within each Plan are normalized on a scale of 1-100.

| Built-in Procedures | |
|---|---|
| `top-is-element?` | Returns true if topmost item is `plan-elt` |
| `top-avg-cost` | Returns average cost of topmost subplan. |
| `top-avg-rel` | Returns average relevance of topmost subplan |
| `top-level-define` | Special form which causes the given symbol to be bound in the top level environment |
| `set!` | Assignment procedure |
| =, >, + ... | The usual meanings |

Table A.1: Built-in procedures used in constructing Filters.

# Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[2] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The world wide web. *Communications of the ACM*, 37(8):76–82, August 1994.

[3] Tim Berners-Lee and D. Connelly. *Hypertext Markup Language – 2.0.* MIT/W3C, September 1995. See also URL http://www.w3.org/pub/WWW/MarkUp/html-spec/html-spec_toc.html.

[4] Tim Berners-Lee, R. Fielding, and Henrik Frystyk. Hypertext transfer protocol – http/1.0, October 1995. See http://www.w3.org/pub/WWW/Protocols/HTTP1.0/draft-ietf-http-spec.html.

[5] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The harvest information discovery and access system. In *Proceedings of the Second International World Wide Web Conference*, pages 763–771, Chicago, Illinois, October 1994.

[6] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, Michael F. Schwartz, and Duane P. Wessels. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, Colorado, August 1994. Revised March 1995.

[7] Britannica Advanced Publishing, Inc., a subsidiary of Encyclopedia Britannica, Inc. *About Britannica Online*, 1995. See also URL http://www.eb.com/help/aboutbol.html.

[8] Andrzej Duda and Mark A. Sheldon. Content routing in a network of wais servers. In *Proceedings of the 14th International Conference on Distributed Computing Systems*. IEEE, 1994.

[9] Paul Francis, Takashi Kambayashi, Shin ya Sato, and Susumu Shimizu. Ingrid: A self-configuring information navigation infrastructure, December 1995. To be presented at the Fourth International Conference on the World Wide Web.

[10] Lewis D. Girod. Public access vs. the information age: Bringing the mesh into public libraries. Paper written in completion of an independent research project, November 1995.

[11] Peter A. Gloor. Exploring large information spaces. Draft Manuscript, 1994.

[12] James Gosling and Henry McGilton. *The Java Language Environment: A White Paper*. Sun Microsystems, Mountain View, California, 1995. See http://java.sun.com/whitePaper/javawhitepaper_1.html.

[13] Frank Halasz and Mayer Schwartz. The dexter hypertext reference model. *Communications of the ACM*, 37(2):30–39, February 1994.

[14] Darren Hardy. The harvest summary object interchange format (soif). Part of the on-line Harvest Manual. See http://harvest.cs.colorado.edu/harvest/user-manual/node141.html#appendixsoif.

[15] Darren R. Hardy and Michael F. Schwartz. Customized information extraction as a basis for Resource Discovery. Technical Report CU-CS-707-94, Department of Computer Science, University of Colorado, Boulder, Colorado, March 1994. Revised February 1995.

[16] Brewster Kahle. *Wide Area Information Server Concepts*. Thinking Machines Corporation, Cambridge, Massachusetts, November 1989. Version 4 Draft.

[17] Brewster Kahle. *Document Identifiers, or International Standard Book Numbers for the Information Age.* Thinking Machines Corporation, Cambridge, Massachusetts, September 1991. Version 2.2.

[18] Brian A. LaMacchia. Personal communication during 1994 and 1995.

[19] Brian A. LaMacchia. Internet fish. Draft of a PhD Thesis Proposal, November 1994. See http://www.swiss.ai.mit.edu/~bal/prop/prop.html.

[20] Catherine C. Marshall, Frank G. Halasz, Russell A. Rogers, and William C. Janssen Jr. Aquanet: a hypertext tool to hold your knowledge in place. In *Hypertext '91 Proceedings*, pages 261–275, December 1991.

[21] Michael L. Mauldin. Information retrieval by text skimming. Technical Report CMU-CS-89-193, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 1989. pp. 124.

[22] Michael L. Mauldin. Lycos project description, June 1994. See http://www.lycos.com/reference/archive-post-01.html.

[23] Michael L. Mauldin. Measuring the web with *lycos.* Presented at the poster session of the Third International Conference on the World Wide Web, September 1995. See also http://www.lycos.com/reference/websize.html.

[24] Gerard Salton and Chris Buckey. Term weighting approaches in automatic text retrieval. Technical Report CORNELL TR-87-881, Computer Science Department, Cornell University, Ithaca, New York, November 1987.

[25] Gerard Salton and Chris Buckey. Approaches to global text analysis. Technical Report CORNELL TR-90-1113, Computer Science Department, Cornell University, Ithaca, New York, April 1990.

[26] Mark A. Sheldon, Andrzej Duda, Ron Weiss, and David K. Gifford. Discover: a resource dicovery system based on content routing. *Computer Networks and ISDN Systems*, 27:953–972, 1995.

[27] Mark A. Sheldon, Andrzej Duda, Ron Weiss, Jr. James W. O'Toole, and David K. Gifford. Content routing for distributed information servers. In *Proceedings of the Fourth International Conference on Extending Database Technology*, March 1994.

[28] Olin Shivers. The scheme underground web system, July 1995. See http://clark.lcs.mit.edu/s̄hivers/su-httpd.html.

[29] Karen R. Sollins and Larry Masinter. Functional requirements for universal resource names. Technical Report RFC 1737, Network Working Group, December 1994. See also ftp://ds.internic.net/rfc/rfc1737.txt.

[30] Karen R. Sollins and Jeffrey R. Van Dyke. Linking in a global information architecture, December 1995. To be presented at the Fourth International Conference on the World Wide Web.

[31] W3C Payments Group. W3c payments resources, 1995 October. See http://www.w3.org/pub/WWW/Payments/.

[32] W3C Security Group. W3c security resources, 1995 October. See http://www.w3.org/pub/WWW/Security/Overview.html.

[33] Yahoo! Corporation. *Yahoo! Frequently Asked Questions*, 1994. See also URL: http://www.yahoo.com/docs/info/faq.html.