Information Agents for Automated Browsing

Chanda Dharap
Philips Research Labs.
chanda@prpa.research.philips.com

Martin Freeman
Philips Research Labs.
martin@prpa.research.philips.com

Abstract

The proliferation of information providers has led to an increased number of users browsing the World Wide Web. Time-consuming interaction and increased network loads are some of the adverse effects of browsing activities. This paper presents a design based on agent technology and the use of structured information, that significantly lowers network load and limits user-interaction by automating the task of browsing.

1 Introduction

The World Wide Web (WWW) [4] and its clients, Mosaic and Netscape, in particular, provide a distributed and heterogeneous environment tied together via hypertext links. However, existing client-server hypertext approaches to content discovery exhibit several properties that limit the effectiveness of browsing.

First, the constantly changing nature of information makes it difficult to keep track of out-of-date and dynamic data. The user has to constantly browse and search for new additions and as yet undiscovered information. This constant act of browsing is a primary cause of communication overhead. In addition, wide-area networks add latency to information retrieval. Second, browsing is a user-guided activity in that the user filters and sorts through a considerable amount of uninteresting information in order to identify desired content. However, in existing models of information discovery, the operations of filtering and scanning the content are executed at the client's workstation. This is clearly inefficient. Third, existing information spaces are unstructured or primitively structured in that even when information is organized in specific patterns, it is not easy to extract and use these patterns effectively. This is a serious disadvantage as search in unstructured information spaces is imprecise — the return set from a search may be unreasonably large. Finally, wide-area information is heterogeneous — documents are stored in various formats, accessed via multiple protocols, and transmitted across multi-platform architectures. Thus it is a truly difficult task to locate, share and organize wide-area information.

The work presented here highlights using both agents and structured information to improve the quality of browsing across wide-area networks. We present an object-oriented implementation of agent-based infrastructure that is in close association with structured representation of information. An important feature of our infrastructure is that it efficiently combines the use of Knowledge Query and Manipulation Language (KQML)[14] and Java[3] to design a scalable model for agent communication. KQML is a protocol for information exchange among agents. Java is an object-oriented, interpretive language specifically developed for heterogeneous, platform-independent, distributed network computing. Our design enables us to pass object-oriented Java code in declarative KQML messages.

It is our belief that building a model based on these ideas, should significantly lower network load and limit user-interaction by automating the task of browsing and composing information. It is our hypothesis that agent technology significantly reduces the complexity of browsing wide-area information by reducing communication and filtering overhead as well as by distributing the workload. In a simplistic view, an agent accepts specifications for the desired information, interacts with relevant information providers, and composes information from various information sources to fit the user's specification. Our goal is to make the infrastructure as general as possible in order to accommodate a wide range of information content and agent-based transactions.

Specifically, our work addresses the issues of automating the task of browsing by executing content filtering and evaluation at remote sites. In addition, we believe that well-structured information content eases the task of automated browsing by agents. We demonstrate this by using the Nebula File System[8] for storage and retrieval of content via agents. The Nebula File System is a prototype wide-area information system, that offers a model for organizing and storing structured information.

Section 2 describes the problems and issues regarding

CIKM 96, Rockville MD USA
© 1996 ACM 0-89791-873-8/96/11 ..$3.50

browsing. In section 3 we present the idea of an agent and discuss some of the design issues involved. We also present the agent primitives – in particular, content primitives, communication primitives and movement primitives for mobile agents. Section 4 demonstrates a small example. Section 5 discusses background and related work.

2 Browsing Issues

Users locate and cluster wide-area information via searching and browsing. Examples of searching systems are commonly available search engines, databases and library lookup systems, where the user provides a query with sufficient information and gets back a set of documents that match the query. However, the precision of results depends entirely on the precision of the query. If the query is not very precise, the user is left with the task of scanning through a large amount of result data to identify documents of interest.

To avoid scanning large quantities of documents, a user browses through various collections before identifying the collection to search. Browsing is different from searching in that it is the act of clustering together information of interest. Typical browsing systems are Gopher and the WWW. The approach of clustering together information has been studied extensively in Information theory. In particular, the Scatter/Gather [10] approach uses a browse paradigm for demonstrating that document clustering can be an effective tool for information access. However existing systems do not include any primitives for organizing information and automating searches. Our goal is to automate browse and search, yet obtain reasonable *recall* [1] values and high *precision* values for searching and browsing. A high recall implies a large quantity of useful material returned and a high precision implies less irrelevant material.

Wide-area browsing also results in time-consuming user-interaction where the user loops through the cycle of evaluating results and visiting and revisiting sites. Consequently, network load and inefficient use of network/server resources become key issues in browsing wide-area information.

To reduce the network load and time-consuming user-interaction, the key idea in our design is to use agents to perform the browsing activity. Agents in our system are capable of carrying state information and partial results from provider to provider in an effort to satisfy the user's request. The key advantage here is that user-interaction is reduced for most requests. For example, when searching the home page of an information provider for content of interest, the user might indicate several different search criteria by clicking on HTML form buttons. To customize the results and organize them as appropriate, the user might sift through a returned list of items and possibly through similar lists from other service providers. We reduce this interaction by performing both the database access and the content filtering at the provider site through agents.

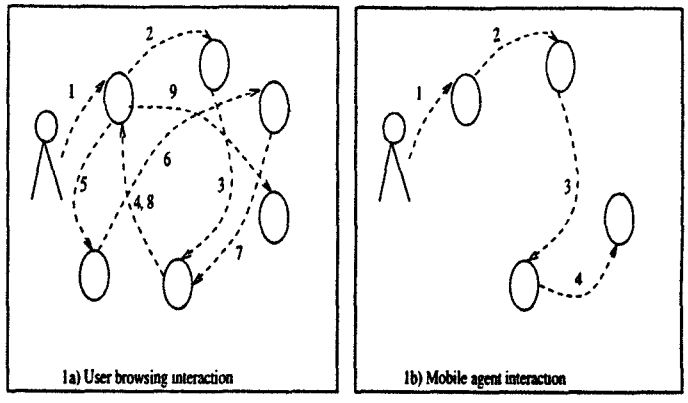

Figure 1: Browsing

Figure 1 shows typical browsing interactions — by a user as well as by an agent. The user's browsing pattern may contain duplicate traversals; for example, in browsing the network, the user may inadvertently click on a hypertext link which leads to a previously visited site. In contrast to this, the mobile agent, that has the capability to move around from site to site, lessens the user interaction and network load by avoiding duplication of traversals.

Toward efficient use of client resources, the key idea is pushing the computation to the server. Even though information servers may have the machine cycles to handle much of the computation associated with information services at their sites, a considerable portion of client-server applications are currently built to run primarily on the user's machine. Thus there is inefficient use of resources. In the domain of information systems, the information provider exists solely to provide the service, whereas for the typical user, information searching and browsing are two of the many activities performed during the workday. Network resources can be used more effectively if the provider can handle most of the application processing, by providing resources to mobile agents.

To summarize, these issues are resolved by re-distributing information access and processing to the server sites rather than the client site.

3 Agent Infrastructure

3.1 Overview

The work presented in this paper is based on two important ideas — the use of an agent network and the use of structured information to improve the efficiency of browsing. We implemented an example agent network and simulated an information provider that supports structured information. The implementation of the agent network is in Java and uses KQML, whereas the presented example and implementation of the information provider is based on the Nebula system.

To simulate an information provider, we chose to index

[1] *Recall* and *precision* are measures used in information retrieval theory. *Recall* is defined as the proportion of relevant material retrieved and *precision* is defined as the proportion of retrieved material that is relevant.

data into Nebula. As mentioned earlier, Nebula is a prototype file system that indexes compact summaries of files. It provides a flexible search interface to accessing information, allowing the dynamic reorganization of the underlying indexed information. Nebula uses structured information collectors, which automate the task of appropriately summarizing documents. The data we indexed is originally from the USENET newsgroup rec.arts.movies. We used Nebula collectors[12] to summarize and index approximately 600 movies of various categories.

Part of Nebula is a lightweight semi-relational database. An added advantage is the ability to maintain context information to refine further queries. It would be quite possible to replace the Nebula component by any suitable commercially available database.

3.2 Agents

In our system, an agent is a software program that communicates with any other agent using a universal agent communication language (ACL). In the literature, there are two general classes of agent-based interactions: prescriptive and descriptive.

In the prescriptive case, programs in the form of scripts are sent between a sender and a receiver, the receiver executing the scripts on behalf of the sender. In the descriptive case, the sender issues declarative statements and the receiver uses an inferencing engine to derive the required results. Additionally, agents can be characterized in terms of their use of static or mobile capabilities[15]. Static agents are associated with a particular client or server. Mobile agents roam the network visiting various servers in carrying out their tasks.

To unify these aspects of agent technology, we view an agent infrastructure as consisting of a confederation of static agents and mobile agents, where static agents communicate with each other through messages (descriptive) and mobile agents (prescriptive). In our design, static agents are gatekeepers for applications and may themselves perform additional functions (e.g.; send mail, filter netnews). In this view, mobile agents combine the functionality of both descriptive and prescriptive interactions. Mobile agents may contain data, programs, procedure calls and query scripts. In addition, they may contain administrative information used by static agents. A detailed description and example is presented later.

3.3 Design Issues

This section discusses the issues of language considerations, agent communication, mobile code and security.

3.3.1 Language Considerations

The agent infrastructure consists of static processes and mobile scripts which collaborate across wide-area networks. Safe execution, flexibility, ease of programming, and scalability and ubiquitousness of the code are of primary importance when considering a language for implementation.

Of the above, safe execution is of singular importance because of the interoperating nature of agent code. Safe execution entails execution in a restricted namespace as well as strong type-checking in order to eliminate common programming errors as have been seen in traditional C programming. We make the tradeoff between speed and flexibility by choosing an interpretive language compiled to an underlying virtual machine. Interpretive languages are easier to program as compared to compiled languages. Interpretive languages also provide for fast prototyping. This is particularly important in order to reprogram an agent, if necessary.

In our infrastructure, the language of implementation is Java. The Java environment provides the ability to build secure, compact, platform-independent and scalable applications. Java is object-oriented and dynamic, allowing for inheritance and code reuse as well as run-time extensions to existing code. We make full use of Java's abstraction mechanism and techniques of information-hiding in order to provide a uniform and protected interface to registered information providers. This is particularly important in our infrastructure design as it is intended to scale to diverse information services with varied interfaces. In the implementation of the infrastructure we provide abstractions for mapping agent requests into the provider's query language of choice. Abstraction, as provided by Java, allows us the sharing of programming interfaces without being completely aware of the implementation.

These considerations make Java a great vehicle for delivering the agent paradigm to the Internet, where a Java encoded agent could move across heterogeneous hardware platforms.

3.3.2 Agent communication language

This section discusses the decisions made in choosing a mechanism for agent communication. Since the agent infrastructure is designed to be general and extensible, the communication between agents must not solely be dependent on procedural implementation (hard-coded); rather it must be based on the ability to exchange information and vocabulary. That is, Tcl-based agent code should be able to communicate with Java-based agent code without understanding the actual implementation language. This is possible via KQML. KQML defines message formats as well as communication primitives to support run-time knowledge sharing among agents. The advantage of using KQML is that we may use any of the existing standards for communication at the transport level; for example, TCP/IP, email, HTTP and CORBA[21]. New and upcoming standards can be folded into KQML as well.

We combined the use of KQML and Java to design a scalable model for agent communication. In order to do this, we implemented a variant of KQML with a core set of KQML primitives for inter-agent, inter-component communication. The infrastructure is extensible in that it is designed to accommodate a gamut of agent technologies — for

example agent communication with other agent applications is possible. Messages that are sent between elements of the agent infrastructure are a variants of KQML messages. For agent systems that support only static agents, standard KQML performatives[2] are used, but additional parameters are supplied associated with authentication, resource use, etc. For mobile agents, an additional **execute** performative is used with an agent program being transmitted as one of the associated parameters.

For KQML messages that support descriptive agents, the message contains a performative and a set of name/value pairs. One of these pairs describes the actions that the agent is to take, but not how the actions are to be accomplished. The receiving agent takes the description and processes it. As a simple example, a KQML message might be issued with the performative **ask** and a content field containing the description of data to be fetched. After due processing at the information provider, a **reply** performative is sent back with the results. KQML has more than two dozen reserved performative names, which fall into specific categories. We implemented some of the reserved performatives as well as extended KQML by adding our own. For reserved performatives, we followed the semantics defined by the DARPA Knowledge Sharing Initiative as closely as possible. (See appendix for some of the currently implemented KQML primitives.)

Our approach is novel in that it shows the way to combine descriptive and prescriptive approaches with the help of static and mobile agents encoded in Java and communicating via an evolving standard for agent communication.

3.3.3 Mobile Code and Security

Mobile agents are used to reduce computing overhead and to increase flexibility, presenting a clean solution with respect to a number of issues.

First we address the issue of safety. As a precursor to allowing an agent to execute on a remote site, we implement a KQML performative to handle the authentication phase. This performative **negotiate** is not in the recommended set of KQML specified performatives. We add this optional performative to enable authentication and negotiation for resources to be encapsulated within the agent communication. The authentication phase carries out the verification of the agent asking to execute on the site. Resource requests are checked by computing the costs of the request at that site and comparing those costs with the agent's available credits. If a request is turned down, the agent can come back with a modified credit or resource requirement or an updated credit amount. Our infrastructure is extensible in that any new authentication protocol can be added to the communication by simply implementing to the exported interface.

Next, we address the issue of actually transferring the agent's code across the network, and executing it in a safe environment. The threat of a virus or malicious agent is a big concern. Since the language of implementation is Java, we use many of its safety features. Java implements a loader for local and remote classes that works as follows. It checks for local versions of any referenced classes. If a local version exists, it is used instead of using the newly loaded remote class definitions. The classes local to the machine reside in one namespace and all other classes that are loaded from remote sites are given separate name spaces. Thus, by using different namespaces for local and remote code, local class definitions cannot be overloaded and redefined by a malicious agent/user.

Java also uses a bytecode[3] verifier, which is essentially a theorem prover. This checks the downloaded code for transgressions. Some of the items that are checked are: forged pointers, access restriction violations, object mismatches in the code, operand stack overflow and underflow, parameter checks and illegal data conversions.

3.4 Content Primitives

An agent in our infrastructure is a code object capable of holding content specifications, resource requirements and partial results. The user specifies the content to be discovered using an interface tool. This specification is translated into appropriate KQML primitives, with structured descriptions of the desired domain of results. Structured descriptions are unordered lists of name-value pairs.

Since users have varied preferences and selection criteria, we propose the use of preference functions that enable the agent to make a choice on the user's behalf. More precise results are possible when the user specifies *preference* functions. In our prototype implementation we use preference functions based on result size. More complex preference functions may be added by the user or administrator. Preference functions [6] are a great help in pruning data and throwing away undesired information. As part of future work we propose to develop a model to define and specify preferences.

4 System Structure

This section outlines the basic system structure, the agent communication flow and the semantics of primitives. The system is comprised of four major classes of objects. The guard class, the static agent class, the mobile agent class and the interface class.

The guard class provides the functionality of a gateway. It supports KQML primitives to communicate with all the other components of the system. The guard maintains two major data structures – a global mapping table and a local mapping table. The global mapping table maintains information about other guards in the system, and the local mapping table maintains information about static agents registered at its site. These mapping tables act as nameservers in that they maintain appropriate information about registered static agent objects,

[2]Description of an action to be taken.

[3]The Java compiler creates a platform-independent set of bytecodes for its own virtual machine, and the interpreter converts them to machine language at run-time.

i.e. key services provided, popular search words, and host and port number information among others. The guard class runs in its own thread and listens for connections. It spawns an agent handler for every new connection from an agent. Once the connection is made, the handler thread listens for a KQML message to arrive. The handler decides the guard to agent communication path based on the KQML performative that is read.

The Static Agent (SA) provides an interface to the information provider's content. It runs in its own thread and exports an interface of abstract methods to map KQML queries into the information providers's database language. Thus, a SA needs to be tailored to the provider's interface. The SA, after opening a connection to a well-known guard, registers itself and closes the connection. The SA spawns a connection handler to handle incoming mobile agent connections. Mobile agents spawned by the Guard use KQML to communicate with the SA.

The Mobile Agent (MA) operates in several stages. It first communicates a *find* request to the Guard. The Guard performs a table lookup in the local table and finds an appropriate Static Agent for the MA to communicate with. If no such Static Agent is available at its site, the request is forwarded to another guard after performing a global table lookup. After an appropriate SA is located, the Guard uses a classloader to load the Mobile Agent and its supporting classes over the connection. The MA wraps itself in a KQML message and exits after the classloader finishes the transfer. At this stage the MA is in stasis. The Guard instantiates the MA and passes to it the SA's connection handle. From that point on the MA communicates with the SA for information.

Some communication primitives and KQML messages are discussed below.

4.1 Communication Primitives

KQML is comprised of three parts: a vocabulary, an inner content language, and an outer transport language. The vocabulary part is related to the application area being addressed, while the content language may be anything from the propositional calculus to being a scriptlike language. A KQML expression consists of a performative and a set of related name-value pairs, some of which are standard. A performative is an action that closely models the actions, assumptions and conventions associated with human interaction. Performatives include **tell**, **ask**, **reply**, etc, whereas the name-value pairs specify associated parameters.

KQML is structured as a three layer architecture: the content layer, the message layer and the communication layer. The communication layer encodes a set of features that describe lower-level communication parameters such as the identities of the sender and receiver of a KQML message, while the message layer provides the performatives that indicate speech acts. KQML messages are not aware of the content that they carry, and thus there are no restrictions on the content language.

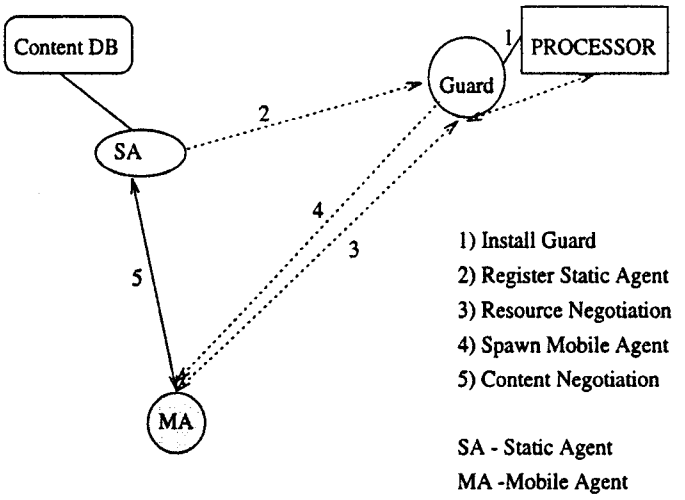

Figure 2: Example Message Flow

An example KQML message:

```
( ask-one:
  :language  SQL
  :content   (SELECT MOVIE WHERE TITLE
                 = "Maverick")
  :receiver  Western-Movie-Server
  )
```

In our implementation one of the content languages of KQML messages are Java bytecodes representing executable Java classes. Java supports the transmission and execution of bytecodes between machines as well as their remote execution. Associated with each Java system is an interpreter that executes the received Java bytecodes. Classes may be transmitted from one machine to another and executed by the receiving machine's interpreter. Each machine has a Java class library, so that agent code need not carry these common classes, thereby saving valuable communication bandwidth and time.

4.2 Movement Primitives (Message Flow)

There are four classes of objects in our infrastructure. This section specifies these classes and shows by example how the message flow between the component objects is achieved, see figure 2.

- **Initialization:** A guard is associated with each server or processor that supports the infrastructure. See 1) in figure 2. A guard process starts up on installation. Its function is to receive a message and spawn a mobile agent process when satisfied that the agent has the appropriate authority and resource requirements. The guard is therefore responsible for preliminary negotiations with the mobile agent. The guard validates the agent's authority to execute, checks the resource graph of the

processor to ensure that the agent's demands can be met and finally negotiates with the mobile agent for resource payment, if any. The prototype does not implement a payment module. However, the infrastructure is built in a way that existing payment schemes can be plugged in with minimal effort. The guard also plays a role in the registration step (below).

- **Registration:** A static agent is associated with each information service. See 2) in figure 2. When first activated, a static agent must register itself with the local guard process at its site. The static agent provides a mapping between the mobile agent's KQML communication and valid queries in the service's query language. In the event that a certain server provides more than one information service, more than one static agent may reside at a server. As part of the registration process the static agent exchanges information with the guard process by registering key attributes of the information service. This provides a mechanism to locate information whose exact placement is unknown to the mobile agent or the user. By further improving the quality of such registered attributes, resource discovery becomes easier.

- **Mobile agent:** When a mobile agent seeks to travel, a KQML message is issued to another guard. See 3) and 4) of figure 2. This message is a wrapped up agent code. A guard is a process which sends outgoing KQML messages, receives incoming KQML messages and processes them. At the receiving end, if the content language is prescriptive like Java, the guard will spawn a thread that executes the Java code; if the content language is KQML, the guard process becomes a local proxy for the sender.

- **Movement:** Mobile agents may negotiate for a service with the service provider's static agent, schedule the delivery of content, and pay for the service. See 5) of figure 2. When this interaction is completed, whether successfully or not, the mobile agent, in consort with its current guard, packages up its state and code and is transported to another service provider. This is so when the agent's global task is incomplete. The agent is sent back to its originator when it completes its mission. Being sent home can also happen if the mobile agent fails on a remote machine or expends its specified resource usage limit.

  In our current implementation, the next place to go is inferred by checking a list of available guards. However, by using Java class libraries, it is possible for the agent code to dynamically load and execute an inference module to make this decision. This inference module could range from primitive to complex prolog or rule-based engines.

Using the infrastructure outlined above it is possible to build a model for automated browsing using agents. For example, Figure 3 shows an example of an architecture where nameservers, brokers and information providers combine to provide a better and more usable access to wide-area information using agents. The nameservers and brokers in this model could be used by mobile agents and their users to narrow the search space. In a business model, the brokers could charge for information requests or even lease out accesses for automated *crawlers* to extract information.

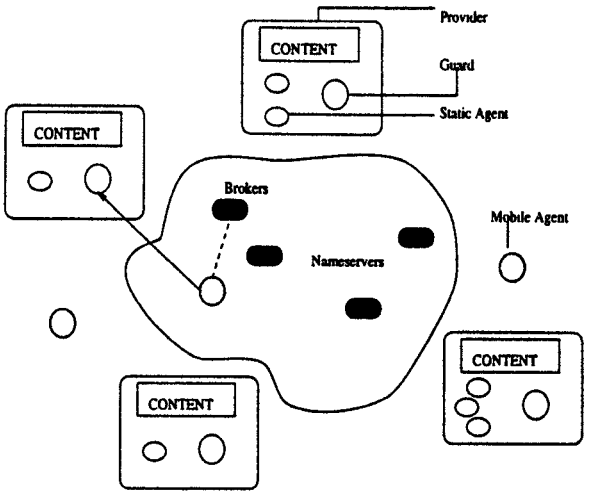

Figure 3: Example browsing model

5 Example

We constructed a prototype of the infrastructure presented in this paper to demonstrate the feasibility of its various components. The prototype implements an information provider with a guard and a static agent and as well as a mobile agent that queries for information.

The key idea behind indexing structured information in Nebula is that interesting attributes of the movie data are made available in the form of compact summaries for structured querying. Some of the attributes we index are **title, production company, summary, cast, writer** etc. The entire document is stored as part of the **text** attribute for a detailed display. The summary may instead contain pointers to the actual location of the data, either on disk or on the video provider who may allow downloading of the movie to the user. Nebula also provides a mechanism to logically organize an information space into contexts based on the attributes they index. We use this mechanism to organize the data by **genre**. For our example, we used two logical partitionings of the movies; **western** movies and **science fiction** movies. In a real world application of this model, the two contexts could represent individual information providers.

Following is an example Nebula object:

```
( (uid          !ab09)
  (title        "9 to 5")
  (production   "20th century Fox")
  (category     "comedy")
```

```
(summary      "Frank Hart is a pig. His 3
              assistants manage to trap
              him in his own house
              and assume control")
(cast         "Jane Fonda")
(cast         "Lily Tomlin")
(cast         "Dolly Parton")
(writer       "Collin Higgins and
              Patricia Resnick")
(text         "Located at .. ")
              )
```

The **uid** attribute uniquely identifies the object and points to a fixed record that contains storage and protection information.

Further, we constructed an agent that looks for movie information and schedules it for viewing. The rest of this section highlights how an agent can step through the information space, by providing refinement queries at each step in order to locate correct information.

Browsing is a two-step activity. First the agent must locate appropriate information providers. The user specifications and preferences are mapped into KQML messages. Some of these preferences may be implicit. For example, let us consider the case where the user is interested in viewing a reasonably recent science fiction movie. The user is also in the mood for light relaxation and places a further constraint on the information sought viz: "A reasonably recent science fiction movie with a comedy theme." However the scheduling for this movie can only be made at a particular two hour time-slot. Obviously the best place to locate such a movie is by searching a movie database. The agent uses a default set of guards to locate candidate static agents associated with information providers. There are many ways in which the agent may get this list. The set of such guards and static agents may be obtained from information brokers for a fee, or by exchanging information with peer agents. This information might also be accessed from newsgroups, mailing lists and electronic bulletin boards.

The second step is to retrieve from the set of candidate information providers, the movies that match the agent's request. This retrieval may be carried out in several iterations of searching and browsing. The agent may compare results from more than one provider in order to obtain a near perfect match. For example, the running time of the movie under consideration has to be within the time-slot available to the user. The agent may search more than one provider for a movie that matches these implicit specifications.

Nebula provides dynamic reorganization of the underlying object space to support browsing. The mobile agent via its interactions with the static agent carries out such reorganizations and clustering until the desired information is obtained. See figure 4. The mobile agent formulates a KQML performative which encapsulates the desired query.

```
( ask-one:
```

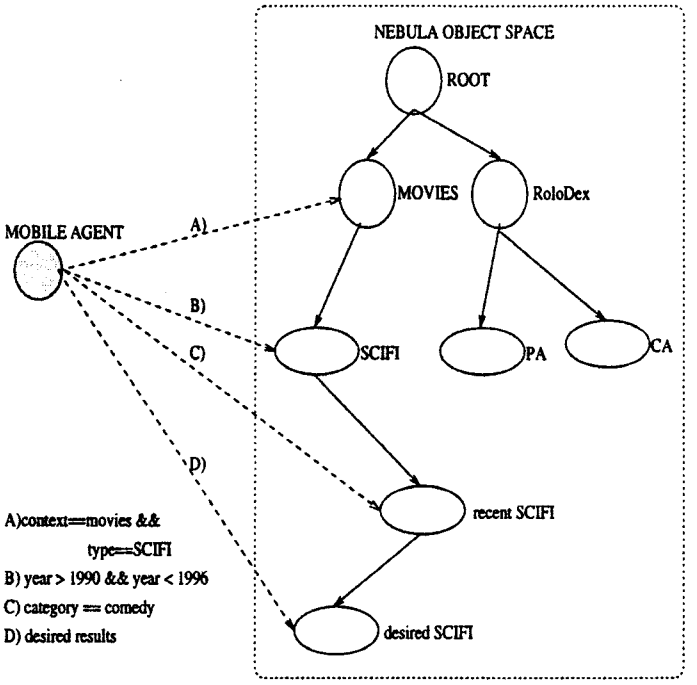

Figure 4: Example

```
:sender 12
:receiver SA1
:language NEBULA-QUERY
: content "context = movies & type == scifi" )
```

In Nebula, information is partitioned into contexts that index the attributes of information within that context, see Figure 2. For example, the "movie" context indexes attributes like title, cast, director, producer, running time, summary and category, among others. The static agent unwraps the KQML performative and translates the query into Nebula expected format. The query may contain additional specifications about actors, directors, writers or key phrases in the summary. Nebula resolution is different from traditional resolution in that the results of a resolution are clustered together in a temporary view. An identifier for this view is passed back along with the objects in the result. This identifier can be used for subsequent refinement by further queries. Thus Nebula provides the functionality to organize information for precise query resolution.

Information within the "movie" context that is about science fiction is clustered together in a temporary view. The identifier of the view and the size of the set is passed back by the static agent to the mobile agent. If information about the year of the movie is absent from the result, the agent may send a query for the date to another information provider, after refining the query (by adding the name of the movie that was just resolved) as follows:

```
:content "view-id = !8 &
( year > 1990 & year < 1996 )"
```

This descriptive name/query is scoped on the view of science fiction movies. Scoping within the earlier view obviates the need to evaluate the query over the entire set of movie objects. This improves the precision of the result set and improves the performance of query resolution. The result of the above is another view along with the result set.

If the provider categorizes only by the genre **science fiction**, the agent may query a comedy movie database for the movie titles obtained in the earlier result. On the other hand, the agent may qualify the query by adding the following and scoping it on the previous results:

summary == "comedy"

Finally, when there are no more implicit preferences to be satisfied, it sends a message to the static agent indicating the end of the protocol. This is a signal for the static agent to send a *destroy-temporary-views* message to the Nebula server.

Complex queries can be constructed from the refinements by using *and* and *or* operators. This complex query can be stored with the agent and used to shortcircuit a path to that precise information cluster in the already browsed information space. Example:

(context == "movies" & type == "scifi" &
summary == "comedy" &
(year $\triangleright$ 1990 & year < 1996)

This complex query can be used to get the same results with possibly less browsing. The information that the agent used to construct the query can be shared with peer agents, thus shortcircuiting a path to most commonly desired information. Sharing provides the basis for building topic-specific agents that act as specialized archives of information.

On successful termination the agent will be packaged up as a KQML message and transported to its next destination. This destination is provided to the guard at termination via the inference mechanism of choice. For the next version, a destination transfer protocol will ensure recoverability by leaving shadow images until the agent has been safely transported to the next destination.

6 Related Work

Previous studies on improving access to wide-area information systems have focused on increasing the efficiency of searching by using better indexing techniques and partitioning of the data [11] [16].

Information systems like Harvest[5], TSIMMIS[9] and the Stanford Data Warehouse[18] provide an architecture that can be used to construct information brokers or service brokers. These brokers lessen the bottleneck at the server site by significantly reducing server load and space requirements at the information provider's domain. However, with the vast expansion of information, the brokers themselves can become the bottleneck.

More recent efforts have concentrated on building indexes of WWW information. Web crawlers, robots and spiders are programs that traverse the World Wide Web and automatically download and index the content. Some automated indexers/classifiers are Alta Vista[1], Yahoo[2], Lycos[20] and MOMspider[13]. These systems try to automate the indexing process, and in many cases even categorize the collected information. However, most of these collect information from around the Internet and index it for further retrieval. While this may increase the recall of the service, it effectively decreases the precision.

Automated indexers do not obviate the need for browsing, nor do they make browsing any easier. Problems with network and server load are not eliminated; they only occur at different times. That is, users do not account for a lot of the server load at browsing time, the server load is caused by web crawlers at downloading and indexing time. Network communication is simply shifted from the user's interaction with an information server to interaction with an indexed server that points to the information. Large quantities of unpruned information still come back to the user, who then goes through the process of eliminating and clustering this information. The communication load is still associated with the user.

However, agents help in moving the computation load from client to server by moving the program from the client to the server. Consequently, server bandwidth is reduced, since already pruned information comes back to the user as opposed to the large number of bytes that have to be pruned. Browsing becomes a much less daunting task as the amount of information to be browsed is already reduced. This situation is further improved by providing more information about the content being indexed.

Prominent examples of agent-based architectures are Telescript [22], Tacoma [19] and Agent Tcl [17], among others.

Telescript is an object-oriented scripting language for agent programming. A Telescript agent uses a **go** command to migrate from one site to another. Each Telescript network site runs a server that executes incoming agents. The server logs the state of resident agents, thereby permitting the agent state to be restored after a site failure.

Tacoma is a mobile agent system that uses Tcl for its agent language. Each agent carries a briefcase of folders containing Tcl programs and data. An agent may execute a *meet* command to meet another agent and pass a briefcase to that agent. Tacoma does not support the interruption of executing Tcl scripts, and interrupted scripts that are passed on must start from the beginning. In our approach, transferring execution state involves modifying the Java interpreter, and we have chosen not to do this at this point. Scripts are migrated by storing state information in the data that migrates along with the agent. In our approach folders correspond to name-value pairs, and the briefcase corresponds to a KQML message.

Agent Tcl is an extension to the Tcl scripting language to allow scripts to be suspended on one machine and to be started up on another machine. Additional Tcl commands are provided to move the agent through the network and to send and receive messages. As with our approach, Agent Tcl requires a server to be operational at each site to which an agent can be sent. The server watches a socket connection and accepts incoming agents.

Our work is distinct from these approaches in that we use KQML to provide more semantic information through the use of *performatives*. In addition, the use of Java offers universality of underlying agent code and any modules, inference engines, interface libraries can be dynamically linked in.

7 Conclusions

This paper presents the use of information agents for automated browsing. Browsing is distinguished from searching in that content may be evaluated at more than one indexed server.

We present a prototype implementation of information agents where agents find requested information by browsing various information servers. The key features of our approach include:

1. The combination of KQML and Java for agent communication,
2. The use of structured meta information to decrease the complexity of browsing.
3. An infrastructure that uses static as well as mobile agents. The use of static and mobile agents allows for both descriptive and prescriptive messages to be used to communicate among peer agents.

It is our expectation that this design will significantly reduce user interaction by automating the task of browsing and composing information. We believe that performance in terms of speed is not as important in this case as performance in terms of what is retrieved. Information theory uses precision and recall metrics to measure the quality of retrieval of an information system. However, there are two problems with this approach. First, it is laden with subjectivity. One needs user feedback to judge the relevance of the retrieved results. Second, and most important, these measures are used for search and retrieval engines. Browsing is relatively ambigious in its definition and does not have a set of benchmarks or benchmarking proceedures. An interesting extension to this work would be to evaluate and come up with possible benchmarks to measure navigation systems and browsing systems. One of our recent papers describes studies of precision measures for structured content [7].

We conclude that we can use agent technology to reduce the complexity of browsing wide-area information in two ways: by reducing communication and filtering overhead, and by moving the computation away from the client.

We believe that information agents can be categorized based on preference functions and associated inference mechanisms used to implement them. We plan on analyzing the class of information agents further in this manner.

References

[1] A fast search engine. At http://www.altavista.digital.com.

[2] Yahoo, a search index. At url: http://www.yahoo.com/.

[3] The Java Language: A White Paper. Sun Microsystems. At url: http://www.java.sun.com/, 1995.

[4] T. Berners-Lee, R. Calliau, and B. Pollermann. World-Wide Web: The Information Univers. *Electronic Networking: Research, Applications and Policy*, 2:52–58, Spring 1992.

[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*, Chicago, Illinois, October 1994.

[6] Mic Bowman, S Debray, and L.L. Peterson. Reasoning about naming systems. *ACM Transactions on Programming Languages and Systems*, 15(5):795–825, November 1993.

[7] Chanda Dharap, and Mic Bowman. Typed Structured Documents for Information Retrieval. To Appear in *Prooceedings of the Third International Workshop on Principles of Document Processing*, Palo Alto, California, September 1996.

[8] Mic Bowman, Chanda Dharap, Mrinal Baruah, Bill Camargo, and Sunil Potti. A File System for Information Management. In *Prooceedings of the International Conference on Intelligent Information Management Systems*, Washington D.C., March 1994.

[9] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information. In *Proceedings of the IPSJ Conference*, Tokyo, Japan, October 1994.

[10] R. Douglas et. al. Cutting. Scatter/Gather: A Cluster-based Approach to Browsing Large Document Collections. *15th Annual Int'l SIGIR*, pages 318–328, June 1992.

[11] Peter B. Danzig, Li Shih-Hao, and Katie Obraczka. Distributed Indexing of Autonomous Internet Services. *Computing Systems*, 5(4):433–459, Fall 1992.

[12] Chanda Dharap and Mic C. Bowman. Structure in file systems. Technical Report CSE-94-021, The Pennsylvania State University, February 1994.

[13] R. T. Fielding. Maintaining Distributed Hypertext Infostructures: Welcome to MOMspider's Web. In *In Proceedings of the First International Conference on the World-Wide Web (WWW'94)*, Geneva, May 1994.

[14] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM '94)*, November 1994.

[15] M. R. Genesereth and S. P. Ketchpedal. Software Agents. *Communications of the ACM*, 37(7), July 1994.

[16] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, O'Toole, and James W. Jr. Semantic File Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25, Oct. 1991.

[17] R. Gray. Agent Tcl: A Transportable Agent System. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management*, December 1995.

[18] J. Hammer, H. Garcia-Molina, W. Labio, and Y. Zhuge. The Stanford Data Warehouse Project. *IEEE Data Engineering Bulletin*, June 1995.

[19] D. Johansen, R. Renesse, and Schneider F. Operating System Support for Mobile Agents. In *Proceedings of the 5th. IEEE Workshop on Hot Topics in Operating Systems*, May 1995.

[20] M. L. Mauldin. Measuring the Web with Lycos (poster presentation). In *Proceedings of the Third International World-Wide Web Conference (WWW'95)*, April 1995.

[21] T.J. Mowbray and R. Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects*. Number ISBN 0471106119. Wiley/OMG, 1995.

[22] J.E. White. *Telescript Language Reference Manual*. General Magic, Inc., Sunnyvale, CA., October 1995.

A Performatives

1. **ask-all**

```
( ask-all:
  :content-language
  :content
  :receiver
  :sender
  :reply-with
  )
```

The **ask-all** is a basic query performative. When the ask-all performative is encountered, the results are sent back as they are computed, until all the processing is finished. Each communication carries the identifier that is supplied as values of the :reply-with parameter. This helps to regroup the replies together.

2. **reply**

```
( reply:
  :receiver
  :sender
  :reply-with
  :content
  )
```

The **reply** performative is a response performative, whose content parameter carries the value of the query resolution.

3. **tell**

```
( tell:
  :receiver
  :sender
  :ack
  )
```

The **tell** performative is used as a generic informational performative. Although it is generally used as purely truth setting operation on meta data, we use it to acknowledge a request.

4. **register**

```
( register:
  :content-language
  :content
  :receiver
  :sender
  :ISname
  :SAname
  :SAhost
  :SAport
  :ISkeys
  )
```

We use the **register** performative to register an information provider. This performative communicates various parameters, like information provider name, hostname, portnumber and some meta data which describes the content at the information providers database.

5. **execute**

```
( execute:
  :content-language
  :content
  :receiver
  :sender
  )
```

In our case the content language for this performative is always Java. The content field contains Java bytecodes, which is the compiled agent code. The agent is thus transported as content across the network infrastructure.

6. **find**

```
( find:
  :keyword
  :receiver
  :sender
  )
```

We use the **find** performative as a request to locate an appropriate information provider.