Scalable Semantic Brokering over Dynamic Heterogeneous Data Sources in InfoSleuth™

Marian Nodine*, Anne Hee Hiong Ngu†
Anthony Cassandra‡, William Bohrer§
Microelectronics and Computer Technology Consortium (MCC)

April 20, 2001

Abstract

InfoSleuth is an agent-based system for information discovery and retrieval in a dynamic, open environment. Brokering in InfoSleuth is a match-making process, recommending agents that provide services to agents requesting services. This paper discusses InfoSleuth's distributed multibroker design and implementation. InfoSleuth's brokering function combines reasoning over both the syntax and semantics of agents in the domain. This means the broker must reason over explicitly advertised information about agent capabilities to determine which agent can best provide the requested services. Robustness and scalability issues dictate that brokering must be distributable across collaborating agents. Our multibroker design is a peer-to-peer system that requires brokers to advertise to and receive advertisements from other brokers. Brokers collaborate during matchmaking to give a collective response to requests initiated by non-broker agents. This results in a robust, scalable brokering system.

Keywords: Multibrokering, Semantic Matching, Facilitation, Multiagents, Information Agents, Heterogeneous Systems

1 Introduction

Distributed architectures partition the execution of tasks over processes spread out over a computer network. A particular process passes off some subtask to another process either by sending it a message and waiting for the response, or via a remote procedure or method call. The brokering function in a distributed system matches a request for a specific service with a remote process that can perform that service, based on some subtask's need for that service. Both distributed object systems and agent systems require the use of the brokering function. A broker or matchmaker is a process that implements and executes the brokering function. In a brokered system, agents advertise themselves to a broker, then the broker responds to queries about agents.

In this paper, we examine brokering from the perspective of tailoring it to add semantic richness, robustness, scalability, and flexibility to either an agent community or a distributed object system. Syntactic brokering uses the structure or format of a task specification to match a requester with a service provider, matching requests to object interfaces or query/scripting languages to decide which service providers to

* *currently at Telcordia Austin Research Center, email:nodine@research.telcordia.com*
† *currently at Telcordia Austin Research Center, email:angu@research.telcordia.com*
‡ *Currently at Tower-J, email:arc@cassandra.org*
§ *Currently at Athens Group*

recommend. For example, "myRelationalQueryAgent" may advertise that it takes its input according to SQL 2.0 syntax. Any request for SQL 2.0 query processing could then be directed to this process.

In a distributed object architecture such as CORBA (Common Object Request Broker Architecture), a process can run a remote method by first accessing a broker (e.g., a CORBA ORB) to locate remote objects that conform to a specific type or method call interface. One basic assumption that these systems make is that the definition of the procedure or method interface uniquely defines its semantics. That is, ORBs do not check to see if the local implementation actually does what the caller intended. By this definition, CORBA currently provides syntactic brokering services.

Agent systems, like distributed object systems, offer the ability to partition the execution of a task over several processes distributed across a network. Within the agent community, agent communication languages such as KQML define a set of *performatives* that effectively represent an ontology that agents can use to advertise their capabilities. This approach reduces the ability to describe agent semantics in terms of the interface to an agent, and thus also implements syntactic brokering. Unfortunately, within an open agent-based system there may be situations where agents present the same interfaces, but implement different functions. Moreover, interactions between agents often take a more complex form than remote procedure calls, encompassing a series of method/message exchanges or *conversations*.

The InfoSleuth agent system [2, 6, 16] adds *semantic* brokering functions to complement the syntactic brokering process. Semantic brokering uses the intended operation and accessed information of the request to match it with the meaning of the offered services of the providing agent. Semantic brokering allows a broker to recommend services based on the semantics that define the sub-task. For example, "myRelationalQueryAgent" might advertise or register with the broker that it has the capability to do query processing of relational algebra queries, but it cannot do any statistical aggregation within those queries. Any time some other agent requires a query to be run over some known set of relational data, and the query fits those constraints, "myRelationalQueryAgent" could be tapped to do the job. One could also envision brokering over other aspects of agents; such as agent load or perceived accuracy of results.

A distributed agents or object system must be able to scale to support many agents/objects in order to be useful. We considered several aspects with respect to scalability in the InfoSleuth brokering system. First, the reasoning engine within each broker must be able to provide a fairly comprehensive reasoning service while remaining fast, with respect to the number of agents being reasoned over. This motivated us to move to an approach that allowed for adequate specification of agent semantics, without relying on the need to reason over logical specifications of those semantics. Our approach is based on simple constraint-based reasoning, and operates efficiently.

We have also designed the brokering service so that it could be distributed among multiple brokers. Brokering could be provided by a single process or agent that saves all agent advertisements in a single repository, and processes all requests for services. In our experience, this architecture works well for agent systems with a few dozens agents. However, the broker then represents a single point of failure and a limit

to scalability. A alternative approach would be replicated brokers, which allow multiple copies of the same broker to exist in the agent system. This eliminates the single point of failure, and allows the queries to be distributed across the replicas, but does not eliminate the issue of having many agents advertising, and having to manage those advertisements.

In InfoSleuth, we implemented a *distributed multibrokering* approach where many brokers collaborate to provide brokering services. In the distributed multibrokered system, many broker agents collectively maintain the information about agents in the system. Each agent in the system must be known to at least one broker, and a given agent may advertise its capabilities to one or more brokers, depending on the level of reliability it requires. When a broker receives a query for services, it not only searches for matches within its own repository, but also propagates the query to other relevant brokers, and collects the results together before returning them to the agent that sent the query. This approach is both robust and scalable.

The remainder of this paper is organized as follows. In Section 2, we briefly review InfoSleuth, our agent-based information discovery and retrieval system where multibrokering is a core function. In Section 3, we discuss the nature of the brokering process that matches a service requester with a service provider, and present the ontologies for describing agent capabilities. Section 4 describes the original, single-broker implementation of InfoSleuth and its reasoning engine. Section 5 describes important multibrokering principles and how they are implemented in InfoSleuth. In Section 6, we discuss the experiments that we conducted to confirm the robustness and the scalability of our multibrokering approach. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 InfoSleuth

The InfoSleuth system consists of a set of collaborating agents that work together for information discovery and retrieval in a dynamic environment, where available information and resources change over time. Ontologies facilitate agent communication in InfoSleuth by providing a common vocabulary, lexicon, or domain model. An ontology represents semantic concepts consistent across all the InfoSleuth agents, and is usually defined independently of the form and the availability of actual data syntax. This is different from an integrated global schema in a multidatabase system [19] whose construction is completely dependent on the local data.

The strengths of InfoSleuth are that it can adapt to a wide range of information retrieval and analysis tasks, and that it can also adapt itself to changes in the availability and capability of the different agents in a given InfoSleuth community. Some of the types of retrieval and analysis tasks include:

- gathering information via complex queries from a changing set of databases and semi-structured text repositories distributed across an internet,
- performing polling and notification for monitoring changes in data,

- analyzing gathered information using statistical and data mining techniques and/or logical inferencing, and
- noticing patterns in how information is changing that may indicate new trends or problems.

In addition to offering such a broad spectrum of information services, InfoSleuth communities also must be able to adapt to changes in their composition, including the addition, removal, or changing capabilities of agents, users, and resources. Much of this adaptability is accomplished by the use of semantic brokering.

Figure 1 depicts the current InfoSleuth agent architecture. The InfoSleuth agents are organized as core agents (those within the cloud) that provide basic information subscription, filtering and fusion capabilities, resource agents and service providers (those to the right of the cloud) that serve as interface to external information sources, and user agents (those to the left of the cloud) that act as proxies for individual users or groups of users.

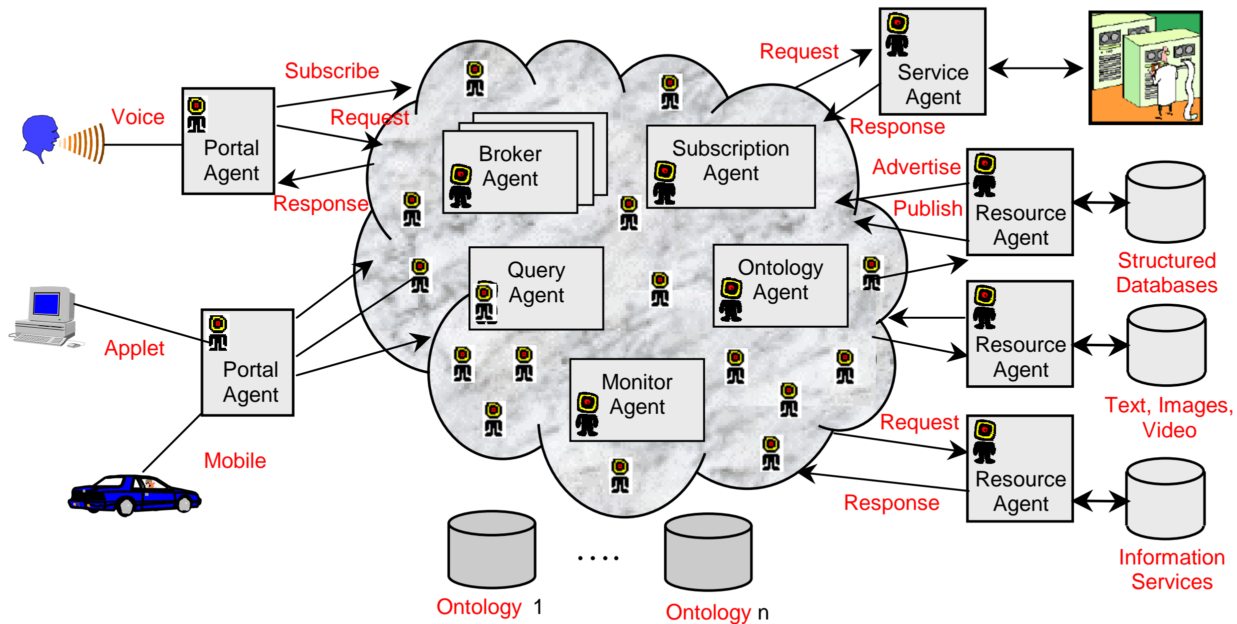

Figure 1: InfoSleuth: Dynamic and Broker-based Agent Architecture

In a given community of InfoSleuth agents, the core agents (e.g. broker agents, control agents, subscription agents, query agents, ontology agents and monitor agents) work together to connect users with the information resources that they need. These agents service requests over a set of common ontologies.

Because of the need to be able to adapt to the changes in available agents, InfoSleuth uses a sophisticated brokering process to match agents to different parts of the information retrieval and analysis tasks posed by its users. In the following section, we discuss this brokering process and indicate how it helps to enhance the flexibility and robustness of the agent community.

3 Brokering

In this section, we discuss how brokering is implemented in InfoSleuth. We begin by describing the broker agents and giving an overview of the brokering system. We then follow with some sections presenting details on specific aspects of brokering, and finish with an example.

3.1 Overview of the Brokering System

In InfoSleuth, the broker agents play a critical role in maintaining an up-to-date repository of all of the agents available for access within an agent system. Without the presence of brokers (or other agents that have similar capabilities), an agent would be unable to locate new agents that could provide services to it. This locational task is critical either in the case where the agent knows about no other agents that can provide a needed service or in the case where an agent is looking for *all* agents that can provide a service. For instance, when the agent system's purpose is to provide information, the broker is needed to ensure that *all* available agents that can contribute information are located, even in a system where such agents may come on- and off- line frequently.

Each broker agent in InfoSleuth maintains a knowledge base of advertised information about other agents, and uses this knowledge to match agents with requested services. For instance, a number of different agents may advertise that they can answer data requests in SQL. However, one agent may advertise that it contains a wealth of information about the healthcare domain, a second agent that it is familiar with the aerospace industry, and a third that its sole function is to do complex query processing to assemble related information from external sources. When an agent requests resources that can process SQL queries and contain healthcare information, the brokers' task is to return only the potentially relevant resources, in this case ruling out all but the agents that have healthcare information. Another important function the brokers provide is to reason over the constraints on the information content of the agent. If for instance, an agent has advertised that it knows about the healthcare domain, it can also advertise the fact that its subsection of that domain is restricted to podiatrists in Dallas and Houston. If the broker receives a query for information resources that do not overlap this subsection of the healthcare domain, it will not match this agent to the query.

This service that the brokers provide is called a *matching* service. The matching service requires two separate components. The first is to maintain a *repository* containing current and correct information about operational agents and the services that they can provide, via new and updated agent advertisements. The second is to accept queries and to match them with advertised agents represented in the repository.

3.2 Advertising and Querying Process

Advertising and querying in InfoSleuth relies on the use of capabilities specified over multiple, focused ontologies to represent both advertisements and queries to the broker. These focused ontologies represent various orthogonal aspects of agent capabilities. For example, there are distinct focused ontologies

advertising the following types of information:

1. the conversations used to communicate about the service, e.g. the *conversation* ontology,
2. the language interface to the service, e.g. the *sql* ontology,
3. the semantic description of the service, e.g. the *query_processing* ontology, and
4. the information the service operates over, e.g. the *healthcare* ontology.

Within an advertisement or query, a given capability is expressed over some set of focused ontologies pertinent to that capability. Each focused ontology in the capability is further constrained to represent exactly the subset of that ontology that is supported within that capability. The notion of focused ontologies and their use in advertising and querying is explained further in Section 3.3.

A broker maintains a current and correct repository containing information about operational agents via the use of agent advertising. When an agent comes online, it announces itself to one or more brokers by advertising to them, using the terms and vocabulary described in the focused ontologies (see Section 3.3). This is shown in Figure 2. Each broker that received the advertisement stores all of the advertised information about each agent in its repository. When an agent's set of available capabilities changes, the agent may re-advertise to each broker that it sent the original advertisement to, so the brokers can update the information in their repositories.

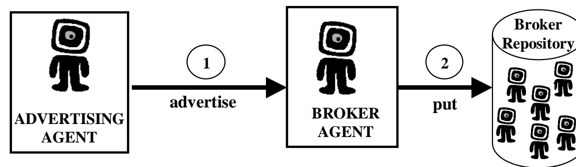

Figure 2: Advertising to the broker

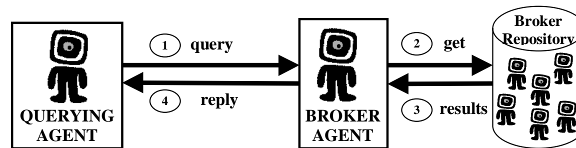

Figure 3: Querying the broker

When an agent goes offline, it first unregisters itself from the brokers it has advertised to. Also, the broker periodically pings each of the agents that have advertised to it, to discover any agents that have failed. The broker removes from its repository all information about agents that have failed to respond to several pings or have unregistered themselves.

Brokers may receive queries from agents that are looking for other agents that can provide specific services. A query is represented as an individual capability specified over some set of focused ontologies.

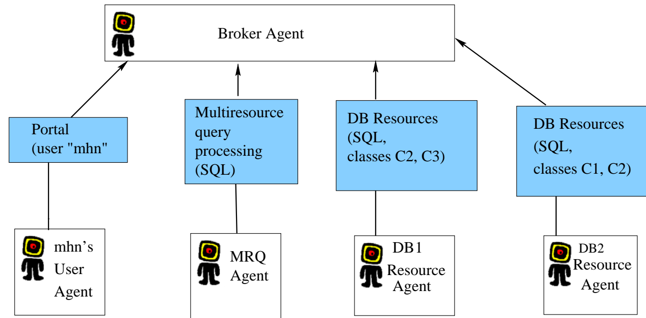

Figure 4: Agents advertising to the broker

A broker agent uses a special-purpose reasoning engine to do constraint-based reasoning over the query and advertisements to determine which agent advertisements match the services requested in the query. This set of matching agents found by the reasoning engine may be combined with those of other brokers, and is then returned by the broker to the requesting agent. This process is shown in Figure 3.

When a given broker agent receives a query, its reasoning engine iterates through each advertisement in its repository, checking whether the query capability matches one of the capabilities in the advertisement (see Section 4 for details). If there is a match, the agent that sent the match is added to the list of agents that will be returned from the broker. The return information may be annotated with additional information about the nature of the match, as requested by the agent that sent the query.

To illustrate this, we will show how the brokering is used to process a simple query over multiple resources in a single broker system. As each agent comes online, it advertises its capabilities as shown in Figure 4. Thus, after the actions in this figure, the broker has a repository containing four advertisements, one for each of the agents: Joe's User Agent, MRQ, DB1 Resource and DB2 Resource.

At some point in time, user Joe submits the SQL query select * from C2 to her user agent. At this point, Joe's User Agent must locate a query processing agent that can assemble all of the information that this community knows about class C2, as shown in Figure 5. In this figure, Joe's User Agent forwards a query to the broker agent asking for the multiresource query processing agent that can accept and process SQL queries. The broker agent replies, stating that MRQ fits the description in the request. The user agent then forwards the query to MRQ. Note that the resource agents are not involved.

When agent MRQ receives the query from the user agent, it looks at the query to determine which classes are required to answer the query, and discovers that it needs to look for class C2. It then queries the broker for all resource agents that can answer an SQL query involving class C2. This is shown in Figure 6. The broker processes the query and returns the responses DB1 Resource, DB2 Resource. MRQ then forwards a query to these two agents, receives the responses, assembles the result, and forwards it back to Joe's User Agent. Note that if the original query had been for class C3, then only the response DB2 Resource would have been returned to MRQ agent.

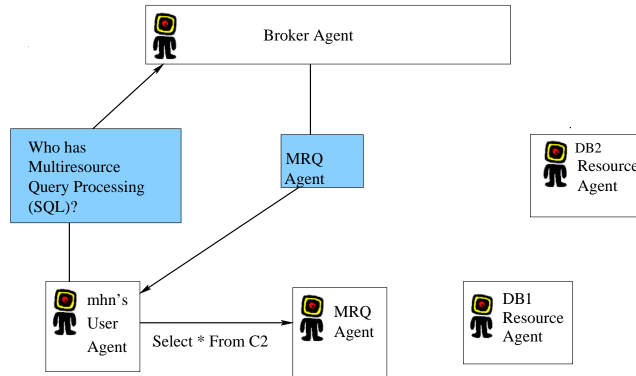

Figure 5: User agent querying the broker for general-purpose SQL query agents. Note the resource agents are not involved

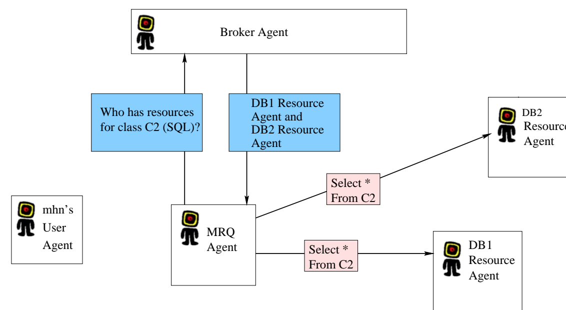

Figure 6: MRQ agent querying the broker for resource agents. Note that user agent is not involved

Let us assume for the moment that a new agent, MRQ2, comes online and advertises to the broker, stating that it is a multiresource query agent that has SQL interface and specializes in queries over the class C2. If now user Joe poses the same query to her user agent, then agent MRQ2 would be recommended to the user agent because it has a better semantic match to the request than does agent MRQ.

3.3 Focused Ontologies and Ontology Fragments

For an open, heterogeneous agent-based system to be functional, it is not enough to define a syntax of communication or even a semantics of the types of conversations that occur between agents. The information that the agents exchange must also be semantically consistent. The ontology must have enough concepts and give enough detail to allow the agents to provide useful information, but it cannot provide too much semantic detail, since this will quickly become overwhelming in terms of labor required to create the representations and the computational resources required to communicate and reason about the representations.

When we consider how to represent the ontological knowledge, there are basically two approaches. The first is to incorporate everything into a single, very large ontology that encompasses all of the knowledge that any agents will need to describe their capabilities. The second is to allow for multiple, focused ontologies that can be composed together.

The advantage of using a single, large ontology is that all relationships among different aspects of an

agent capability can be represented within advertisements and queries. Thus, the reasoning process can be tailored to operate efficiently within that ontology. When you consider the implications of a unified ontology, however, there are definite disadvantages. For instance, if agents can subscribe to multiple information domains, all information domains would be present in the ontology. The management of such an ontology would become very difficult, both from the point of view of merging in ontological concepts from a new domain and managing inter-domain relationships.

Given that expanding into a new domain may be a common operation for an open agent-based system, one could also envision having a single, large ontology per domain. This would mean, for instance, that concepts for information processing (such as subscription capabilities) would be included along with concepts for the specific information domain. This approach, however, would force the copying of common service concepts into different information domain ontologies.

Our solution, based on the notion of focused ontologies, segregates the ontological information into separate, orthogonal small ontologies. Each of these focused ontologies consists of a set of classes (possibly arranged in one or more class hierarchies), class attributes and relationships among classes. In the example of the previous paragraph, concepts relating to subscription would be in one focused ontology and information-domain-specific concepts would be in other ontologies. Table 1, for instance shows a simple service ontology for subscription and query services.

```
< ontology NAME="services" VERSION="1.0" >
  <class NAME = "subscription_query">
     <slot NAME = "computation" > </slot>
     <slot NAME = "adaptability" > </slot>
     <slot NAME = "response-type" "> </slot>
  </class>
  <class NAME = "query">
     <slot NAME = "computation" > </slot>
     <slot NAME = "response-type" </slot>
  </class>
  <class NAME = "data-response">
     <slot NAME = "delivery" > </slot>
     <slot NAME = "language" > </slot>
     <slot NAME = "annotations" </slot>
  </class>
</ontology>
```

Table 1: Simple service ontology for subscriptions and queries

Agents would represent their capabilities over this ontology and their respective domain ontologies. Unfortunately, if an agent implicitly maintains any constraints among the terms in the focused ontology that it uses to describe its capabilities, these may not be representable explicitly using this methodology. However, our experience shows that functionality and information concepts are often partitioned, with information domain ontologies already existing, and ontologies over other knowledge such as communication aspects or services also cleanly factorable into separate units. On the other hand, one clear advantage of this approach is that it is well-suited to open environments, as adding new focused ontologies to the agent

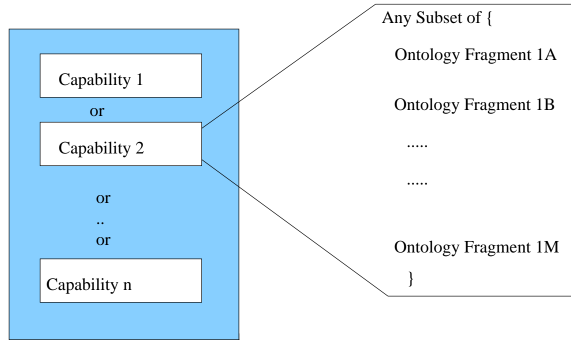

Figure 7: Agent Capabilities and Advertisements

system is straightforward.

The use of focused ontologies leads us to express information about the capabilities of agents in terms of *ontology fragments*. Each ontology fragment is specified against a single, focused ontology; therefore it is tagged with the ontology name and version against which it is specified. The focused ontology is constrained within the fragment to contain only the supported classes, slots, and values of slots. These constraints encapsulate the subset of the focused ontology that is supported by some aspect of the agent's functionality.

3.4 Agent Capabilities, Advertisements and Queries

Agent capabilities in InfoSleuth provide a mechanism for agents to advertise their abilities and characteristics to a broker agent, and to query the broker for sets of agents meeting specific criteria. An *agent capability* can then be defined as a conjunction of any number of separate ontology fragments. We treat the ontology fragments orthogonally, assuming that the concepts are independent. While we recognize that there may be deeper semantic relationships between the ontology fragments, we have chosen not to explicitly represent these relationships, as this would add a significant amount of complexity to the capabilities and the reasoning engine. In practice, we have seen that the agents that use the capabilities can adequately disambiguate and infer these relationships themselves.

Note that our composition of individual ontology fragments into capabilities is a very general structure that can provide us with a uniform view of information and services. This makes the representation and reasoning over the representation extremely general. In practice, we have used it successfully not only for agents that encapsulate information sources, but also agents that encapsulate new and legacy services, core agents such as broker agents, and agents that enact simple workflows.

An agent's advertisement declares its capabilities to a broker. An advertisement contains one or more agent capabilities. When more than one agent capability is advertised, the semantic interpretation is similar to a disjunction: the agent is capable of any one of the advertised capabilities, and each capability is considered independently. This is shown in Figure 7. Within the ontology fragments of an advertised capability there will appear one or more classes from the ontology, meaning that the agent supports any subset of

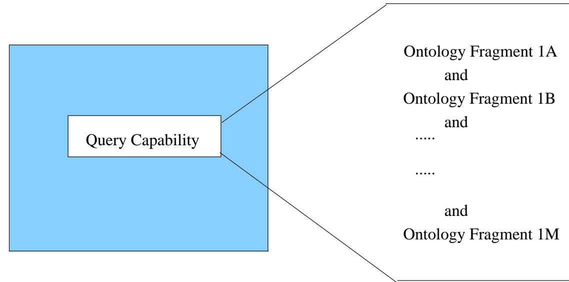

Figure 8: Agent Capabilities and Queries

those advertised classes. An advertised class can contain one or more advertised slots and, similar to the classes, this means the agent supporting any subset of these slots. Finally, there can be constraints on an advertised slot, which represent the range of values of that slot that are present in the underlying data, where the interpretation is either a disjunction or union of all the constraints. The slot's values are considered to be unconstrained when there are no advertised constraints.

A query consists of a single capability, along with some directions on the information to be returned if a match is found. A query capability is a conjunction of ontology fragments (see Figure 8), where each fragment can specify one or more classes, slots from each class and constraints on the slots. However, since this appears in the context of asking a question, it will have slightly different semantics in the reasoning engine. In particular, for a query capability to match an advertised capability, the query classes must be a subset of the advertised classes (possibly taking into account the class-subclass hierarchy), and the constraints on the query classes must intersect the constraints on the advertised classes.

3.5 Example

In this subsection, we show examples of advertisements and queries in XML. The tags used in the specifications are defined by the Document Type Definitions (DTDs) as shown in Table 2.

Advertisement Table 3 presents an example of an advertisement from ResourceAgent5 as generated in the InfoSleuth system. Resource agents are the back-end agents within InfoSleuth that act as proxies for structured or semi-structured repositories. The locational information is advertised using _infoSleuth ontology fragment. It tells the broker that ResourceAgent5 can be contacted at port 4356, on the host machine b1.mcc.com. The fragment of the _conversation ontology specifies that the agent accepts conversations of the form ask-all and ask-one using the language KQML. The fragment of the sql ontology specifies that it knows how to execute queries specified in SQL select-statement. The fragment of the healthcare ontology specifies that it deals with diagnosis and patient classes of the healthcare domain and the patient data is restricted to patients between the age of 43 and 75. The fragment of the service ontology consists of two classes data-response and query. The characterics of data-response services

```
<!ELEMENT message (advertisement|query|..)>
<!ELEMENT advertisement (capability+) >
<!ELEMENT query (capability+) >
<!ELEMENT capability (ontology_fragment+) >
<!ATTLIST capability NAME CDATA #REQUIRED>
<!ELEMENT ontology_fragment (class* ) >
<!ATTLIST ontology_fragment
        NAME CDATA #REQUIRED
        VERSION CDATA "1.0"
        RETURN_CLASSES CDATA "false"
        CLASS_SEMANTICS CDATA "any"
<!ELEMENT class (slot* key* constraint_conjunct?) >
<!ATTLIST class
        NAME CDATA #REQUIRED
        QUERIED_NAME CDATA " "
        RETURNED_SLOTs CDATA "false"
        RETURNED_KEYS CDATA "false"
        INFERENCE_DEPTH CDATA "-1"
        SLOT_SEMANTICS CDATA "any" >
<!ELEMENT slot (set_constraint||interval_constraint)* >
<!ATTLIST slot
        NAME CDATA #REQUIRED
        RETURNED_CONSTRAINTS CDATA "false"
        VALUE CDATA "">
...
```

Table 2: Sample DTD for advertisements and queries

are described by its slot values. The delivery slot value specifies that results can be delivered offline using http, the annotations specifies that results can be annotated with additional information such as the originating sources, and the language specifies that results are displayed in relational database tabular form. The class query specifies that the computation of the data is direct (accessed locally within the agent). The slot value of response-type is first-solution which specifies that the agent will return the first solution as soon as it becomes available. Upon receipt of this advertisement, the broker parses and validates the advertisement and asserts it in its repository.

Querying Table 4 shows the content of a query to the broker to find which resource agents can answer a request for patients between the age of 25 and 65 with diagnosis code 40w. The source that can provide this information must be able to accept query expressed in sql syntax and can compute the answers to the query locally.

A query is specified as a *single* capability over different ontology fragments. A query needs to explicitly specify the type and the amount of information to be returned. When attribute values RETURN_SLOTS, RETURN_CLASSES, RETURN_CONSTRAINTS and RETURN_KEYS are set to the value true, these respective information in an ontology fragment will be returned. A query also specifies the semantic interpretation of the class and slot values of each ontology fragment using CLASS_SEMANTICS and SLOT_SEMANTICS respectively. A value of all specifies that all the queried classes or slots have to match. A value of any specifies that the query only need to match any one of the slot queried.

The details of how the matching of a query to advertisments is done are discussed in section 4. Note

```
<advertisement>
  <capability NAME=”ResourceAgent5Cap”>
    <ontology_fragment NAME=”_infosleuth” VERSION=”1.0”>
      <class NAME=”agent”>
        <slot NAME=”name” VALUE=”ResourceAgent5”> </slot>
        <slot NAME=”host” VALUE=”b1.mcc.com”></slot>
        <slot NAME=”port” VALUE=”4356”></slot>
        <slot NAME=”type” VALUE=”resourceagent”></slot>
      </class>
    </ontology_fragment>
    <ontology_fragment NAME=”_conversation” VERSION=”1.0”>
      <class NAME=”conversation”>
        <slot NAME=”type”>
          <set_constraint>
          <![CDATA[”ask-all”, ”ask-one”]]>
          </set_constraint>
        </slot>
        <slot NAME=”name” VALUE=”kqml”></slot>
      </class>
    </ontology_fragment>
    <ontology_fragment NAME=”sql” VERSION=”1.0”>
      <class NAME=”select-statement’></class>
    </ontology_fragment>
    <ontology_fragment NAME=”healthcare” VERSION=”1.0”>
      < class NAME=”diagnosis”>
        < slot NAME=”diagnosis-code” > </slot>
      </class >
      < class NAME=“patient”>
        < slot NAME=”patient-age” > </slot>
          < constraint_conjunct >
          < set_interval>
            MIN_VALUE=”43”
            MAX_VALUE=”75”
          </set_interval>
          < /constraint_conjunct >
        < key NAME=”patient-id” > </key>
      </class>
    </ontology_fragment>
    <ontology_fragment Name=“service” VERSION=“1.0”>
      < class NAME=“data-response”>
        < slot NAME=“language” VALUE=“tuple-format”> </slot>
        < slot NAME=“delivery” VALUE=“http”> </slot>
        < slot NAME=“annotations” VALUE=“source-tagging”> </slot>
      </class>
      < class NAME=“query”>
        < slot NAME=“computation” VALUE=“direct”> </slot>
        < slot NAME=“response-type” VALUE=“first-solution”> </slot>
      </class>
    </ontology_fragment>
  </capability>
</advertisment>
```

Table 3: Advertisement of resourceAgent5

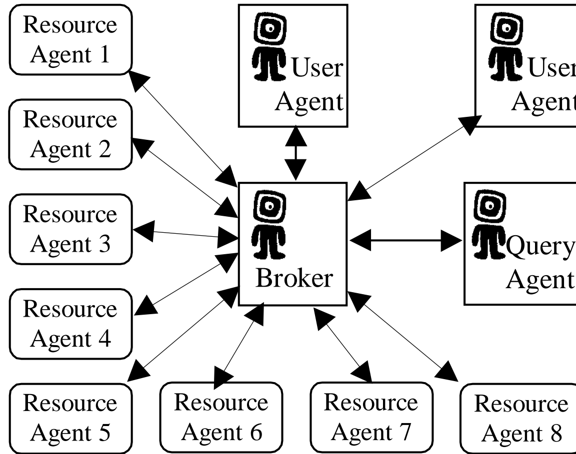

Figure 9: Single Broker Architecture

that the reasoning engine would match the agent that advertised knowledge about patients between 43 and 75 (i.e., ResourceAgent5).

4 Single Brokering

4.1 Single Broker Architecture

In a single broker architecture, the broker is the central repository for all information about available agents and resources in the system. Rather than caching information about the entire system, when agents come online they need only know the possible locations of the broker and how to query it for information to locate other agents in the system. Each agent then advertises itself to the broker and queries the broker when it needs to locate other agents. The single broker architecture and its relationship to the InfoSleuth agents that advertise and query to it is illustrated in Figure 9. In this view of the system, there are eight resource agents (Resource Agents 1-8) advertised to the broker, two user agents and one query agent.

4.2 Broker Reasoning Engine

In this section, we describe the algorithms we use to search through the respository and check for matches between advertisements and queries.

4.2.1 Iterating Through the Repository

When a broker receives a query, it must determine which of its advertisements match the query. This is done as follows:

1. Iterate through all advertisements in the repository, checking if the advertisement matches the query.

2. To see whether an advertisement matches the query, iterate through the capabilities of query making sure each one matches one of the capabilities of the advertisement. The algorithm to do capability matching is described in the next section.

```
<query>
  <capability NAME=“_generic_query_capability”>
    <ontology_fragment NAME=”_infosleuth” RETURN_CLASSES=“false”>
      <class NAME=“agent” RETURN_KEYS=“false” RETURN_SLOTS=“false” SLOT_SEMANTICS=“all”>
        < slot NAME=“name” RETURN_CONSTRAINTS=“true”> </slot>
        < slot NAME=“host” RETURN_CONSTRAINTS=“true”> </slot>
        < slot NAME=“port” RETURN_CONSTRAINTS=“true”> </slot>
        < constraint_conjunct >
          < constraint_disjunct >
            < slot NAME=“type” VALUE=“resourceagent” > </slot>
          < /constraint_disjunct >
        < /constraint_conjunct >
      </class>
    </ontology_fragment >
    <ontology_fragment NAME=“sql” RETURN_CLASSES=“false”>
    </ontology_fragment >
    <ontology_fragment NAME=“healthcare” RETURN_CLASSES=“true” CLASS_SEMANTICS=“any”>
      <class NAME=“patient” RETURN_KEYS=“true” RETURN_SLOTS=“true” SLOT_SEMANTICS=“any”>
        < constraint_conjunct >
          <slot NAME=“patient-age” RETURN_CONSTRAINTS=“true” >
              < set_interval >
              MIN_VALUE = “25”
              MAX_VALUE = “65”
              </set_interval >
        < /constraint_conjunct >
      </class>
      <class NAME=“diagnosis” RETURN_KEYS=“false” RETURN_SLOTS=“true” SLOT_SEMANTICS=“any”>
        < constraint_conjunct >
          <slot NAME=“diagnosis-code” RETURN_CONSTRAINTS=“true” >
              < set_constraint >
              <![CDATA[“40w”]] >
              < /set_constraint >
        < /constraint_conjunct >
      </class>
    </ontology_fragment >
    <ontology_fragment NAME=“services” RETURN_CLASSES=“false”>
      <class NAME=“query” RETURN_KEYS=“false” RETURN_SLOTS=“false” SLOT_SEMANTICS=“all”>
        < slot NAME=“computation” RETURN_CONSTRAINTS=“true”> </slot>
        < constraint_conjunct >
          < constraint_disjunct >
            < slot NAME=“computation” VALUE=“direct” > </slot>
          < /constraint_disjunct >
        < /constraint_conjunct >
      </class>
    </ontology_fragment >
  <capability>
<query>
```

Table 4: Content of QueryAgent2

3. If all the query capabilities have a match, add the queried information from the advertised capability to the result list.

4. Once all capabilities have been checked, return the result list to the agent that sent the query.

Note that it is possible for more than one of an agent's advertised capabilities to match the query.

4.2.2 Matching an Advertised Capability to a Query Capability

Matching of capabilities is effected using a top-down process.

Matching Queries with Advertisements At the top-most level, the set of ontology fragments in the capability of a query is interpreted as a conjunction. An advertised capability matches this query if it has advertised at least all those ontology fragments, and each of the individual queried fragments matches the corresponding advertised fragments, where the matching criteria is discussed below. Note that the advertisement can contain other ontology fragments that do not appear in the query, but all the ones that do appear in the query must have a matching fragment in a particular advertised capability.

Matching Ontology Fragments First, the names of the ontologies these fragments are associated with must be identical. Next, the advertised classes must match the queried classes, but here the query has the option of specifying the set of classes as either a conjunction or a disjunction by setting CLASS-SEMANTICS to all or any. For a conjunction of queried classes, the advertised ontology fragment must contain at least all of the queried classes and the classes must match (the matching criteria is defined below). Note that the advertised ontology fragment may have more classes than are queried. For a disjunction of classes, it is enough for any one of the queried classes to match one of the advertised classes.

Matching Ontology Classes A queried class can contain one or more queried slots. As with classes within an ontology fragment, a queried class has the option of querying on a conjunction or disjunction of slots by setting SLOT-SEMANTICS to all or any. A conjunction means all of the queried slots must match some advertised slots. A disjunction means that at least one of the queried slots must match in the advertised class.

With the ontology fragments, the first criteria for matching required the clases to have the same name. However, since ontologies may contain class hierarchies, matching of advertised and queried class names differs. In particular, class names match if either they are identical or the advertised class is a subclass of the queried class.

In a queried capability, every class specified has a taxonomic inference depth to help bound the subclass traversal in the reasoning engine. The class inferencing is always from the queried class to its subclasses, since matching on super-classes could result in agents whose data does not answer the query being asked. The inference depth represents the maximum number of class-subclass relationships to traverse

while checking if an advertised class is a subclass of the queried class, where zero means that the advertised and queried class names must be identical, infinity means that there is no limit, and all other values represent some finite limit.

Matching Ontology Slots For a query slot to match an advertised slot, it must have the same name and there must be a non-empty intersection between the queried and advertised constraints. Both the advertised and queried constraints represent a subset of all the possible values a slot can have, so a slot that matches with a non-empty intersection semantically means that it is at least possible that it will have relevant information or services.

5 Multibrokering

Distributed multibrokering allows the matching process to be distributed across multiple collaborating brokers, each representing a different set of agents. When a broker receives a request for an agent with specific capabilities, it looks for matches in its own repository of agent information and may also query other brokers to find external agents with the needed capabilities. In this section, we present some principles and methodologies we have used in building a scalable multibroker system.

5.1 Principles for Scalable Multibrokering

The major goals of multibrokering center around robustness, flexibility and scalability. Multibrokering allows processing load to be more evenly distributed around the system, as well as allowing the parallel development of more precise reasoning over narrower domains. The following principles guided us in the development of InfoSleuth multibrokering.

Peer-to-peer Architecture We use a peer-to-peer topology for inter-broker connectivity. Peer-to-peer brokering allows brokers to freely advertise and unadvertise themselves to other brokers, so entry to and exit from the group of brokers is easy. Newly advertised brokers' data will be integrated into each new search as if the data had always existed, and brokers which have unadvertised themselves will simply cease to exist so far as the rest of the system is concerned.

Peer-to-peer brokering is more scalable because it allows brokers to freely advertise and unadvertise themselves to other brokers. More importantly, this topology ensures that there is not a single point of failure, as would be found in (say) a hierarchical brokering system. The only major disadvantage of a peer-to-peer architecture is the cost of inter-connection. When the number of brokers become very large, the connectivity cost could be significant. However, we may be able to reduce the connectivity cost on a per-search basis using known techniques from graph theory.

Non-broker agents must advertise Non-broker nagents must advertise their capabilities to at least one broker. Robustness increases if agents advertise redundantly to several brokers. This ensures that an agent

is still available if one of the brokers it has advertised to goes down. In theory, it is possible for every non-broker agent to connect or advertise to every broker in the system. This would be very robust, but is impractical for large numbers of agents and brokers.

It is the agents' responsibility to ensure that the different copies of their advertisements are kept consistent in all the brokers to which they advertise. This shifts the responsibilities for ensuring correctness to the agents that know already when advertisements are inconsistent.

Brokers may specialize In a world with multiple systems and brokers interoperating, brokers may start to consolidate advertisements about related domains and services. In this case, an agent should take care to ensure that it advertises to the brokers that best represent its interests. For example, if a food supplier agent advertises to a broker that only brokers healthcare information, the broker should forward it to a broker that can deal with food suppliers. If no such broker exists, it may reject the advertisement. With this approach, the brokers will need metrics to measure how well the advertisement fits within the broker's advertised purpose. When brokers specialize in certain domains, it is possible to develop optimized reasoning over a narrower domain and hence lead to better performance when the number of agents and brokers becomes very large. To prevent the possibility of all brokers rejecting some agent whose advertisement fits in with no broker, cooperating brokers should contain at least one general-purpose broker for queries not covered by the specialized brokers.

5.2 Collaborative Reasoning

In a multibroker environment, a broker not only keeps information about other agents in its repository, it also keeps information about other brokers that it knows about. Brokers are connected in a directed graph structure, with the nodes representing brokers and the arcs representing knowledge of other brokers' current advertisements. Thus if Broker1 has advertised itself to Broker2, there is an arc from Broker2 to Broker1. The connectivity should be sufficient to ensure that each broker is either directly or indirectly connected to all the other brokers in the system (i.e. no disconnected sub-network of brokers). Figure 10 shows four fully interconnected brokers with eight resource agents, and some user and query processing agents.

Each broker maintains, in addition to its repository of advertised agent information, a reasoning engine that matches queries to its own set of agent advertisements. This reasoning engine may also match broker queries to other brokers that may contain advertisements for other agents that can service the query, thus enabling a broker to locate other brokers that fit certain criteria. Each broker request is forwarded to relevant other brokers, which then may propagate the requests further. Searches are restricted internally to prevent undesirable propagation of requests, e.g., cyclical propagation and prolonged search in a very large system. The response to the broker query contains the union of all brokers' responses.

In a multibrokering environment, there is also the question of when to start looking at other brokers when processing a brokering request. Suppose a broker gets a request for information about database resources dealing with the healthcare industry in Dallas, Texas, but finds that it does not have any information stored

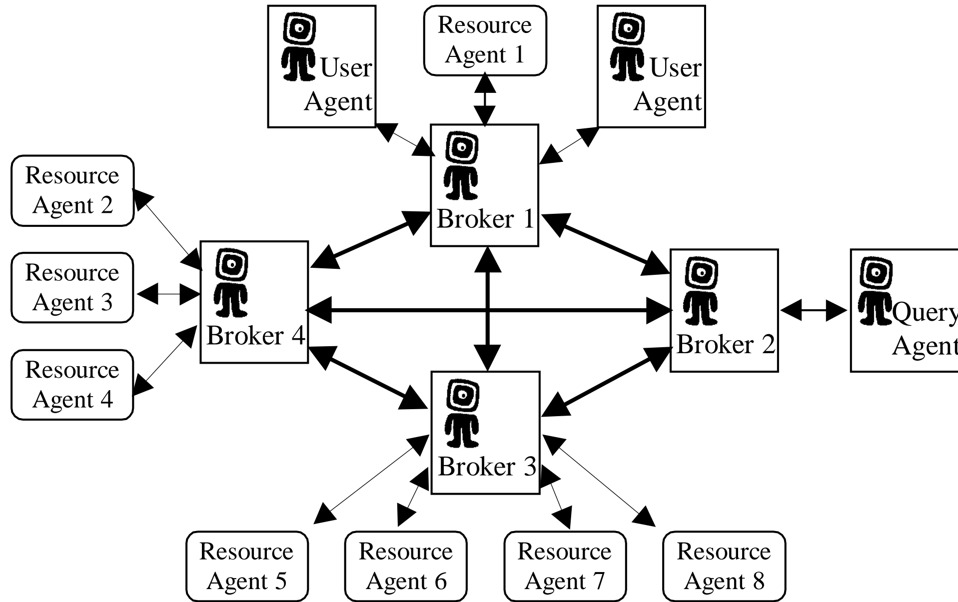

Figure 10: Multibroker architecture

about agents that would meet these particular criteria. The broker then initiates an inter-broker search, guided by the nature of the request. Generally, if the original query was for finding one agent with the requested capability, brokers are searched one-by-one in a breadth-first manner until a match is found. If the original query was for finding all the agents with the requested capability, the brokers are searched in parallel until all accessible brokers have been queried, and the matches are assembled into a single response. A special property, hop-count, defines how many hops through the broker digraph should be traversed for a given query before the brokers should give up. The search mechanism also terminates a branch of the search once a broker loop has been detected.

As shown in Figure 10, when one of the other brokers receives the request from Broker1, each of these brokers then use the same procedures as were used by Broker1 to match for potential agents that can provide the requested service. Eventually, the other brokers return their results to Broker1, which combines them with its own (possibly empty) list of providing agents, eliminating duplicated entries. Broker1 then returns the combined list to the requesting agent.

5.3 Integrating New Brokers

New brokers make themselves known to other brokers through advertisements. Each broker is configured with either a list of other brokers, or a well known port, over which it should advertise. By default a broker must advertise at least its locational information. However, brokers also could advertise their capabilities to other brokers, who could then reason over the brokers' capabilities during an inter-broker search and eliminate brokers that should not be contacted, thus improving the processing time by ruling out unnecessary queries. This feature has not been well-tested.

Each non-broker agent is configured with a list of known brokers ordered by preference to connect to on startup. This known-broker-list represents its initial entry point(s) into the brokering system. Once operational, the agent can change to a different broker, even one that is not in its known broker list. To do this, it sends a query to a known broker for one or all of the brokers that are available in the system with the capabilities and data domain that it is interested in. It then picks one in the list to use. Alternatively, the agent might use a known broker and keep a history of how this broker handles its request. If over a period of time, the system discovers that the known broker always forwards the request to a different specific broker or set of brokers, the system will add the new brokers to the known broker list.

5.4 Maintaining Connectivity

Redundant advertising and robust connectivity are keys to maintaining a reliable network of brokers. The redundant advertising algorithm works to ensure that the number of outstanding advertisements is maintained at a configurable maximum. The robust connectivity algorithm works to ensure that, if an agent has advertised to a broker, that broker is still operational and still knows about the agent. These algorithms work in tandem to maintain the agent-broker connectivity.

Redundant Advertising Redundant advertising is the practice of agents and brokers making identical advertisements of their services to more than one broker. Every agent or broker maintains a configuration parameter defining how many brokers that agent or broker should advertise to. All agents, including broker agents, keep track of two lists of brokers: a list of brokers that they know about (known-broker-list), and a list of brokers they have successfully advertised to (connected-broker-list).

The advertisement process occurs when an agent or broker comes up, and periodically thereafter. The agents are configured with a maximum number of brokers to maintain contact with at a given time / advertisements outstanding at a given time. Each agent or broker advertises to brokers on the known-broker-list but not on the connected-broker-list, until the connected-broker-list reaches this configured maximum. When an advertisement is successful, the broker that kept the advertisement is added to the connected-broker-list. Once either the number of such connected brokers reaches the configured number of redundant advertisements or all known brokers have been tried, the advertisement process stops. This process is repeated periodically to ensure that appropriate connectivity is maintained. If at the end of a given advertisement period the agent is connected to no brokers, it will enter a dormant state, wait until the next advertisement period, and attempt to reconnect.

During operation, an agent may also discover or be informed of other brokers that it deems appropriate to advertise to. In this case, the agent adds them to its known-broker-list.

Robust Connectivity In the event that a broker should unexpectedly leave the agent community, it is the responsibility of each agent connected to is to detect that the broker has left and re-initiate the advertising process. Agents periodically test to see whether all of the brokers they have advertised to still know about

them. At some (configurable) periodic interval, agents cycle through the connected-broker-list, and query each broker in turn to see if it still knows about them.

If a broker has gone offline, either the agent will be unable to make the connection to the broker or the broker will fail to respond. In the event that a broker is alive but does not have information about the agent that is doing the querying, the agent will receive a reply containing no matches from the broker that it queried. In either case, the agent will remove this broker from its connected-broker-list. However, as long as the agent has advertised redundantly to multiple brokers, it will still be locatable via the remaining brokers that it has advertised to.

Once an agent has completed the above polling cycle, having successfully traversed its connected-broker-list, it checks to see if it needs to re-advertise. This will occur when the connected-broker-list is shorter than the configured maximum number of redundant advertisements. If this is so, it re-initiates the advertisement process as described in the previous section.

6 Experimental Results

This section presents two types of empirical evaluations of InfoSleuth's multi-brokering mechanisms. In section 6.1 we present some experiments done using the InfoSleuth system directly. However, because running multiple, large-scale experiments with InfoSleuth poses some difficulty, in section 6.2 we discuss some of these difficulties and present some simulation-based results.

6.1 Multibrokering behavior in InfoSleuth

We performed a set of experiments using the InfoSleuth system directly to determine if multibrokering was feasible and if specialization helps. We tested the response time for queries under different configurations of resource agents, number of brokers, number of query agents and user agents. Each experiment consisted of issuing SQL queries (encapsulated in KQML messages) to an InfoSleuth agent community for processing, which induces an indirect load on the brokering system. The response time is the total time for the user to get the result displayed on the screen from the time the query is submitted. This includes CPU, disk I/O, communication among agents and graphical display of results.

Table 5: Experimental Query Streams

name	# Resource Agents
SA (single agent)	1
DA (double agent)	2
4A (four agent)	4
VF (vertical fragmentation)	4
CH (class hierarchy)	4
FH (fragmentation & class hierarchy)	4

The types of queries submitted are characterized in Table 5, and are representative of the majority of the

types of queries that InfoSleuth currently handles. Their precise meanings are:

- SA - single agent, represents the set of queries which only accesses one resource agent.
- DA - double agent, represents the set of queries which accesses two different resource agents.
- 4A - four agent, represents the set of queries which accesses four different resource agents.
- VF - vertical fragmentation, represents the set of queries which accesses four different resource agents and reasons over fragments of entities from different resources.
- CH - class hierarchy, represents the set of queries which accesses four different resource agents and reasons over class-subclass relationships.
- FH - fragmentation and class hierarchy, represents the set of queries which accesses four different resource agents and reasons over both fragmentation and class hierarchies.

Table 6 summarizes the different query streams used for the different experiments. In the multibroker case, each broker is running on a different Sparc Ultra 1 machine. The single-broker variant of each experiment has all the agents and a single broker running on a single Sparc Ultra 1 machine. All of the Sparc Ultra 1 machines were running the SunOS 2.5 operating system.

Table 6: Experimental configurations

Experiment	4A	DA	SA	VF	FH	CH	#Resource Agents
A	√						4
B	√	√	√				4
C	√	√	√	√			8
D	√	√	√	√		√	12
E	√	√	√	√	√	√	16

Each experiment is repeated 3 times. The difference between different experiment configuration is in the number of resource agents used and the type and number of query streams used. Table 7 shows the average response time expressed as a ratio of multibroker/single broker for each of the above type of experiment. A ratio of less than 1.0, implies improved performance of multibrokering over single brokering under that type of query stream.

Table 7: Experimental results

Experiment	4A	DA	SA	VF	FH	CH
A	1.00					
B	1.04	1.05	1.01			
C	1.12	1.01	1.05	0.85		
D	0.98	0.95	0.91	0.77	0.86	
E	0.3	0.31	0.47	0.76	0.63	0.67

When the system is underloaded (Configuration in experiment A-C), the response time for queries is slightly better in a single broker versus a multibroker system (the ratio is greater than 1.0). However, the difference is less than 0.1 in most cases. Thus we can conclude that the response time for a query did not degrade with using multibroker. On the other hand, when the system is loaded (Experiment D-E), the response time in multibroker systems is better for all the queries.

We also conducted a sixth experiment (F) to check the effect of specialization of brokers in a multibroker environment. This experiment used the same agents and query streams as Experiment E, but with all the resources associated with a given query stream (such as the type of resource agents, the query agents and the user agents) kept at a single broker. Table 8 shows the average response time expressed as a ratio of multibrokering with specialization/multibrokering without specialization.

Table 8: Experiment F Results

Expt	4A	DA	SA	VF	FH	CH
F	0.86	0.86	0.87	0.74	0.6	0.29

This experiment shows that there is an improvement in response time for all the above type of queries with specialization of brokers (ratio less than 1.0). Intuitively, this is because the individual brokers reason over less information, and therefore the reasoning is more straightforward and less costly. Moreover, less inter-broker communication cost is incurred due to specialization. The effect of inter broker communication cost is discussed in more details in the section on simulation-based experiments.

6.2 Simulation-based Experiments

There are many obstacles involved with running large scale experiments using actual agent applications, many of which we experienced while conducting the experiments of the previous sub-section. The mechanisms for managing very large numbers of agents do not exist in the current InfoSleuth system nor do they exist in any other agent system at present. This makes the set-up, execution, monitoring and result gathering for the experiments extremely difficult, especially as the number of agents to be managed increases. Additionally, to run experiments with hundreds of agents requires enough resources for all the agents to run; including both hardware and time. Furthermore, if these are real agents, they will each need to be configured, possibly requiring fabricating enough data for all the agents to use. Another factor in trying to evaluate an isolated portion of the InfoSleuth system, which is a complex prototype, is that non-essential components with less than optimal implementations can degrade the performance to the point where it masks the true impact of the system characteristic being evaluated. Finally and more generally, evaluation of agent system properties is often desired at design-time before any application exists.

A simulation-based approach to evaluating an agent system overcomes all of these obstacles. A simulation gives complete control over all of the agents; it doesn't need to run in real time; each agent requires far fewer resources; there is no need for real data; and you can model only the relevant characteristics that

affect the performance of the system, eliminating any negative effects caused by non-critical components.

At present, the resource constraints have made it very difficult to run controlled experiments in the existing InfoSleuth system, especially when there are more than a few dozen agents. Because of this and the desire to demonstrate that multi-brokering is robust and can scale up to non-trivial numbers of agents, we have used a simulation-based approach for evaluation. The agent simulation is built upon a discrete-event simulator, modeling both machine characteristics and network connections. The broker behaviors were implemented to closely mimic the behaviors of the brokers in the actual InfoSleuth system. Below, we present a broad overview of the simulator, since space constraints of the paper prevents a more comprehensive description.

There were fewer types of agents used in the simulation experiments than were used in the InfoSleuth experiments. Since we wanted to focus on the broker characteristics, we limited the types to broker, resource and query agents. The query agents are simply a mechanism for putting a load on the brokers, while the resource agents simply defined the amount and type of information the brokers have to reason about. The focus of the experiments is on the performance of the brokers under various configurations.

6.2.1 The Simulator

The simulator was built in-house at MCC as part of an independent project. Below we provide a brief description of the simulator as space requirements prohibit a more detailed discussion. The main components of the simulator are:

- **processor model** - each agent is simulated to run on its own processor and the speed of each processor is set to be identical for each agent.
- **network model** - all inter-agent communication is modeled to be carried by a network whose simulated bandwidth was set to be 250 kilobytes per second with a set-up latency time of 0.1 seconds per message.
- **hardware reliability model** - for the experiments that test the robustness of multibrokering, both individual processors and network connections have a failure model that causes them to be inoperable according to an exponential distribution.
- **common agent model** - common to all the simulated agents is a parameter specifying the maximal amount of time one agent will wait for another to reply, which was set to 300 seconds.
- **resource agent model** - this model allows simulation of different ontologies and the experiments were set up so that for any single query on an ontology, there would be 4 resource agents capable of satisfying the query. In addition, there are simulation parameters that model the complexity and coverage of the query as well as the amount of data that the resource agent was managing.

- **query agent model** - query agents generate the load on the system according to an exponential distribution, randomly choosing a broker when it was connected to multiple brokers. Queries are also randomly assigned parameters specifying its complexity, coverage and the ontology required.
- **broker agent model** - parameters for the broker agent include the speed of the broker reasoning engine and the size of the resulting responses, both of which will depend on the number and sizes of the advertisements it is maintaining.

Multi-brokering The multi-brokering behaviors in the simulator were tailored to mimic those of the InfoSleuth system. In particular, is can simulate the effects of the various interbroker search patterns. For all experiments here, all relevant brokers were searched.

6.2.2 Simulation Results

In this section we present some results from our simulation experiments. First we show the inherent problems with single broker networks and how broker specialization compares to a system where all brokers are replicates, maintaining complete knowledge of all other agents. The second set of experiments explores the scalability of a multi-brokering system with the behavioral characteristics of the InfoSleuth brokers. Multi-brokering imposes extra overhead due to the communication between the brokers, and we want to ensure that as the number of agents in the system increases that this overhead does not degrade the overall performance. The final set of experiments demonstrates how a multi-brokering system provides robustness. In this experiment, individual broker fails using an exponential distribution with varying means.

Each individual experiment was the simulation of 4 hours of system execution time. Because the simulations are based upon pseudo-random inputs, we ran each set of experiments 10 times and averaged the results. This helped ensure that we were not reporting results from a particular anomalous pseudo-random number sequence.

Single versus Multiple Brokers Our first set of experiments were aimed at demonstrating that a single broker architecture was inferior to multiple broker systems for moderate to high query loads. In this experiment we used 24 resource agents and vary the mean time between queries from 5 to 30 seconds in increments of 5 seconds. In the single brokering arrangement all 24 resource agents are in a single repository and the parameters of the experiment dictate that each query will take a minimum of 24 seconds to process. Thus, only when the query frequency is higher than once every 24 seconds does the single brokering scheme begin to give reasonable response times. In contrast, the two multiple broker architectures begin to give acceptable responses for query frequencies as low as every 10 seconds. The two types of multiple broker arrangements are explained and contrasted in the subsequent paragraph.

Replication versus Specialization One of the multiple broker arrangements used in the first experiment consists of replicated brokers where there are 20 brokers, each one having identical copies of all 24 resource

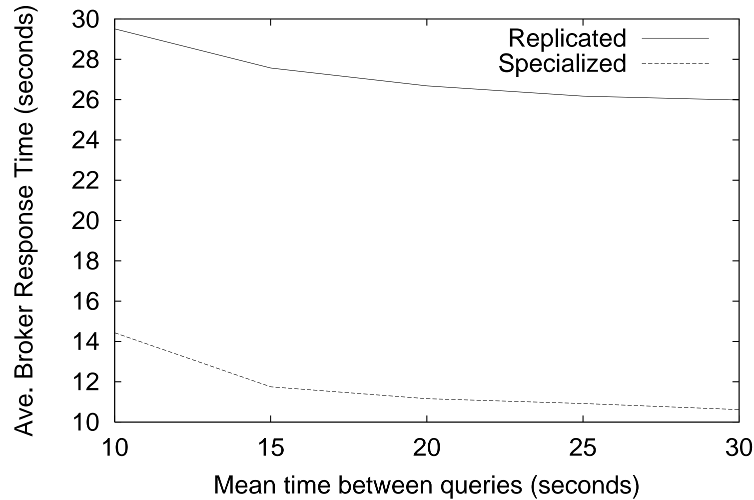

Figure 11: Replicated brokering versus specialized brokering with 20 brokers and 24 resource agents.

agents' advertisements. Individually, any one broker will still require 24 seconds to process a query, but since there are more brokers to answer queries, the queries can be distributed among the brokers, allowing the system as a whole to handle a higher frequency of queries. The other multiple broker scheme has each broker specializing in certain advertisements which results from each resource agent advertising only to a single broker. In this arrangement, every broker must be involved in answering each query, since there is no single repository of all the resource agent advertisements. Answering queries in this case requires the brokers to communicate results with each other and give a combined response to the query.

For both the replicated and specialize multiple brokering architectures, reasonable response times can be attained with query frequencies as high as every 10 seconds; a time that is not achievable in the single broker arrangement, where the theoretical minimum is being able to handle a query every 24 seconds and the experimental results show that the query frequency cannot be below one every 30 seconds.

The advantage of the two multibrokering configurations over the single brokering setup is clear, but somewhat less obvious is the whether replication or specialization of brokers is the better solution. Figure 11 shows the comparison between the two multibrokering schemes for mean query intervals of 10 seconds and greater. Here the gains in computing the answers in parallel across multiple brokers outweighs the extra overhead involved with the inter-broker communication.

Figure 12 shows the same experiments as Figure 11 except that here there are only 4 brokers in the system, though still 24 resource agents. This shows that even with a higher resource-to-broker ratio, specialization of the brokers helps.

Scalability This set of simulation-based experiments varies the number of agents in the system, while maintaining all other system parameters. We simulated systems with the following numbers of resource agents: 12, 24, 48, 72, 96, 120, 144, 192, and 240. Since our focus is on the inter-agent communication

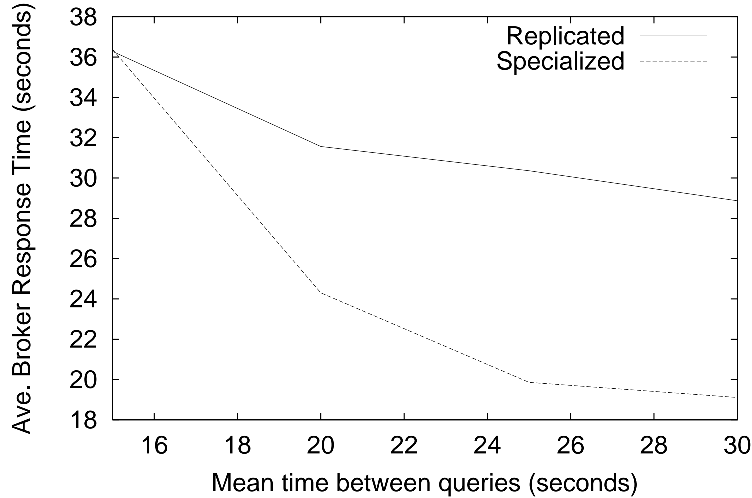

Figure 12: Replicated brokering versus specialized brokering with 4 brokers and 24 resource agents.

overhead, we needed to ensure that the broker agents' local computations remained the same across this range. Thus, we defined that each broker would, on average, have the advertisements for 12 resources. Thus the number of brokers for each of the above resource agent sizes are 1, 2, 4, 6, 8, 10, 12, 16, and 20 respectively.

Each resource agent's advertisement size was set to 1 megabyte and the processor speed and broker reasoning engine speeds were set to require one second of processing time for each megabyte of advertisements. Thus, on average, a broker will need 12 seconds to compute the query answer based on its local advertisements. This presents a theoretical lower bound on the response time in the multi-brokering system. Note that if you wanted to have multiple brokers each with identical copies of all advertisements, then the response times would definitely not scale well with the number of resource agents, since it will take each broker one second per resource agent to answer a query or a total of 240 seconds for the largest case we look at here.

The metric of interest here is the average response time to the query agent from the brokers. Unlike the InfoSleuth experiments which had some extra overhead for the processing and rendering of the result, this simulation data is purely the time between when the query is issued to the broker and when the reply is received from the broker.

Figure 13 shows the results of varying the number of agents in the system and for varying query frequencies ("QF" is the mean time between queries.) If the overhead of communication presented an obstacle to scalability, then one would expect the response times to get dramatically worse as the number of agents (both brokers and resources) increased. However, as the data in Figure 13 shows, the response times tend to level off, and certainly do not show any catastrophic behavior.

The results of this experiment shows that multi-brokering systems, despite the extra overhead, do scale up nicely. With a multi-brokering system, the gains achieved by distributing the query processing exceed

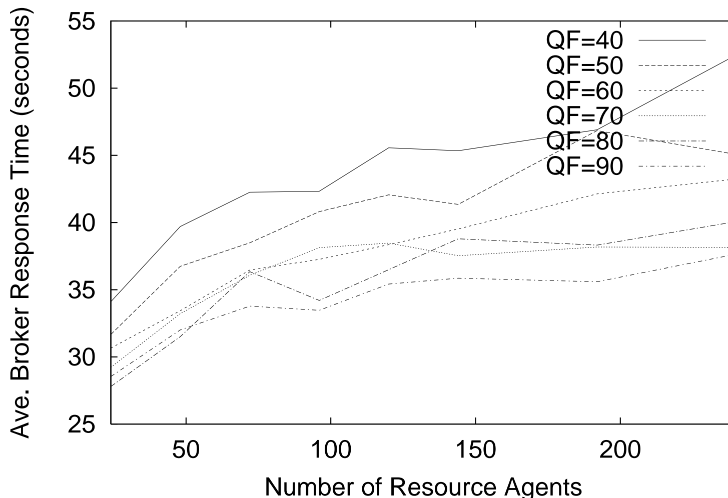

Figure 13: Scalability of broker specialization across a range of number of resource agents and system query frequencies (QF).

Failure Mean (secs.)	**Advertisement Redundancy (Number of Brokers)**				
	1	2	3	4	5
1000000	99.56%	97.37%	100.00%	99.14%	100.00%
3600	77.64%	70.71%	69.87%	61.26%	63.45%
1800	37.50%	44.40%	46.69%	44.64%	59.41%
900	34.05%	26.47%	17.87%	22.90%	16.79%

Table 9: Percentage of queries that brokers reply to.

the overhead incurred as the number of agents in the system increases.

Robustness In this set of experiments, we fixed the number of brokers and resources at 5 and 20 respectively. The query frequency was fixed to have a mean query time of once every 60 seconds to ensure that the system was operating in a range that did not saturate it processing capabilities. The parameters we vary are the mean failure time of the brokers and the amount of redundancy in the number of brokers that each resource agent sends their advertised to. The mean failure rates used are 1000000, 3600, 1800, and 900 seconds. We vary the number of brokers each agent advertises to from 1 to 5.

Table 9 shows the results for the number of replies from the broker to the number of queries asked of the broker (expressed as a percentage). Naturally, as the failure frequency goes up, the more likely we are to contact a broker that does not respond. Aside from the variation due to the random nature of the experiments, these percentages should be independent of the redundancy of the advertisements, since it only measures whether a broker replies not whether it actually located an agent to satisfy the query.

The robustness of the system is evaluated by looking at only those queries for which the broker responded. In these cases we want look at the quality of the broker's response. In this particular experiment

Failure Mean (secs.)	Advertisement Redundancy (Number of Brokers)				
	1	2	3	4	5
1000000	100.00%	100.00%	100.00%	100.00%	100.00%
3600	75.00%	92.90%	92.22%	97.42%	100.00%
1800	75.86%	85.44%	95.58%	100.00%	100.00%
900	20.25%	76.19%	69.05%	86.67%	100.00%

Table 10: Robustness experiments: percentage of queries successfully answered.

each resource has a unique data domain, so there is exactly one agent that should match each query. The quantity of interest is the number of resources queried versus the number of broker replies that are received, since this is directly correlated to the number of times the proper resource agent was located by the broker network. As seen in the first row of Table 10, when the brokers are very unlikely to fail, the number of resource agents queried will be identical to the number of broker replies (i.e., 100% of the queries answered had found the matching resource agent.)

The last column shows that with complete redundancy, you can always find the agent if you get a reply at all. In this case, the brokers have no real reason to communicate since all brokers know about all resources. However, for the other cases where inter-broker communication is needed to answer the queries, you can see the definite trend that the more redundancy there is, the more robust the system is to failures.

7 Related Work

There are several research projects and systems that address the issue of integrating heterogeneous systems. These systems rely on some form of brokering or mediation to achieve semantic integration. We evaluate two issues when comparing brokering in InfoSleuth to brokering in other systems – moving beyond syntactic brokering and using distributed multibrokering.

The CORBA Trading Object Service [18] implements multibrokering through a network of traders, and also provides mechanisms for incorporating some semantic brokering, however, they have not actually incorporated any reasoning. In fact the level of semantic brokering is equivalent to the look up of yellow pages in a telephone book. The knowledge that can be expressed in a yellow page is a structured list of properties. It is not possible to describe constraints such as range of data involved, relationships between input and output, subsumption relationships between concepts, and correctness and completeness of data. Additionally, the federation of traders is always formed statically with a fixed topology. Only recently have commercial implementations of CORBA traders been available (e.g., [8]). No information is available regarding the scalability and robustness of any of the CORBA traders. DISCO [22] uses CORBA brokers to do their brokering. Syntactic brokering functions are sometimes also incorporated into commercial agent frameworks such as Zeus [17].

Several information retrieval systems use semantic brokering with respect to information sources. These

systems include SIMS [1], TSIMMIS [15], InfoMaster [7] and Information Manifold [12, 13]. They all evolved from research in multidatabases where a canonical model (global ontology) is used. The way these systems work is to define a common vocabulary, or ontology, to define the objects in their information domain. Individual information sources that contain these objects then describe constraints on the objects that they can provide, in terms of this common vocabulary. The broker then uses these constraints to determine how to process queries from users that involve one or more of these resources. The capabilities of these systems are similar to InfoSleuth's in that they reason over the information content of the agents and constraints over that content; however, their idea of a service ontology only encompasses domain information and not any other agent capabilities. Thus, these systems implicitly do syntactic brokering when matching resources, as they expect that each resource supports a single underlying query language.

The SHADE project [14, 10, 11] at Lockheed Palo Alto Research Labs extended the notion of syntactic brokering by including information about services represented using KIF, a knowledge interchange method that represents first-order logic expressions. Queries concerning agents were matched with these advertisements using unification. Additionally, SHADE provides some facility to broker over constraints on the values of the data, similar to the semantic brokering over data that we have done in InfoSleuth. However, we have not found a clear description of how this facility works. Since SHADE relies on a shared ontology to represent data, they also propose a companion matchmaker named COINS that uses TF/IDF (term frequency / inverse document frequency) filtering to match document characteristics to free text [11].

The LARKS matchmaking system [21] in RETSINA [20, 4, 5], has attempted to address the issue of semantic brokering by providing input-output descriptors and term frequency measures to determine whether or not a semantic match occurs between a requested service and a service provider. RETSINA matchmakers describe the semantics of their offered services both in terms of signatures (inputs and outputs), but also in terms of the relationships between the inputs and the outputs. They also use TF/IDF techniques to categorize the semantic relevance of a query to an advertisement. The term frequency measures are similar to those used in COINS. The matching process in RESTINA is structured such that users can select a trade-off between performance verses quality of matching.

In addition, there are a few general papers of interest on brokering and agent architectures. These include a second developed agent framework similar to InfoSleuth and its approach to brokering (though different in other aspects) in [3] and some general discussions in information agents [11, 9].

8 Conclusions

The brokering function matches specific requests for services with providers that can satisfy those requests. Syntactic brokering does this match based on purely syntactic characteristics of the requested service such as method signatures, query languages or input forms. Semantic brokering takes into account the nature and characteristics of the requested service - what functions do you want to perform and what data do you want to access. A good brokering system, such as InfoSleuth's, should go beyond syntactic brokering.

We described our focused ontologies approach, where agents can specify different aspects of their functionality using shared, specialized knowledge. Advertising and querying is done in terms of ontology fragments expressed using XML syntax. This enables the broker to both access stored information gleaned from agent advertisements and reason over that information when determining which agents provide a set of requested services.

A single broker can accomplish much in the way of recommending resources and assisting in maintaining a dynamic set of distributed computing and information resources. However, a single broker architecture presents a barrier to scalability and robust operation. We presented a multibrokering peer-to-peer architecture, where brokers maintain up-to-date information about other brokers as well as other agents, couched in terms of a broker advertisements. We described how robust multibrokering can be implemented, especially in terms of how brokers discover other brokers and how to implement inter-broker searches. We showed empirically the feasibility of multibrokering, and explored its effectiveness. Our preliminary experiments were encouraging in that they showed the feasibility of and hinted at the scalability of the multibrokering system. The distributed multibrokering is functional in InfoSleuth and has been used successfully in EDEN project[6] and Technology Tracking project[16].

Because it was impractical to study multibrokering in very large agent-based systems, we made use of an agent simulator developed here at MCC to analyze the robustness and scalability of our multibrokering approach as the agent community grows in size. We analyzed the scaling behavior of our specialized brokering approach and showed its superior scalability with respect to replicated brokering systems. We experimented with various topologies and connectivity properties of the brokers, enabling us to determine in the future what the most efficient tradeoff is between connectivity and brokering speed and robustness. However, we do recognize the drawbacks of a simulation-based approach to such experimentation as well, including the need to validate the behavior of the simulator against the real-life system (whether implemented or not), and the accuracy of the settings of the myriad of simulation parameters such that the model faithfully represents the real-life components. Our confidence in our simulator and our configuration settings should continue to grow as we gain more simulation experience.

Acknowledgments

The authors would like to acknowledge the members and sponsors of the InfoSleuth group for their help in the brokering ideas presented in this paper, including Amy Unruh, Gale Martin, Ray Shea and Marek Rusinkiewicz for their helpful comments on the text. In addition, Damith Chandrasekara contributed significantly to both the multibroker design discussions and the implementation.

References

[1] Y. Arens, C.A. Knoblock, and W. Shen. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems*, 6(2), 1996.

[2] R. Bayardo and et.al. InfoSleuth: Agent-based semantic integration of information in open and dynamic environments. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, 1997.

[3] D. Moran D. L. Martin, H. Oohama and A. Cheyer. Information brokering in an agent architecture. In *Proc. Int'l Conference on the Practical Application of Intelligent Agents and Multi-agent Technology*, 1997.

[4] K. Decker and K.P. Sycara. Intelligent adaptive information agents. *Journal of Intelligent Information Systems*, 9(3), 1997.

[5] K. Decker, M. Williamson, and K. Sycara. Matchmaking and Brokering. In *Proc. Int'l Conference on Multi-Agent Systems*, 1996.

[6] Jerry Fowler, Marian Nodine, Brad Perry, and Bruce Bargmeyer. Agent-based semantic interoperability in infosleuth. *Sigmod Record*, 28(1):60–67, March 1999.

[7] M.R. Genesereth, A. Keller, and O.M. Duschka. Infomaster: An Information Integration System. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, 1997.

[8] IONA. White paper on orbix trader. Technical report, IONA Technologies, http://www.iona.com/, 1999.

[9] L. Kerschberg. The role of intelligent software agents in advanced information systems. In *Proc. British National Conference on Databases*, 1997.

[10] D. Kuokka and L. Harada. On using KQML for matchmaking. In *ICMAS*, pages 239–254, 1995.

[11] D. Kuokka and L. Harada. Integrating information via matchmaking. *Journal of Intelligent Information Systems*, 6(2), 1996.

[12] A. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems*, 5(2), 1995.

[13] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. Int'l Conference on Very Large Data Bases*, 1996.

[14] McGuire, Kuokka, Weber, Tenenbaum, Gruber, and Olsen. SHADE: Technology for knowledge-based collaborative engineering. *Journal of Concurrent Engineering: Research and Applications*, 1(3), 1993.

[15] Garcia Molina and et.al. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2), 1997.

[16] Marian Nodine, Jerry Fowler, Tomasz Ksiezyk, Brad Perry, Malcolm Taylor, and Amy Unruh. Active information gathering in InfoSleuth. *International Journal of Cooperative Information Systems*, 9(1/2):3–28, 2000.

[17] Hyacinth Nwana, Divine Ndumu, Lyndon Lee, and Jaron Collis. ZEUS: A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence Journal*, 13(1):129–186, 1999.

[18] OMG. OMG trading object service specification. Technical Report 97-12-02, Object Management Group, http://www.omg.org/corba, 1997.

[19] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[20] Katia Sycara, Matthias Klusch, Seth Widoff, and Jianguo Lu. Dynamic service matchmaking among agents in open information environments. *SIGMOD Record*, 1999.

[21] Katia Sycara, Jianguo Lu, Matthias Klusch, and Seth Widoff. Matchmaking among heterogeneous agents on the internet. In *Proceedings of the AAAI Spring Symposium on Intelligent Agents in Cyberspace*, 1999.

[22] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. In *Proc. Int'l Conference of Distributed Computing Systems*, 1996.