An Exploration in MAS Scalability

Content Areas: Multiagent Systems, Software Agents, Distributed AI, Autonomous Agents
Tracking Number: 794

Abstract

Naming and capability-mediated services, such as Agent Name Services and middle agents, perform a fundamental task in the Multi-Agent Systems (MAS) infrastructures, because they allow agents to find each other and interoperate. Because of their importance in the MAS, the scalability of naming and capability-mediation services is crucial to guarantee the overall scalability of the MAS. Surprisingly though, while MAS infrastructure has received a fair amount of attention, the quantitative evaluation of infrastructure components has been lacking. In this paper, we present an experimental evaluation of the agent name services and the capability-mediation used by the RETSINA infrastructure. In addition, we perform experimental comparisons of the same services offered by the CoABS GRID that utilizes Sun's $Jini^{TM}$. This evaluation is used as a framework to discuss evaluation metrics, design issues for agent naming and discovery services, and to provide suggestions for improvements of these services. [1].

1 Introduction

Open MAS , in which agents can appear, move to a different location and disappear more dynamically than typical information sources on the Internet, require services that abstract agents from their capabilities and their physical location. Such infrastructure services have been named Middle Agents and Agent Name Services (ANS). The ANS solves the location problem by associating the id of the agent (typically its name) to its physical location. An ANS acts like a Domain Name Service (DNS) but with increased flexibility for real-time updates, discovery services, and automatic "pushing" and "pulling" of agent registrations. Middle agents are the MAS counterpart of search engines on the Internet. They provide a mechanism to register which functionalities are provided by the agents in the MAS at any given time, and then other agents in the system find service providers by interrogating the middle agents [Decker, Sycara, & Williamson, 1997; Wong & Sycara, 1994]. For example, middle agents can identify the agent services currently in the system that provide information on the stock market. Examples of middle agents include the OAA Facilitator [Martin, Cheyer, & Moran, 1999], the RETSINA Matchmaker [Sycara *et al.*, 1999] and the Infosleuth Broker [Perry, Taylor, & Unruh, 1999]. Middle agents maintain an up to date registry of agents that have made themselves known to the MAS community, along with the services that each agent provides.

In this schema, we can distinguish 3 types of agents: agent providers, agent requesters and middle agents. The process of capability-based mediation has the following steps: (1) providers advertise their capabilities with middle agents, (2) middle agents store these advertisements, (3) a requester asks a middle agent to locate providers with desired capabilities, (4) the middle agent processes the request against its knowledge base of advertisements, and finally (5) depending on the type of middle agent, e.g. matchmaker or Facilitator the result could be either the stored advertisements of the found agents (if the middle agent is a Matchmaker) or the result of the complete transaction (when the middle agent is a Facilitator). Note that agent providers and requesters are not mutually exclusive, since agents often act both as providers and requesters. This is the case when an agent requires the service of another agent while processing a request itself.

Naming and capability-mediation infrastructure services are indispensable for finding agents and they constitute a fundamental functionality of MASs . The scalability of naming and capability-mediation services is therefore crucial for the overall scalability of MASs . The scalability of these services can be evaluated along two dimensions: (1) the number of requests they can process and (2) the number of concurrent agents they can handle. In this paper, we concentrate on the first dimension, and therefore measure the scalability of the RETSINA MAS infrastructure as the time required by the ANS and Matchmaker to register and lookup agents under heavy load conditions. We will use the Grid as benchmark, because the Grid is a MAS implemented in $Jini^{TM}$ that takes advantage of the $Jini^{TM}$ Lookup service (LUS) [Edwards, 1999] which is a scalable industrial strength lookup service. Furthermore, it is possible to compare the ANS and the Matchmaker with the $Jini^{TM}$ LUS used by the Grid

[1]This material is based on work supported in part by MURI contract N00014-96-1222 and CoABS DARPA contract F30602-98-2-0138 and NSF grants IRI-9612131 and IRI-9712607

since they follow similar registration and lookup protocols.

The rest of the paper is organized as follows: Section 2 describes, in more detail, the capability and location problem and the tradeoffs between RETSINA and the Grid ; section 3 describes the experimental results of different configurations of the ANS ; section 4 shows the empirical results of comparing the Grid and RETSINA, and finally we conclude in section 5.

2 The RETSINA Infrastructure

RETSINA [5,6] is a distributed multi-agent infrastructure for open environments, such as the Internet. It allows heterogeneous agents to find each other by name, or by capability, and to interact with each other. RETSINA has been used to develop a variety of applications [16, 17]. In contrast with other MAS architectures such as the Grid and OAA [Grid, ; Martin, Cheyer, & Moran, 1999] that cluster the functionality of the ANS and Capability-Based services in the same lookup services ($Jini^{TM}$ LUS for the Grid , and the Facilitator for OAA), RETSINA explicitly distinguishes between the two services: an Agent Name Service and middle agents. The Agent Name Services (ANS) allows agent registration by name and provides resolution of an agent name to a physical location (for example: machine host and port). A set of middle agents (e.g. matchmakers, brokers etc) provide capability-based mediation services, namely finding agents that can provide desired services. Decoupling the two services is justified by the fact that agent discovery by name is a system-level service, whereas capability-based lookup is a service at the knowledge level. This is demonstrated by fact that agents' capabilities and locations can change independently: the agents can move to a different location without loosing or gaining any capability, or acquire new functionalities without changing location.

The explicit distinction between ANS and Middle Agents has important consequences: first, RETSINA could seamlessly use other naming services, such as the DNS, especially when the new proposals for enhanced DNS services get implemented. This is very convenient to integrate services that support different protocols such as wireless services. Second, some agents may not want to advertise their capabilities but only request services. Interface agents that act exclusively as clients in lieu of a human user are an example of agents that may not want to register with a Middle Agents. Through the Interface agent, the user accesses the services provided by the agents in the MAS, but the user does not offer any service to those agents. Finally, capability descriptions, advertisements and requests, have sophisticated representations, possibly containing logical and ontological formalisms, and hence are much more computationally expensive to store and match. RETSINA allows localization of expensive services that the agents may access sparingly and cheaper services that agents may access more easily. If an agent knows another's name and capability already, it is very inefficient to force this agent to do a lookup in a capability-mediation service (such as $Jini^{TM}$ LUS or Matchmaker) rather than using a much more efficient agent naming service.

3 RETSINA ANS

As described above, the role of the ANS is to resolve the name of the agent into its physical address: this is the most basic functionality that allows agents to inter-operate. Since the ANS plays such an important role, it should be very reliable and scalable. In this section we will discuss the basic functionalities of the ANS and then we will measure their scalability.

Upon launching, an ANS broadcasts its presence in the system on a multicast channel, which notifies all agents and ANSs in the LAN of the new ANS in the system. Agents autonomously decide whether to use it or not, while other ANSs contact it to exchange registrations and requests. The result is that the ANSs in the LAN form a *Discovery Group* (DG): a group of ANSs aware of each other and able to exchange information. Similarly, when agents enter the system, they multicast a request for an available ANS to the local DG . Every ANS in the LAN will answer the call and the agent decides with which one to register[2].

Reliability and fault tolerance is achieved in the ANS by running multiple ANS servers that store redundant information. Consequently, even if one or more ANS is not available, their information is not lost, but can be retrieved by querying other ANSs that are still running. Redundancy is achieved in two ways: *Client Push* and *Server Push*. In the Client Push case, agents, that are the clients of the ANS , push their registrations to many or all ANSs they find; Server Push is performed directly by the ANS servers that push information on their registrations to other ANSs in their DG .

An agent can query one ANS or multiple ANSs at the same time. We call this option *Client Pull*. In this case, the agent queries all viable ANSs that it is aware of for a result. Alternatively, the agent can query only one ANS and let this ANS query other servers to find the address the agent is seeking. We call this mechanism *Server Pull*: if an ANS cannot find the address of the agent in its own data base, it forwards the query to other ANSs in its DG .

The mechanisms described so far are based on the ANSs that can be discovered in the LAN through multicast. If we want MASs to work across the Internet, agents should not be restricted to the local area net, but they should be able to communicate with agents anywhere on the net. For this reason, each ANS also has access to a *Hierarchy list*: a list of references to other ANSs that are outside the LAN. If an ANS cannot resolve a reference to an agent within it own LAN, it queries the ANS , in its Hierarchy list, which in turn will perform similar searches. The advantage of the Hierarchy list is to provide a way to find references of agents across the Internet, without the burden of registering each agent with every ANS in the net.

3.1 Scalability of the ANS

In the previous section we showed that the agent can be configured to delegate to the ANS , to find the information it is seeking using a Server Push/Pull, or it can keep control of the

[2]Agents may also have an explicit reference to an ANS , in such a case they have no need to multicast any request, rather they register directly with the referenced ANS .

process using a Client Push/Pull. In addition, the agent can strike a balance between the two options by using the Server Push/Pull on some ANS and Client Push/Pull on others.

The decision of which modality, of interaction with the ANS , to use depends on the priorities of the agent; but usually time is one of the most important factors on the decisions that have to be made. For this reason our experiments test the ANS under the heaviest load conditions possible. The experiments below were set up to estimate the time penalties the agent would incur by selecting one option versus the other. In the first set of experiments, we compared Server Push and Client Push. In both experiments we use a set of ANSs (1 to 5[3]) and a Test program that registers agents on the ANS . Both trials were performed within the University public network with a bandwidth of 10Mbps using 6 publicly available machines (Sun Ultra), under average usage load. All the ANSs and the Test programs were written in Java and compiled in the version 1.2, while the communication between the agents was based on TCP/IP sockets.

The Test program registered 100,000 agents with these ANSs . In the Server Push case, the Test program registers 100,000 agents with the same ANS which then takes the responsibility of distributing these registrations to the other ANSs . In the Client Push case, the Test program registers the 100,000 agents with each ANS, and no Server Push is performed. Registrations are performed sequentially as soon as the Test program receives the acknowledgment of the success of the previous registration. Each registration action included the creation and dissolution of the TCP socket between the test client program and the ANS server(s). The results of the experiments are shown in table 1. The times reported were measured by the Test program as the time passed between the sending of the message and the reception of an acknowledgment from the ANS.

Number of Servers	Server Push	Client Push
1	5	5
2	6	21
3	8	35
4	10	43
5	14	60

Table 1: Average time (in milliseconds) per Agent Registration in the Server and Client Push conditions.

The results show that despite the high number of registrations, the ANS performs its tasks extremely fast: just few tens of milliseconds, with the Server Push condition being faster than Client Push. This data shows that the ANS can indeed easily handle a great number of registrations. The Client Push case results are the slowest because in this case the Test program has to register 100,000 agents with all ANSs, while in the Server Push condition the ANS reply immediately, and then it pushes the registration with to the other servers taking advantage of the intrinsic parallelism of MASs.

[3]Due to the intense network traffic generated by the experiments, we were not able to run more than 5 servers without heavy loss of transmission packages.

In the second experiment we compared Server Pull vs Client Pull. Here, the Test program sent 5000 requests to each ANS . None of these requests could be resolved with the registrations stored in any of the ANSs, as a consequence all the ANSs were queried. In the Server Pull condition the agent sent a request to one single ANS which in turn forwarded it to the other ANSs . In the Client Pull condition, the Test program waited for a reply from the ANS before testing the next one. This experiment was performed with an experimental setup similar to the previous one and the results are shown in table 2.

Number of Servers	Server Pull	Client Pull
1	0	1
2	4	9
3	6	15
4	12	24
5	19	34

Table 2: Average time (in milliseconds) per Agent Lookup in the Server Pull and Client Pull conditions

The results show that the average time of a lookup is extremely low: just a few teens of milliseconds despite the high load of registrations and the frequency of requests. This result combined, with the results in table 1, shows that the ANS is indeed a very scalable service that it can support a very high number of agents in the system.

The experiments so far showed the performance of multiple ANS servers that were queried sequentially, but very rapidly, by the same client. An alternative question is what would be the behavior of the ANS when queried by multiple clients asynchronously. In the next set of experiments an increasing the number of clients from 1 to 5 register 100,000 agents with a single ANS . Furthermore, in a parallel set of experiments we measured the average time of lookup the ANS under the same set of conditions. The results of these experiments are shown in table 3.

Number of Clients	Push	Pull
1	5	4
2	6	6
3	8	7
4	13	10
5	16	13

Table 3: Average time (in milliseconds) per registration and lookup of one ANS queried by 1 to 5 clients under a load of 100,000 registrations.

Although not conclusive, these experiments suggest that the ANS is not heavily affected by the asynchronous messages: the data shows that the response time of the ANS increases of just about 2msec per each client added. This experiment reinforces the conclusion above: the ANS can indeed support a high number of agents and it can reply to them in a matter of few milliseconds despite the load and the asynchronous messages. Furthermore, the experiments show that the MAS is more likely to hit the limits of the network that the agents are using as communication channel, rather than the computational limits of the ANS, since when more than 5 servers are used, the number of conflicts and loss of packages drastically increased.

4 RETSINA Matchmaker

As described above, the ANS constitutes only part of the whole infrastructure: Middle Agents constitute the other major component. The role of Middle Agents is to map the id of the agent into a description of its capabilities. While middle agents come in many sorts [Wong & Sycara, 1994], here we concentrate on a specific type of Middle Agent, called Matchmaker, that closely matches the $Jini^{TM}$ LUS used by the Grid.

The Matchmaker is found by agents in the system through discovery following the same mechanism described above with the ANS. Upon finding a Matchmaker, agents advertise their capabilities; or look for agents that advertised services that match a specific request. The task of a Matchmaker is to collect and store advertisements it receives and to match them against the incoming requests. To accomplish this task a RETSINA Matchmaker uses a matching engine that performs both syntactic and semantic analysis of the advertisements to find exact or partial matches.

An advertisement in RETSINA consists of the “context”, ie a description of the domain of a service/agent (e.g. selling shoes), the input and output parameters of the agent (so that queries to it can be formulated), input and output variable types and input and output constraints. The matching algorithm can also avail itself of ontological services, either in the form of taxonomic inferences using the WordNet ontology[Fellbaum, 1998], or in the form of logical inferences using a terminological language ITL [Sycara *et al.*, 1999]. Using an ontology, allows the matching algorithm to (a) make partial matches based on similarity of input/output variables and constraints, and (b) make more sophisticated matches than “string only” matches. For example, the matching algorithm is able to detect that a request for an agent that provides information on “dogs”, can be satisfied by a registered advertisement of an agent that provides information on “animals”. The matchmaker also implements a monitoring and notification service, so that agents are notified of the existence of a new service they are interested in as soon as this service advertises its availability.

The RETSINA Matchmaker provides a requester agent with the contact information of relevant providers to allow them to directly interact with a service. This crucial difference with other middle agents such as Brokers and Facilitators makes the RETSINA Matchmakers less of a single point of failure, since after a requester has been given a list of providers, it can continue its transactions directly, even when no Matchmaker is present. In addition, a requester can cache providers’ contact information and reuse them without resorting to the matchmaking process every time. Finally, the Matchmaker’s get-out-of-the-way behavior removes a bottleneck from the system, increasing the overall efficiency of the MAS and contributing to the security of the transactions since there is no single node crossed by all messages exchanged by the agents in the MAS.

4.1 The Grid benchmark

In order to evaluate the scalability of the Matchmaker, we compared its performance with the Lookup Service of the CoABS Grid that is based on the $Jini^{TM}$ LUS . The Grid is an open MAS infrastructure that supports interoperation of agents and $Jini^{TM}$ services[4], based on the $Jini^{TM}$ infrastructure: specifically it uses the $Jini^{TM}$ LUS as its core component to find agents in the MAS.

In $Jini^{TM}$, service providers enter the system by registering with the LUS . This registration[5] consists of two components: a Service Object and Service Attributes. The Service Object is a proxy of the provider it represents, while the Service Attributes describe features of that provider. Upon lookup, the Service Object is sent to the requester and it acts as an interface to connect the provider and the requester, in the sense that all the functionalities that the provider advertised will be available to the requester through the Service Object. The Grid uses a set of standardized libraries built on top of $Jini^{TM}$ to access the LUS, and to describe the agents in the system.

$Jini^{TM}$ LUS and the Matchmaker have similar functions in the system: they provide the point of contact of agents in the MAS, but neither the Matchmaker nor the $Jini^{TM}$ LUS manage the interaction between the requesting agent and the service provider. Furthermore, both the $Jini^{TM}$ LUS and the Matchmaker concentrate on the identification of the agents on the bases of what they provide rather than of their location as is the case of the ANS. The agent location function performed by the ANS in RETSINA is instead performed by Service Objects in $Jini^{TM}$, since these objects are responsible to contact the agent.

While there are some similarities between $Jini^{TM}$ LUS and the RETSINA Matchmaker, there are also striking differences between them. RETSINA’s advertisements describe the functionality of the agents: what inputs they require and what outputs do they produce. Furthermore, as seen above, RETSINA uses a flexible matching that can recognize the similarity between the request and the advertisement even though there is no perfect syntactic much between the two. $Jini^{TM}$ on the other side checks only whether the type of the object requested matches with any of the types of the objects in the LUS, and whether the values of the attributes of the request matches the attributes of the provider. Furthermore, since $Jini^{TM}$ compares objects rather than descriptions of functionalities, the attributes list and type of the object is unconstrained, which forces the requesting agent should know the service it is looking for, in advance of the search. This is not the case with RETSINA, since any agent in the RETSINA system needs to know only what outputs it expects from a service provider and possibly what information it is willing to pass as inputs; the matchmaker then assumes the task of selecting the best matches for that request.

[4]In the remaining sections of the paper we will not distinguish between $Jini^{TM}$ services and agents since the Grid does not make any distinction either.

[5]The registration operation is called ”join” in $Jini^{TM}$ since it is performed when a service joins the system.

4.2 The Scalability of the Matchmaker

We measured the scalability of the Matchmaker along two dimensions: registration time and lookup time. The tests were performed by installing one agent that sequentially transmitted advertisements to a Matchmaker running on another machine. The data was collected by the agent as the total time required to send the advertisement and receive an acknowledgment, or the total time required to send a request and receive an answer. Similarly, we ran a Grid agent that advertised or queried the $Jini^{TM}$ LUS using the same settings used in the Matchmaker experiments. In all conditions, the agents and the Matchmaker or $Jini^{TM}$ LUS ran on different 600 MHz pentium machines running Linux and Java 1.3. The same machine was used to run the Matchmaker and $Jini^{TM}$ LUS . All experiments were performed within the University public network, with normal conditions of traffic.

Agent Registration

In the registration experiments (equivalent to the "push" experiments for the ANS), the agents advertised themselves with the Matchmaker or the $Jini^{TM}$ LUS. The results of the experiments are displayed in table 4, for comparison and ease of evaluation, the results of the experiments with the ANS are added.

Agents Registered	Matchmaker	Grid	ANS
100	64	219	4
200	62	208	4
500	81	NA	3
100,000			5

Table 4: Registration time with the Matchmaker, the Grid and the ANS

The experiments show that the Matchmaker takes less than 100 ms to register an agent even with a load of 500 agents. This result is very encouraging because it shows that the Matchmaker can handle the registration load despite the size increase of the MAS. The combination of these results with the ANS results, shows that the RETSINA MAS infrastructure can indeed handle a fair number of agents. The results show also that a registration with the Grid takes about three times longer than with the Matchmaker. This is due to the fact that while the Matchmaker indexes the advertisement and stores it in its internal DB, while the $Jini^{TM}$ LUS needs to reconstruct the serialized object that it receives which apparently is a costly operation. Furthermore, the Grid failed to register more the 205 agents without issuing an out of memory error. It is impossible for us to identify whether the problem is due to a limitation in the $Jini^{TM}$ LUS or on the size of the objects used by the Grid to represent agents.

Agent Lookup

To analyze the scalability of the Matchmaker lookup, we again compared the lookup time of RETSINA agents with the Matchmaker and the lookup time of Grid agents with the $Jini^{TM}$ LUS . Since $Jini^{TM}$ does not perform any ontological match, the Matching engine used by the Matchmaker did not perform any ontological match either. Furthermore, we measured how the performance of the Matchmaker lookup degrades when the number of querying agents increases. We performed this experiment running 1, 2, 3, 4 and 5 clients, each of which was sending randomly generated lookup requests. As in the previous case, we ran the Grid in the same conditions. The results of the experiments are displayed in table 5.

Agents	Grid 100	MM 100	Grid 200	MM 200	MM 500	ANS 100000
1	330	82	353	93	128	4
2	332	161	470	155	221	6
3	373	190	540	235	332	7
4	407	247	636	305	440	10
5	441	313	694	379	563	13

Table 5: Average Lookup Time, in Milliseconds, for the Grid, the Matchmaker and the ANS.

The results of this last experiment again suggest that the Matchmaker scales as the number of lookup requests increases. Furthermore, it shows that the lookup time with the Matchmaker greatly dominates the lookup time with the ANS, which is not a surprising result since the task performed by the Matchmaker is more complicated than the task performed by the ANS. The correct management of the Matchmaker is therefore more critical than the management of the ANS. Furthermore, the Matchmaker matching time increases linearly with the number of clients, because the Matchmaker handles one request at a time (any attempt to multithread the matchmaker showed worse performances because of the overhead of managing the extra threads). Nevertheless, RETSINA scales better than the Grid. As the data shows, the Matchmaker answers, on average, about twice as fast as the $Jini^{TM}$ LUS with the Grid registrations. The Matchmaker is faster than the $Jini^{TM}$ LUS even with the greater load as the data at 500 agents shows. The Matchmaker has two advantages on the $Jini^{TM}$ LUS: first, advertisements are strings of a few hundred characters, while $Jini^{TM}$ has to instantiate the Service Objects that it receives from the providing agents; the second advantage is that the Matchmaker's engine efficiently accesses the slots of each advertisement and requests, while the $Jini^{TM}$ LUS has been constructed to index Service Objects by their type while the matching on the bases of the attributes is very inefficient[6], but this seems to be exactly the type of access provided by the Grid.

5 Discussion and Further Research

This paper discussed the scalability of the main components of the RETSINA MAS infrastructure: namely the ANS and the Matchmaker. Specifically, we were interested in evaluating how well that these components scale under heavy registration load. The results show that both the ANS and the Matchmaker can support a high number of registrations. Specifically, the ANS can perform a registration or a lookup in a few milliseconds; even when loaded with up to 100,000 registered agents and receiving many concurrent requests. In

[6]We thank Jim Waldo for this explanation

this case, the MAS infrastructure is limited more by the scalability of the communication network rather than by the computational load on the ANS. The Matchmaker also scales very well, taking on average 128 ms per lookup, with a load of 500 agents registered. Furthermore, it scales well when compared with the Grid, even with 5 agents querying concurrently at high speed.

Additional tests will be needed to evaluate the scalability of the ANS and Matchmaker when queried by a high number of concurrent agents, rather than a few sequential agents as in the experiments discussed here. Furthermore, the tests with the Matchmaker did not include any ontological matching. Ontological inference is usually computationally heavy, and it is bound to dominate the Matchmaker's lookup time. Therefore we envision a distributed Matchmaking system where each Matchmaker is specialized and has efficient access to one specific ontology as described in [Jha *et al.*, 1998], rather than distribution based on the topology of the local network as in the case of the ANS and $Jini^{TM}$ LUS. Empirical results will show which configuration will scale better. Finally, the scalability of the ANS will be improved by replacing the static Hierarchy configuration described in this paper with a self-configuring one. In addition mechanisms for security, leasing and integrity checks will be added and their effect on the registration and lookup time will be measured.

6 Conclusion

The experiments described in this paper show that the RETSINA MAS infrastructure is scalable well beyond the current size of MASs. Although the ANS and Matchmaker were queried sequentially, the queries were sent at a frequency that is bound to be much higher than the frequency of queries and lookups in a typical application under normal operating conditions. Furthermore, it compared well against the Grid which is based on the $Jini^{TM}$ LUS. These experiments therefore show that the RETSINA infrastructure can reliably support large concurrent MAS applications.

References

[Decker, Sycara, & Williamson, 1997] Decker, K.; Sycara, K.; and Williamson, M. 1997. Middle-agents for the internet. In *Proceedings of IJCAI97*.

[Edwards, 1999] Edwards, K. 1999. *Core Jini*. Prentice-Hall.

[Fellbaum, 1998] Fellbaum, C. 1998. *WordNet: An Electronic Lexical Database*. MIT Press.

[Grid,] Grid. http://www.globalinfotek.com/.

[Jha *et al.*, 1998] Jha, S.; Chalasani, P.; Shehory, O.; and Sycara, K. 1998. A formal treatment of distributed matchmaking. In *Agents 1998*.

[Martin, Cheyer, & Moran, 1999] Martin, D.; Cheyer, A.; and Moran, D. 1999. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence* 13(1-2):92–128.

[Perry, Taylor, & Unruh, 1999] Perry, B.; Taylor, M.; and Unruh, A. 1999. Information aggregation and agent interaction patterns in infosleuth. In *Proceedings of CIA99*. ACM Press.

[Sycara *et al.*, 1999] Sycara, K.; Klusch, M.; Widoff, S.; and Jianguo, L. 1999. Dynamic service matchmaking among agents in open information environments. *ACM SIGMOD Record* 28(1):47–53.

[Wong & Sycara, 1994] Wong, H.-C., and Sycara, K. 1994. A taxonomy of middle-agents for the internet. In *Proceedings of ICMAS2000*.