Matchmaking and Brokering

Keith Decker, Mike Williamson, and Katia Sycara
The Robotics Institute, Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
(decker,mikew,sycara)@cs.cmu.edu

May 16, 1996

Abstract

In this paper we define the notions of agent *matchmaking* and *brokering* behaviors that are used while processing requests among initially unacquainted sets of agents. These behaviors are basic components of common organizational roles and thus must be understood by computational agents. Brokering and matchmaking behaviors can be used to construct organizational forms such as centralized or distributed markets, ad hoc teams, and bureaucratic functional or product hierarchies. Each behavior brings with it certain performance characteristics—cost, robustness, and adaptiveness qualities—that are related to characteristics of the external environment and of the agents themselves. For example, while brokered systems are more vulnerable to certain failures, they are also able to cope more quickly with a rapidly fluctuating agent workforce. We present several agent design constraints and related models, which are experimentally validated using the WARREN multi-agent portfolio management system.

Topic Areas: Organization and Social Structures, Communication Issues and Protocols
Word Count: 4993

1 Introduction

One of the basic problems facing designers of open, multi-agent systems for the Internet is the connection problem [4]—finding the other agents who might have the information or other capabilities that you need. The fact that the system is open (participants may enter and exit at any time) and distributed over the entire Internet precludes broadcast communication solutions.

The solutions that have been proposed rely instead on well-known agents and some basic interactions with them: matchmaking and brokering [8, 10]. Standard agent communication languages (i.e., KQML [7]) even define specific 'performatives' (RECRUIT, BROKER, FORWARD) for these behaviors. These behaviors are also common in human open systems as well.

Since these behaviors are both common and appear to be useful for computational multi-agent systems, it is important to carefully define these behaviors and understand their effects with respect to environmental and individual agent characteristics. Sections 2 through 4 will define these behaviors; rather than define them in terms of new classes of communicative acts, we will define them in terms of simpler request actions. This has two benefits: first, the semantics of requests are well-understood [11, 2], and second, such a definition allows us to build simpler agents that can work in a open environment with hybrid behaviors (both matchmaking and brokering). By defining these behaviors and how they interact, we are able to better understand what constraints there are on agent architecture and behavior design.

The decision to use matchmaking or brokering to solve the connection problem offers performance tradeoffs along a number of dimensions, both quantitative (such as the time needed to fulfill a request) and qualitative (such as the robustness and adaptivity of the system to the failure or addition of agents). Our ongoing research aims to develop empirically-validated models of all the relationships between the various performance attributes and system parameters. In this paper, though, we will focus our attention on one particular issue: what is the comparative performance of a brokered vs. a matchmade system?

The questions we will be examining include first the quantitative end-to-end response time advantages and disadvantages of matchmaking and brokering behavior. Secondly, we will examine characteristics of these behaviors with respect to robust and adaptive open systems, where agents might enter and exit the system at any time. Experimental results are reported in Section 5, using our implementation of the WARREN multi-agent portfolio management system.

Another problem faced by open multi-agent system designers is the ontological mismatch problem—we will not address this problem in this paper (c.f. [3]). We

will assume that useful, shared domain ontologies exist and are being used by enough agents to provide basic domain services. Another assumption that we will make is that agents are sincere in their communications to one another [2].

2 Approach

We have chosen to define matchmaking and brokering behaviors so that they use requesting speech acts (in KQML, ASK, ASK-ALL, STREAM-ALL, etc.) and avoid using the additional special KQML performatives RECRUIT, BROKER, and RECOMMEND for several reasons:

1. Requests are fairly well defined and understood. Although no complete formal semantics for KQML exists yet, several definitions exist for simple requests [11, 2]. This previous work allows us to understand more clearly what is going on with respect to matchmaking and brokering

2. Requests are very basic communication actions, and our agents necessarily have considerable knowledge about how to deal with them: how to construct simple or periodic queries, how to monitor for future events of interest, how to process various replies, and how to deal with problems that arise when the communication is sent or if the remote agent later has a problem. Making requests is such a basic activity that it is hard to imagine an agent without this behavior. By using requests as a foundation for matchmaking and brokering behaviors, we can capitalize on this part of an agent's knowledge.

3. By eliminating the communicative difference between answering and brokering a query, the resultant system of agents gains in openness—the ability to add, subtract, and reorganize participants—without adding communicative complexity (new performatives and associated behaviors for every participant).

3 Matchmaking

In general, the process of matchmaking allows one agent with some objective to learn the name[1] of another agent that could take on that objective. This process involves three different agent roles (Figure 1):

[1] We will assume that knowing a fully qualified name is enough to enable direct communication.

Requester: an agent with an objective that it wants to be achieved by some other agent.

Matchmaker: an agent that knows the names of many agents and their corresponding capabilities.

Server: an agent that has committed itself to fulfilling objectives on behalf of other agents.

A single agent may (and often does) take on more than one of these roles. Because the successful fulfillment of the matchmaker role requires collecting and maintaining special knowledge, real systems typically allow one or a small number of agents to specialize in the matchmaker role.

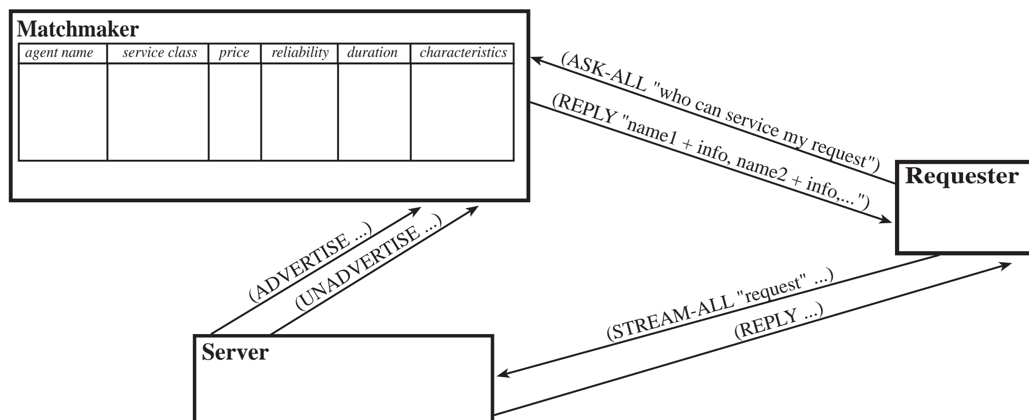


Figure 1: Communications between three agents taking on the roles of matchmaker, requester, and server.

3.1 Requester Role Behaviors

We will discuss the behaviors involved in the requester role in some detail, because they are basic to the construction of larger multi-agent systems as a whole, and are the *same* for each of matchmaker, broker, and hybrid matchmaker/broker systems.

An *objective* is a generalization of a simple goal. While a goal is either achieved or not, an objective may be partially achieved to some level of “quality” (in a worth-oriented environment [13]). An agent R taking the role of a requester has some objective that either:

1. R believes it cannot achieve on its own, or

2. R believes might be better achieved elsewhere to maximize its own or the system's utility (the objective can be achieved with higher quality elsewhere, or at a lower cost).

In particular, we do not require at this point (before communicating with the matchmaker) that the requester has made an internal commitment to the objective (in the Cohen & Levesque sense[1, 2]). This is a fairly important assumption, as it simplifies the reasoning that can be expected at the matchmaker—since agents may ask questions about services without any formal intention of using those services. For example, asking who might be able to repaint my car does not give you license to reason that, in fact, I am committed to actually having my car repainted. This assumption does *not* hold when a requester communicates with either a server or a broker.

The behavior of a requester with respect to accomplishing some objective remotely is thus, in general:

1. Find the names of agents who are able and willing to take on the objective. In addition, discover what is known about the utility of using each such agent (e.g., will the quality of the result vary? how long will it take? is there a price involved?). This behavior is accomplished by contacting an agent that can answer such a query—one that has taken on the role of a matchmaker.

2. If you decide to commit, chose a server and send the initial request.

3. Form an expectation with respect to the future arrival of communications from the server (results, status reports, or even errors or other failure notifications).

The requester agent builds an appropriate KQML request (e.g. ASK-ALL) to query the matchmaker about who might be able to accomplish the agent's objective, and at what cost (money, time, solution quality, etc.).

A requester agent can also locally cache matchmaking knowledge. Cached knowledge can make the system more robust in case of a matchmaker failure. Locally cached matchmaking information can also reduce the load on a matchmaker. The downside is that the cache may be outdated. In our implementation, a matchmaker agent is a type of *information agent*. Any information agent can handle queries stated so that the requester is kept informed if the results of this query change (i.e. new agents enter the system, agents leave the system, etc.).[2]

[2]Kuokka [10] cites several other justifications for this useful behavior.

3.1.1 Querying the Matchmaker

The point of a matchmade system is to keep the decision-making responsibility at the requester (compare this to the brokered system, were the broker takes on much of this responsibility). Thus the matchmaker should *not* limit the potential names returned in any way unrelated to the query itself.

To construct the matchmaker query, we are developing an agent service ontology [5]. Agents in a server role advertise their services using this ontology (committing to certain future classes of action under specified conditions) and requesters can query such advertising assertions. The advertisement serves as a model of another agent's general capabilities. We will discuss advertisements in more detail in the section on server role behaviors; for now it is sufficient to comment that a requester can query for general service characteristics (such as duration or price constraints) as well as for service-specific characteristics.

3.1.2 Robust Requests

In an open system, agents may come and go at any time. Agents playing a requesting role need to orient their behaviors to deal with such a dynamic environment. A basic robustness behavior is to detect and recover from matchmaker failure (when there is no local cache). Either the current plan is aborted, or the requester waits and tries to contact the matchmaker again. This behavior depends on the assumption that matchmaker services are themselves robust, and that if a matchmaker goes down, it will be back as soon as the underlying hardware and software can recover.

If the attempt to find an agent to service an objective succeeds, the agent may decide to make an internal commitment to the objective, and make a request to that remote agent to service the objective. At this point several things can still go wrong. First, even though each agent acts to the best of its beliefs about the true state of the world, the beliefs may be incorrect. For example, the remote agent may not exist or no longer be servicing the stated objectives. The remote agent may even accept the objective, but later come to believe that it cannot be achieved. As with human agents, even the most rational, considerate, and well-behaved agents may simply die without warning. In the first two cases, the requester is notified of the failure. The default requester behavior (remember, the requester at this point is internally committed to the objective) is then to retry to achieve the objective via other means (typically, other agents that may be able to service the commitment). Only after all such attempts are exhausted might the agent have to abandon the commitment as unachievable. Currently, we assume that all agents in the system are semi-permanent, and thus will

return eventually. Under this assumption, it makes sense to retry a previously failed servicing agent.

The final error, one of untimely agent death without warning, is potentially the most problematic. For structured objectives, it may be possible to monitor the behavior of the servicing agent. Currently, no special KQML performatives exist for this type of request, so we would suggest using the existing KQML request performatives and defining a meta-objective ontology to delineate questions that can be asked about ongoing processes: "When do you think you will send me the next message in reply to objective request X?" Of course, when an agent is monitoring for a condition, in many cases there is no expectation of when that condition will actually become true. Since accidents do happen, there are two possible solutions. One solution is to subsume agent death as one way an objective may become "unachievable"—but unlike the other cases this state cannot be recognized by the dead agent. The requester is left to periodically murmur "Are you still there (and thus still committed to the objective)?". The second solution is to require that the agent architecture support the persistent storage of remote commitments (so that they will survive the agent) and the automatic restart of failed agents. The re-animated agent then carries on with the commitments it had in its previous life.

Finally, if the requester agent wishes to withdraw from the organization and has active outstanding requests at server agents, it must inform those server agents of the fact that it is abandoning those commitments.

3.2 Server Role Behaviors

Agents taking on a server role in a matchmade system also need a set of standard behaviors. In particular, any agent that will take on a server role must clearly declare this intention by making a long term commitment to taking on a well-defined class of future requests. This declaration is called an *advertisement*, and it is communicated to the matchmaker. The advertisement contains both a specification of the agent's capabilities with respect to the type of requests it can accept, and both general and service-specific constraints on those future requests (see [5]). It is possible in an advertisement to express that some request constraints cannot be determined in advance of a specific request. For example, the price of a service may not be fixed beforehand, but need to be quoted (and perhaps negotiated) on a case-by-case basis. In this paper, we will only concern ourselves with services whose characteristics are fixed beforehand.

Thus, when an agent takes on a server role it creates an advertisement and sends it (using the KQML ADVERTISE performative) to the well-known matchmaker. If

the matchmaker is not up, it will continue to attempt to advertise until it is successful. When it succeeds, the agent has publicly declared a commitment to take on any future requests that fit the advertisement's constraints. A successful advertisement behavior also places the agent in a state in which it cannot leave the system without withdrawing the advertisement (otherwise the advertisement would not be a true commitment). An agent that has undertaken a server role must thus send a successful UNADVERTISE KQML message before it can intentionally terminate. We view advertisement and unadvertisements, then, as commisive speech acts. Note however that the advertisement makes no meta-commitment about the length of the server role—the agent can leave the system at any time but must unadvertise (decommit) first.

The primary behavior of an agent taking on a server role is to respond to requests that the agent achieve objectives that meet the advertised conditions. These requests might contain conditions such as deadlines, but the server role is still an extremely basic one. In particular, a server does not enter into a shared plan [9] or communicate with the requester about its internal objectives [6].

Our most carefully worked out and empirically validated server role behaviors center on answering queries. We have created a class of agents called *information agents* that can answer one-shot and periodic queries, or monitor for conditions, on external information sources. This class of agents can work either with traditional databases or the more ubiquitous Internet information sources such as WWW pages and Usenet news. Upon receiving a KQML request (i.e. ASK, ASK-ALL, STREAM-ALL, etc.), the server creates a local commitment to achieve the stated objective. By definition, the agent cannot give up that commitment until it is achieved or believed to be unachievable—this could occur because of pending agent failure (i.e. scheduled maintenance) or an external source failure. In either case the requester is notified (via the SORRY performative). Alternately, with long-term requests (i.e. monitor for a condition), the requester might wish to retract the request (most commonly because the requester is withdrawing from the society). This also causes the server to give up on the request.

3.3 Matchmaker Role Behaviors

A matchmaker's most important feature is that it has a well-known name, i.e., all agents have a predefined, static procedure for contacting a matchmaker.[3] An agent

[3]For example, using the Lockheed KAPI Agent Name Server, the matchmaker can be given a fixed name to satisfy this constraint (it does not have to reside at a fixed physical address or port

fulfilling the matchmaker role participates in at least two different types of conversations. The first is its interaction with agents who take on server roles. These agents send advertisements and unadvertisements which are used to update the matchmaker's local database. The second is its interaction with requesters, agents that want to find the names of servers for their objectives. These interactions take place exactly as if the matchmaker were a server—in fact in our implementation matchmakers are a subclass of information agents and reuse all of their standard behaviors.

Should there be more than one matchmaker, and if so, how many? Certainly as the number of agents grows, the ability of a single matchmaker to handle all requester queries in a timely manner will falter. The onset of this saturation point can be mitigated by local caching as previously mentioned. While it is possible to use several "well-known" matchmakers, this solution is not a very open one. Instead, we assert that complex multi-matchmaker organizations should be insulated from ordinary requesters by allowing *brokering matchmakers*, i.e., matchmakers that may (transparently) contact others in order to answer queries. Any system designed to use a single matchmaker can then add an arbitrary multi-matchmaker organization at any time, and it will be transparent to the existing agents (other than a potential increase in matchmaker query service times). We will not discuss multi-matchmaker organizations further in this paper; one possible organizational structure often mentioned might mirror Internet domain name resolution systems [7].

4 Brokering

In general, the process of brokering involves how one agent with an objective comes to have that objective achieved by another agent. We intentionally and clearly define brokering behavior as separate and different from matchmaking (agent-name-finding) behavior. Hybrid organizations, where for example a requester uses a matchmaker to find a broker, are both possible and desirable. The brokering process again involves three different agent roles (Figure 2):

Requester: an agent that has an objective that the agent wants to has achieved by another agent.

Broker: an agent that knows the names of some other agents and their corresponding capabilities, and advertises its own capabilities as some function of the capabilities of these other agents. From the requester's point of view, a broker is

number). Thus an agent becomes a matchmaker—committing to answer queries about other agent's capabilities—just by using this well-known name.

indistinguishable from a server (with perhaps slower response time and higher price characteristics).

Brokered Server: an agent that has committed *to the broker* to taking on a predetermined class of objectives.

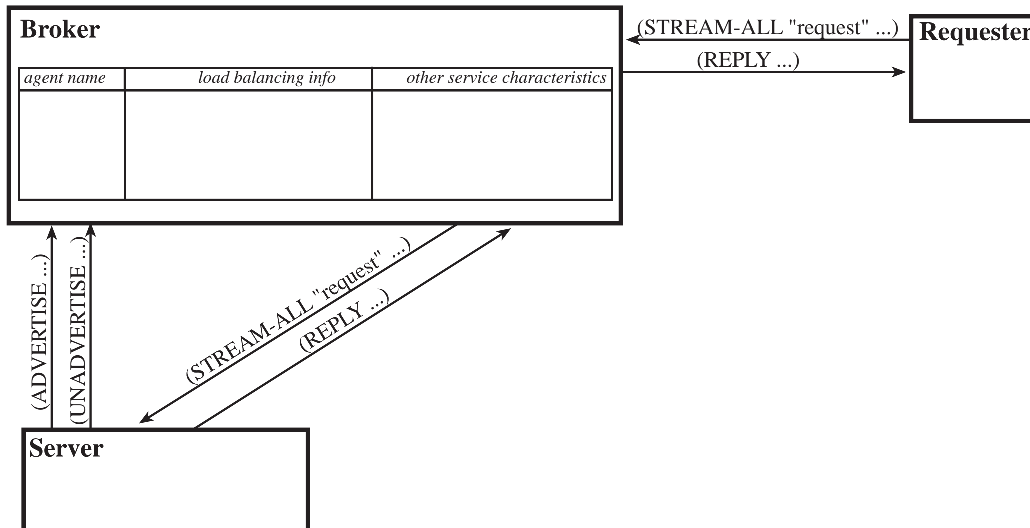


Figure 2: Communications between three agents taking on the roles of broker, requester, and brokered server.

4.1 Requesters and Servers in a Brokered Organization

By design, an agent making a request changes little in a brokered system. In a purely brokered system there are no matchmakers, so a requester must rely on knowledge of who the brokers are. In the limit, such as "facilitator" systems [8], an agent goes through a single broker for *everything*, so the necessary pre-compiled knowledge at an agent differs little from a matchmaker system. Although allowing brokers to specialize in different classes of objectives can add to the amount of static knowledge agents need in order to make requests, it does not necessarily limit the openness of the resulting system, since many different agents at many different physical locations may advertise and unadvertise with the appropriate brokers just as in a matchmade system. What *is* quite different is the ability to add new classes of brokers dynamically, and the way the system acts under a failure of a broker or matchmaker.

Similarly, agents taking on a server role in a brokered organization use mostly the same behaviors. The only difference is in the advertising behavior. In a purely brokered organization, an agent that wants to take on a server role advertises to the appropriate broker—knowledge of that broker name must be static. This is a clear case where hybrid organizations can excel: a server may contact a matchmaker in order to find an appropriate broker, and if a broker does not exist, may advertise with the matchmaker directly.

4.2 Brokering Role Behaviors

The core behavior of an agent taking on a brokering role is quite different from any of the previous roles (requester, server, matchmaker) that we have discussed. It does share some basic behaviors with each of these other roles:

- Like a matchmaker, a broker accepts advertisements and unadvertisements from servers.
- Like a server, a broker accepts requests from requesters, and supplies appropriate replies.
- Like a requester, a broker (having accepted an objective from another requester) passes on an objective to a server agent and processes the replies.

The behaviors associated with advertisements are identical for brokers and matchmakers. Unlike real servers, however, when a broker accepts a request, it does nothing locally to process the request, but instead uses its internal matchmaking information to choose a server for the request. It then contacts the server and passes on the request and formulates an expectation to process the eventual reply(s).

Because a broker stands at a filtering and control point in the information flow of an organization, one natural service to provide is load balancing—the broker can choose servers not just to satisfy requester objectives, but also to optimize resource usage of the servers. However, powerful, low-overhead load balancing algorithms require a strong constraint on server behavior: servers cannot advertise with more than one broker (or with both a broker and multiple matchmakers in a hybrid system). There was no such constraint in a pure matchmaker system—servers can advertise with as many matchmakers as they want. In a brokered system the broker needs to control all request flows to the servers in order to provide cheap and efficient load balancing (this may also include keeping track of other information, such as the amount of work in each request and the resources available at each server). There

do exist more complex load balancing algorithms that do not require this constraint (e.g., where the load at the remote agent is reverified when the request is made [14]), but they are also applicable *at the requester* in a pure matchmaker system. In our implementation, the work associated with each request is assumed to come from a single statistical distribution, and agents are assumed to have similar computational resources.

5 Performance tradeoffs: some experimental results

The decision to use matchmaking or brokering to solve the connection problem offers many performance tradeoffs. System performance is dependent upon a large number of parameters, including the rate at which service requests are generated; the number of servers in the system; the time needed by each server to fulfill a request; agent failure rates, and so on.

5.1 Experimental systems

We will consider two alternative systems. Each consists of some number of homogeneous servers and requesters. All agents run on serial processors, and the basic service action is non-interruptible. In the "Matchmade" system (Figure 1), each server advertises itself to one *matchmaker* agent. Requesters query the matchmaker to obtain a current list of servers, choose one randomly, and send it a service request. In the "Brokered" system (Figure 2), servers advertise themselves to a distinguished *broker* agent. Requesters send all service requests directly to the broker, who farms them out to the servers—seeking to equalize the load among them.

Our implementation of these systems consists of real WARREN agents, who experience real communication and processor latencies, etc. The broker and matchmaker are the same agents used in the actual portfolio management system. Servers and requesters are instances of WARREN servers and requesters, but we have standardized on a single abstract service to be provided (modeled after stock ticker services).

There are several parameters that we can vary in our systems. We can specify the number of servers (N), the time required by each server to fulfill a request (T), and the periodic rate at which new requests are generated (P).[4] The actual service time for each request and the period between requests are generated randomly; the service

[4] Given some reasonable independence assumptions, the number of requesters is irrelevant. There is no difference between five requesters each sending two requests per minute, and one requester sending ten requests per minute. We therefore only discuss variation of the request rate.

time is distributed normally around the mean T, and the request generation period is distributed exponentially around the mean P. But because we are using real rather than simulated agents, some parameters are beyond our control, such as the inter-agent communication latency, the computational needs of the broker or matchmaker, and the amount of time spent by the servers and requesters on planning, scheduling, and other internal operations.

We make some further assumptions about the ranges of values that our system parameters will take on. First, we assume that the service time is relatively long compared to the computational overhead of the matchmaker, broker, and servers themselves. This is consistent with actual WARREN agents, which typically require 30 seconds or more to access Internet resources. Second, we assume that the number of servers is relatively small (we have experimented with systems of up to 20 servers), which is again consistent with the operational WARREN system.

5.2 Theoretical expectations

The main performance attribute which we will measure is R, the total elapsed time taken by a requester to satisfy a service objective. It includes:

- Time spent by the server providing the service (defined by the controllable parameter T).
- Time spent planning and scheduling by the requester, matchmaker or broker, and server, and time spent communicating between agents. This is a fixed feature of our agents, denoted F.
- Time spent waiting at a server which is busy fulfilling prior requests, denoted Q. This is a function of the request generation period, P, and of the number of servers, N.

Our system can be roughly described by a queuing network model [12]. According to queuing network theory, the total elapsed time to fulfill a request is $R = D + Q$, where D is total computational demand of the request (in our case, $D = T + F$). Note that R, like Q, is a function of the request generation period and of the number of servers. If requests are generated at a rate greater than the maximum system throughput, i.e. if

$$P < \frac{D}{N}$$

then the system will be *saturated* and R will grow without bound. Otherwise, a fundamental result of queuing theory is that the expected elapsed time per request is:

$$R = \frac{D}{1 - \frac{D}{PN}}$$

This result depends on the service request load being equally distributed across all servers, or else the elapsed time per request will be greater. Since the brokered system precisely balances the load, while the matchmade system only stochastically does so, we would expect the broker to provide better elapsed times.

5.3 Experiment One: Response Time

In our first experiment, we validate the theoretical model, and empirically compare the brokered and matchmade systems. For a fixed service time and number of servers, we vary the request generation period (the independent variable). For each period, we generate 100 requests, and measure the mean and standard deviation of their elapsed times (the dependent variable). Figure 3 shows the results as the request generation period varies between 5 and 15 seconds, for systems with 3 servers and a service time of 15 seconds.

Note that despite the crudeness of the model, it gives a good indication of the expected response time, especially for larger request generation periods. (For shorter periods, when the system is more highly loaded—or even saturated—our measured values fall below the predictions because we are performing only 100 queries. The earlier requests experience less queuing time, and so skew the results downwards.) In any case, it is clear that the load balancing of the brokered system confers a response time advantage over the matchmade system.

5.4 Experiment Two: Server Failure and Recovery

In our second experiment, we investigate the effect of server failure and recovery on our two systems. We begin with three servers, and fix the service time and request generation period at 15 and 10 seconds, respectively. After five minutes, we kill one of the servers, and after five more minutes, we kill a second one. Five minutes after that, we bring one of the severs back on line, and then ten minutes later the third one returns. When a server dies, it sends a SORRY message for each outstanding request, and they must be reallocated (by either the broker or the original requester) to another server.

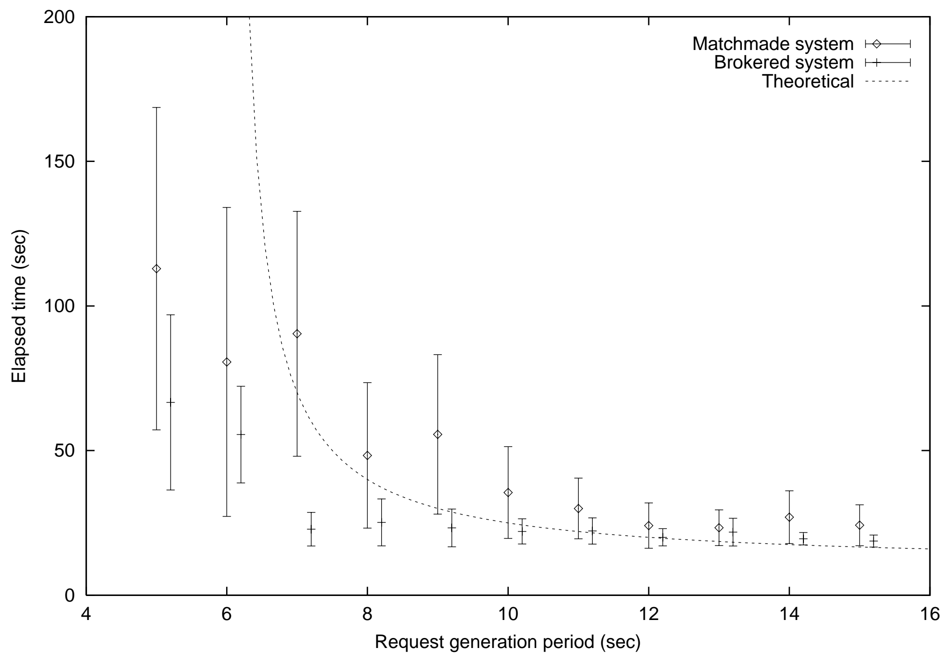


Figure 3: Mean time for 100 service requests as a function of request generation period

Figure 4 shows the results of this experiment. Each point represents the completion of a service request. The dashed line represents the number of servers active at each time. The response-time superiority of the brokered system is very dramatic. It stems from the difference in behavior of the two systems when the failed servers come back online. When there is only one server, the system is saturated, so that server begins to build up a large backlog of requests. When the second and third servers become available again, the requesters in the matchmade system continue to allocate one-half or one-third of their requests to the overloaded server, so the backlog persists for a long time.[5] In the brokered system, on the other hand, all new requests are allocated to the new server, allowing the backlog at the congested server to quickly dissipate.

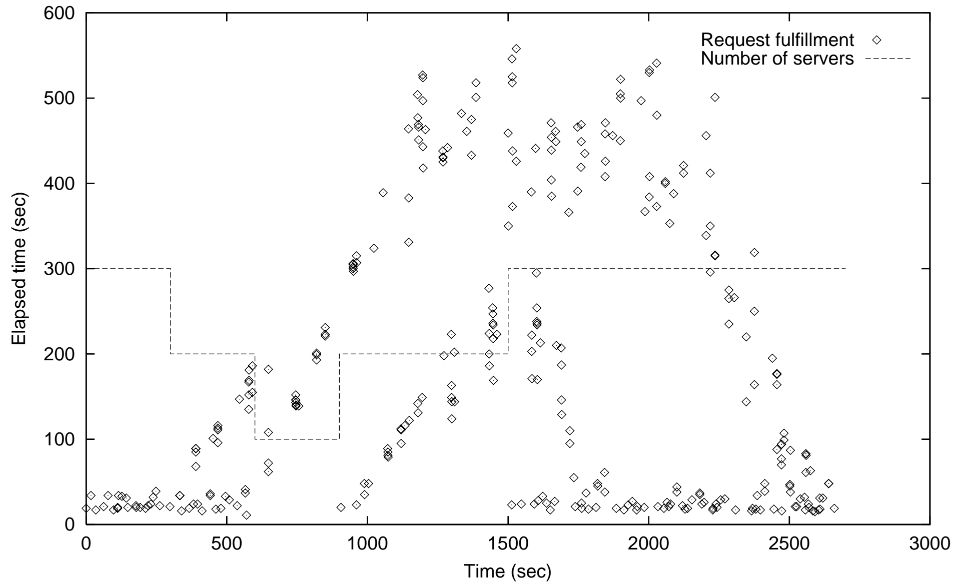
6 Conclusions

This paper has provided a detailed description of two very important basic open multi-agent system behaviors—matchmaking and brokering. For each behavior, we described the component agent roles, and the individual agent behaviors within those roles. When design options were present, we discussed the effect of the choices on common performance criteria (e.g., local caching to increase reliability in the face of matchmaker failure). Finally, we conducted an experimental evaluation of these pure organizations. We showed that they obey basic theoretical models, but also that they can differ substantially in some characteristics such as recovery times when agents enter and exit an open system.

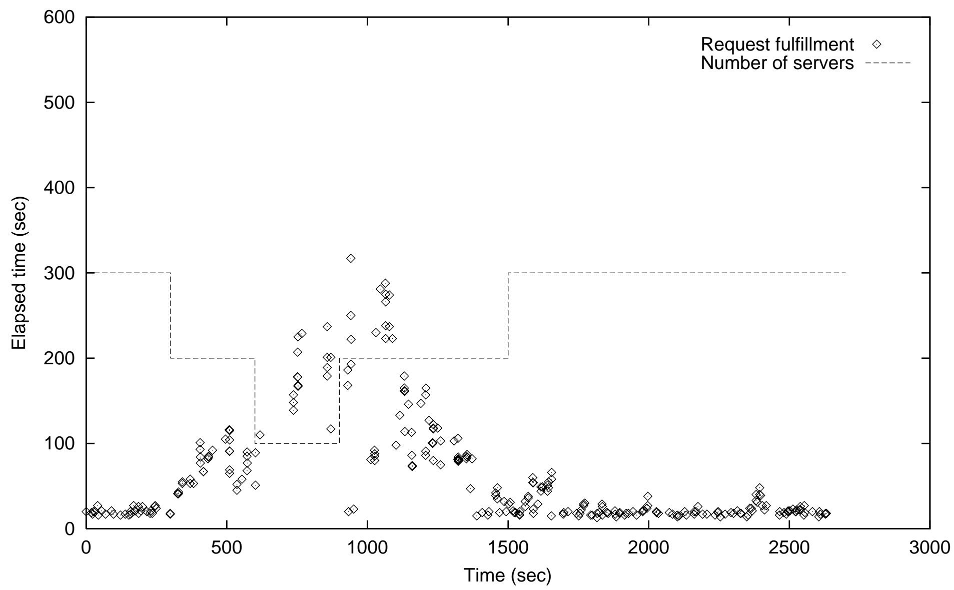
Matchmaker-based organizations can be expanded into organizations such as decentralized markets. They offer the most flexible responses to arbitrary agents entering and exiting the system, and each agent keeps total control over its own control decisions. However, pure matchmaker systems present a single point of failure (mitigated by local caching or multi-matchmaker structures), and each agent needs to be smart enough to construct a meta-query and evaluate the resulting alternative server choices. No easy load balancing is possible. Using a matchmaker has slightly higher overhead due to the extra queries, and there is no centralized ontological translation facility.

Brokered organizations can be expanded into centralized markets or traditional bureaucratic managerial units. They can also handle the dynamic entry and exit of agents, and provide an easy way to do load balancing, centralized ontological translation, or even simple integration facilities. However, they suffer from the need of

[5] This effect could be reduced if requesters make an effort at active load balancing.



(a) In a matchmade system



(b) In a brokered system

Figure 4: Effect of server failure and recovery

agents to have static knowledge of the brokers and also form a communication bottleneck (since all requests and replies need to go through the brokers). Worst of all, they form a single point of failure that cannot be mitigated via local caching.

The solution that we are currently working on is a hybrid system with both matchmakers and brokers. By design, our specified agent behaviors work interchangeably with both organizational roles. A hybrid system allows us to capitalize on the lower overhead and efficient load balancing of a brokered system while retaining the dynamic naming capabilities and greater robustness of the matchmaker system.

Acknowledgments

This work has been supported in part by ARPA contract F33615–93–1–1330, in part by ONR contract N00014–95–1–1092, and in part by NSF contract IRI–9508191.

References

[1] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(3):213–261, 1990.

[2] P.R. Cohen and H.J. Levesque. Communicative actions for artificial agents. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 65–72, San Francisco, June 1995. AAAI Press.

[3] C. Collet, M.N. Huhns, and W. Shen. Resource integration using a large knowledge base in Carnot. *Computer*, December 1991.

[4] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20(1):63–109, January 1983.

[5] K. Decker, M. Williamson, and K. Sycara. Modeling information agents: Advertisements, organizational roles, and dynamic behavior. In *Proceedings of the AAAI-96 Workshop on Agent Modeling*, 1996.

[6] Keith S. Decker and Victor R. Lesser. Designing a family of coordination algorithms. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 73–80, San Francisco, June 1995. AAAI Press. Longer version available as UMass CS-TR 94–14.

[7] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management CIKM'94*. ACM Press, November 1994.

[8] M.R. Genesereth and S.P. Katchpel. Software agents. *Communications of the ACM*, 37(7):48–53,147, 1994.

[9] B. Grosz and S. Kraus. Collaborative plans for group activities. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambéry, France, August 1993.

[10] D. Kuokka and L. Harada. On using KQML for matchmaking. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 239–245, San Francisco, June 1995. AAAI Press.

[11] Y. Labrou and T. Finin. A semantics approach for KQML. In *Proceedings of the Third International Conference on Information and Knowledge Management CIKM'94*. ACM Press, November 1994.

[12] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice Hall, 1984.

[13] J. S. Rosenschein and G. Zlotkin. Designing conventions for automated negotition. *AI Magazine*, pages 29–46, Fall 1994.

[14] J. A. Stankovic. Simulations of three adaptive, decentralized controlled, job scheduling algorithms. *Computer Networks*, 8:199–217, 1984.