A Taxonomy of Middle-Agents for the Internet[*]

H. Chi Wong and Katia Sycara
The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA, 15213
hcwong@cs.cmu.edu, katia@cs.cmu.edu

Abstract

Multi-agent systems deployed in open networks, where agents can come and go, need middle-agents. Like middle-men in physical world, middle-agents assist in locating and connecting the ultimate (information or goods) provider with the ultimate requester in the electronic world.

Different types of middle-agents exhibit different performance characteristics in terms of privacy, robustness, adaptiveness, etc. Which type of middle-agent to use in a system thus depends on the requirements of the application. It is therefore essential that we identify different design possibilities for middle-agents, and investigate the requirements they satisfy.

In this paper, we identify a basic list of dimensions along which middle-agents can differ from one other, and derive a taxonomy.

To better characterize different types of middle-agents, we give a formal specification of their protocols. Formal specification is important because it can improve communication between designers and developers, and enable formal verification of these protocols.

Our work could be the first step towards standardizing middle-agents and their protocols.

Introduction

As the Internet moves from a networked set of documents to a set of services, provided by autonomous agents, a crucial problem that arises is the connection problem (Davis & Smith 1983) - finding the other agents who might have the information or other capabilities an agent needs. SUN's JINI (Sun), for example, assumes a networked system of services that can locate each other through registries of services. In (Decker, Sycara, & Williamson 1997), it was proposed that agents that help others locate agent providers of services be called *middle-agents*. Like middle-men in physical world, middle-agents assist in locating and connecting the ultimate provider of services (information, expertise, or goods) with the ultimate requester in the electronic world. In a capability-based coordination, what providers can provide (i.e., the providers' functionality) and requesters need are given by specifications called *capabilities*. In other words, agents are not found by their names only, but by their functionality.

Different systems define their middle-agents differently. For example, facilitators in Genesereth's federated systems (Genesereth 1997) and SRI's Open Agent Architecture (Martin, Cheyer, & Moran 1999), and matchmakers and brokers in Retsina (Sycara *et al.* 1996) all differ in their interactions with service providers and requesters. And as demonstrated through experimentation (Decker, Sycara, & Williamson 1997), different types of middle-agents exhibit different performance characteristics in terms of privacy, robustness, adaptiveness, etc. For example, systems that use matchmakers are more robust, while those using brokers achieve a better load balancing. The requirements of an application thus dictate the type of middle-agent to be used in the system. It is therefore essential that we identify different design possibilities for middle-agents, and investigate the type of requirements they satisfy.

In this paper, we take one of the first steps towards a comprehensive and systematic study of middle-agents by formulating a taxonomy of middle-agents. To formulate the taxonomy, we first identify a number of dimensions along which middle-agents can differ, and then possible values for these dimensions. In this paper, we focus on capability-based middle-agents involved in agent location and transaction intermediation.

To better characterize different types of middle-agents, we give a formal specification of their protocols. Formal specification is important because it

[*]This work is supported by ONR contract N00014-96-1222, DARPA contract F30602-98-2-0138.

can improve communication between designers and developers, and enable formal verification of these protocols.

Our work takes the first step towards standardizing middle-agents and their protocols.

The Model

In an open multi-agent system, there are two types of agents: *end-agents* and *middle-agents*. End-agents need or can offer services. Middle-agents exist to enable interactions among end-agents. End-agents act as *providers* when they offer services; and act as *requesters* when they need them. Of course, an agent can act both as a provider and as a requester in a system. Strictly speaking, middle-agents offer services too; services they offer include locating end-agents for each other, and intermediating their transactions and dispute resolutions. In this paper, we focus on agent location and transactions themselves.

In capability-based coordination, providers and requesters specify services they provide or need in *capabilities* and *requests*. Capabilities are specifications of services providers can offer. They are sometimes accompanied by *service parameters*, which specify conditions under which services will be offered. For instance, price, availability, quality of service are all service parameters. An example of a capability and its service parameter is: agent X is capable of providing information about weather for any US city for 2 cents per query. Requests are specifications of services requesters need. They can be accompanied by *preferences*, which are counterparts of service parameters.

In our model, agents communicate through message passing, and are capable of intelligent internal processing.

In the rest of this work, we refer to requeters as she, providers as he, and middle-agents as it.

A Taxonomy of Middle-Agents

The goal of this section is to determine different modes under which middle-agents can operate. To this end, we have identified 6 dimensions along which middle-agents may differ, and the possible variations within each of the dimensions. We do not claim that these 6 dimensions are complete or unique. But we believe that they constitute a basic set in characterizing typical location and transaction interactions, and in discriminating different types of middle-agents.

To find agents with desired capabilities, end-agents can adopt two different approaches. They can 'push' or 'pull'. In the 'push' approach, they send information about themselves to the middle-agent. In the 'pull' approach, they ask the middle-agent information about what is available in the system. Thus, the first dimension that characterizes a middle-agent is:

P1: Who sends information to middle-agents?

There are two possibilities[1]: providers (0) or requesters (1). Providers and requesters need to act in complementary ways: if providers 'push', requesters 'pull'; and vice-versa.

When end-agents send information about themselves to a middle-agent, the information becomes public, in that it will be known by the middle-agent and potentially by those who get information from the middle-agent. Depending on their privacy concerns, end-agents may want to send only capabilities/requests, or they may want to include parameters/preferences as well. Thus, the second dimension of a middle-agent description is:

P2: How much information is sent to the middle-agent?

There are again two possibilities: capabilities (or requests[2]) (0), or capabilities plus service parameters (or requests plus preferences[3]) (1).

Once it is determined who sends what to the middle-agent, we can ask:

P3: What happens to the information middle-agents receive?

Again, there are two possibilities here. It can be broadcast[4] (0) or kept in a local database(1).

If the information middle-agents receive is stored in the middle-agents' databases, the following question arises:

P4: How is the content of the database used?

Here, again, there are two possibilities. It can be browsed (0) or queried (1). If the database is browsed, then the 'puller' of the information receives the content of the whole database. Otherwise, the 'puller' gets only a subset of the information in the database. The subset is determined by the information the 'puller' specifies in the query.

[1]For a purpose that will become clear later, we enumerate the values within each of the dimensions, starting from 0.

[2]The alternative between capabilities and requests is determined by the value of *P1*.

[3]This alternative is also determined by the value of *P1*.

[4]In broadcast systems, 'pullers' must register with middle-agents to express their interest in getting the broadcast. This is because, in an open system, it is not known which agents are in the system at any given time.

The process of determining the relevant subset - called *matching* - (e.g., of the advertised capabilities of the providers to an issued request) - will be discussed later in this paper.

Note that from the 'puller''s point of view, browsing and querying lead to different privacy guarantees. The 'puller' does not reveal information about itself to the middle-agent when it browses; the information specified in the query is revealed, however, when the 'puller' queries. This leads us to the fifth dimension of a middle-agent:

P5: How much information is specified in a query to the middle-agent?

There are two possibilities. One can provide only the essential information - capabilities/requests (0); or one can provide additional information - service parameters/preferences (1) - as well. Depending on the amount of information one specifies in a query, a greater or lesser amount of privacy is guaranteed.

Finally, middle-agents may or may not act as intermediaries in end-agent transactions:

P6: Does the middle-agent intermediate transactions?

Here too, we have two possibilities: yes (0) or no (1). There are a number of reasons for middle-agents to intermediate transactions. For example, they may do so to implement anonymity of the parties involved in the transaction; to guarantee fairness; or to collect affidavits for possible future disputes. When middle-agents intermediate transactions, they get to know who transacted with whom, which may be sensitive information by itself. However, if end-agents trust middle-agents more than they trust each other, then having middle-agents intermediate transactions can be preferable. For example, in the NetBill electronic payment system (Cox, Tygar, & Sirbu 1995), a trusted middle-agent is used to make the exchange of payment and delivery of goods atomic (by, for example, acting as an escrow agent), thus guaranteeing the fairness of a transaction.

With the above 6 dimensions and their respective values, we can now devise a taxonomy of middle-agents. Given that each dimension only allows for 2 possibilities, we can use an identification scheme where different types of middle-agents are identified by binary numbers of 6 digits: given a middle-agent, its type identifier is a number whose n-th digit reflects the value of its n-th dimension.

Not all combinations of 0s and 1s are meaningful. For example, if $P3 = 0$, then the values of $P4$ and $P5$ are irrelevant. That is, in a system where the 'puller' receives broadcasts, one cannot talk about databases or database queries. Using valid combinations of values in the 6 dimensions we consider, we can obtain 28 different types of middle-agents. For example, middle-agents [001100], [011100], and [011110] are different variants of what is known in the literature as matchmakers (Decker, Sycara, & Williamson 1997) (Fig. 1). A matchmaker allows providers to advertise their capabilities (and service parameters), and requesters to send requests. In response to a request, a matchmaker returns the contact information of appropriate service providers. The requester agent then chooses a service provider and interacts with it directly. Matchmakers do not intermediate transactions. Middle-agent [011111]

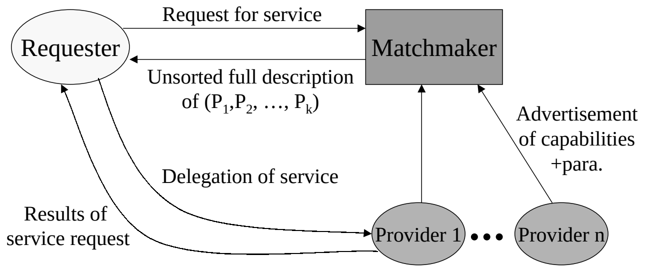
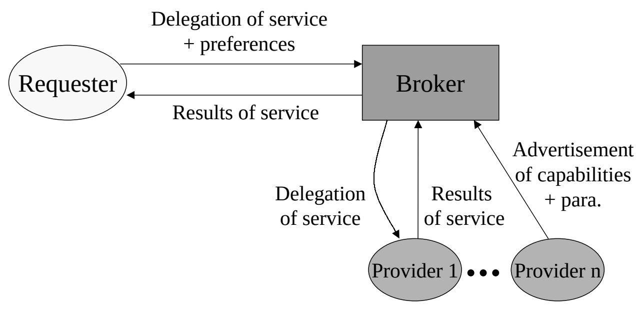

Request for service
Requester
Matchmaker
Unsorted full description of (P_1,P_2, ..., P_k)
Advertisement of capabilities +para.
Delegation of service
Results of service request
Provider 1
Provider n

Figure 1: Matchmaker.

is known in the literature as a facilitator (Martin, Cheyer, & Moran 1999) (Fig. 2). Facilitators work as follows. 1) Provider agents advertise their capabilities with the facilitator. 2) The facilitator keeps the advertised capabilities in its database. 3) A requester makes a request with the facilitator for a service provider that can provide a particular service. 4) in response to a request for service, the facilitator selects one of the appropriate providers and delegates the service request to it. The provider does the service and returns the result to the facilitator, who then forwards the result to the requester. Note that, in contrast to match-

Figure 2: Facilitator.



makers, facilitators "stay in the loop" throughout the transation; i.e., they intermediate transactions. The OAA system uses such a facilitator in its architecture.

Example Protocols

In this section, we present informal protocol specifications. Different protocols presuppose different local processing in andividual agents. For example, if a middle-agent only allows its database to be browsed, then no major processing is required of it. On the other hand, if its database can be queried, then it needs a matching engine capable of filtering out database entries that match the specification provided in the query. If both parameters and preferences are used in the matching, then the output produced by the matching is a sorted list; otherwise it is unsorted. LARKS (Sycara *et al.* 1998), for example, is a matching engine and a language for capability and request specification used in Retsina.

Matching processes can be studied independently of protocols. Protocols concern interactions among agents, while matching takes place within a single agent. In this paper, we focus on the protocols.

Protocols

Due to space constraints, we present protocols only for facilitated systems.

In what follows, P denotes provider, R denotes requester, and M denotes middle-agent; $A \rightarrow B : m$ denotes agent A sending message m to agent B; if an agent can send m_1 or m_2, we denote the alternative by $m_1 \mid m_2$. The actual format of the messages can be expressed as KQML (Finin *et al.* 1994), FIPA (FIPA 1997), or other agent communication language. Some of our performatives can be mapped to one of the reserved, predefined, KQML performatives. For example, our advertise-capability and make-request can be expressed in terms of the KQML's "advertise". Finally, *pid, rid, cap, param, req, pref, input*, and *res* are parameters to be instantiated with provider and requester ids, capabilities, (service) parameters, requests, preferences, input to and result of an execution respectively.

Our example (Fig 3) shows the protocols needed in a facilitated ([011111]) system. The two protocols in Fig. 3 are for advertising and delegating respectively. Note that, in the delegation protocol, a matching process occurs in between delegate and perform (the facilitator matches the request with advertisements in its database), and provider P is the one that best matches the requester's preferences. Also note that the requester is not consulted once she submits the delegate message; she will only receive the result of the transaction. From the point of view of privacy, both requesters and providers expose information about themselves to the facilitator.

Advertisement protocol :

$$\begin{array}{rl} P \rightarrow M : & \textsf{advertise-capability}\ \textit{pid cap param} \\ M \rightarrow P : & \textit{success} \mid \textit{fail} \end{array}$$

Delegation protocol :

$$\begin{array}{rl} R \rightarrow M : & \textsf{delegate}\ \textit{req pref (input)} \\ & \text{[matching: internal to } M\text{]} \\ M \rightarrow P : & \textsf{perform}\ \textit{req pref (input)} \\ P \rightarrow M : & \textsf{result}\ \textit{res} \\ M \rightarrow R : & \textsf{result}\ \textit{res} \end{array}$$

Figure 3: Protocols for facilitated systems.

Protocols: A Formal Specification

In this section, we use a state machine formalism to specify protocols. We chose to use *Input/Output (IO) automata* (Lynch 1996), but other specification tools such as Z, CCS, and CSP, could have been used.

Here, we give a brief overview of IO automata. IO automata is a state machine formalism for describing asynchronous concurrent systems in general. Formally, the first thing that gets specified for an IO automaton is its "signature", which is a description of its input, output, and internal actions. Given a signature S, $acts(S)$ is the set of all actions in S.

An IO automaton A, henceforth simply called an automaton, consists of five components:

- $sig(A)$: a signature;
- $states(A)$: a (possibly infinite) set of *states*;
- $start(A)$: a non-empty subset of $states(A)$, known as *initial states*;
- $trans(A)$: a *state-transition relation*, where

  $trans(A) \subseteq states(A)$ x a$cts(sig(A))$ x $states(A)$;

  this relation must have the property that for every state s and every input action π, there is a transition $(s, \pi, s') \in trans(A)$;
- $tasks(A)$: a task partition.

  We will not make use of the notion of $tasks(A)$ in this paper.

Typical output actions in IO automata modeling message passing processes are $send(m)_{ij}$, which represents process P_i sending a message m to process P_j. Typical input actions are $receive(m)_{ji}$, which represents process P_i receiving a message m from process P_j. When the automaton performs an action, it may also transition to a new state.

In our specifications, the transition relation is described in a precondition-effect style. The code specifies the condition under which the action is permitted to occur, as a predicate on the pre-state s. Then it describes the changes that occur as a result of the action, in the form of a simple program that is applied to s to yield s'. The entire piece of code gets executed indivisibly, as a single transition.

In what follows, we model each agent as a process, and specify each process as an IO automaton. In the specifications, messages are concatenations of atomic messages, possibly preceded by a performative. For example, in the specification below, the middle-agent can send only messages *success* or *fail* to P, indicating the result of capability advertisement. On the other hand, it can receive from R 'delegate' messages (delegate *req pref input*), which consist of the primitive delegate, followed by specifications of a request, its associated preferences, and the input needed for execution of the service.

Atomic messages can be drawn from different sets. In our specification, *req, pref, cap, param, pid, rid, tid, input, and res* are drawn respectively from the set of requests, preferences, capabilities, parameters, provider ids, requester ids, transaction ids, inputs needed for service executions, and results they generate.

Due to space constraints, we present only the facilitator automaton.

Signature :

Input:

$receive_{pm}$(advertise-capability *pid cap param*),
$receive_{rm}$(delegate *req pref input*),
$receive_{pm}$(result *res tid*).

Output:

$send_{mp}(msg)$, $msg \in \{success, fail\}$,
$send_{mp}$(perform *req pref input tid*),
$send_{mr}$(result *res*).

Internal:

$match_m(db^c, (rid,\ req\ pref\ input))$.

States :

db^c**:** a capability database, initially empty;

*ack-queue***:** a queue whose elements are pairs (*pid, success/fail*), each enclosing the result of an advertisement. This variable is initially empty;

*delegation-queue***:** a queue whose elements are pairs (*rid, req pref input*), each associating delegations to their respective delegators. Requests in this queue have been received by the middle-agent, but have not yet been processed;

match-found-queue : a queue whose elements are pairs (*pid*, (*rid, req pref input*)), each associating delegations to the result of the match - the id of the provider that best matches the request in the delegation;

*result-waiting-set***:** a set whose elements are pairs (*rid, tid*) mapping transaction ids - identifying requests - to their respective requester ids. *tid*'s that appear in this set identify requests that have been sent to the matching provider, but that are still waiting for the result;

*res-queue***:** a queue whose elements are pairs (*rid, res*) associating results of service requests with their requesters' ids. Results in this queue are ready to be returned to their requesters.

Transitions :

```
receive_pm(advertise-capability pid cap param)
  Effect: $db^c$ := add (pid cap param) to $db^c$;
  if (add (pid cap param) to $db^c$) is successful then
    ack-queue := append(ack-queue, (pid, success))
  else ack-queue := append(ack-queue, (pid, fail)).

send_mp(msg)
 Precondition: head(ack-queue) = (pid, msg);
 Effect: ack-queue := tail(ack-queue).

receive_rm(delegate req pref input)
 Effect: delegation-queue :=
   append(delegation-queue, (rid, req pref input))

match_m($db^c$, (rid, req pref input))
 Precondition:
   head(delegation-queue) = (rid, req pref input);
 Effect:
   delegation-queue := tail(delegation-queue);
   match-found-queue :=
     append(match-found-queue
            (pid, (rid, reqprefinput))),
     where pid is the best match for the request.

send_mp(perform req pref input tid)
 Precondition:
   head(match-found-queue) =
     (pid, (rid, req pref input));
 Effect:
   match-found-queue := tail(match-found-queue),
   result-waiting-set :=
     result-waiting-set ∪ (rid, tid).

receive_pm(result res tid)
```

```
Effect:
  result-waiting-set :=
    result-waiting-set - (rid, tid),
  res-queue := append(res-queue, (rid, res)).
```

```
$send_{mr}$(result res)
  Precondition: head(res-queue) = (rid, res);
  Effect: res-queue := tail(res-queue).
```

Conclusion

In this paper, we have presented a taxonomy for middle-agents, and given a formal specification of their protocols.

This work is significant for three reasons. First, middle-agents are becoming common in open MASs, and even make inroads in the commercial world (Sun). Second, different middle-agents exhibit different performance characteristics in terms of privacy, robustness, adaptiveness, etc. And finally, MASs using different middle-agents are being asked to work together. Thus, it is critical to provide a taxonomy of different types of middle-agents, with their respective characteristics. This taxonomy will enable designers of MASs to choose the type of middle-agent that best satisfy the requirements of their application. As agents are proliferating on the Internet, this could have significant impact. Also composition o middle-agents can lead to interesting protocols; e.g., ContractNet is a combination of blackboard and brokering middle-agents.

The formal protocol specifications, in their turn, can guide developers in concrete implementation of their protocols. In addition, they enable formal verifications of the protocols.

Finally, this work provides the first step towards standardization.

To our knowledge, no one has presented such a comprehensive and systematic taxonomy for middle-agents. The only other similar effort (Decker, Sycara, & Williamson 1997) focuses on quantitative performance comparisons between matchmaking and brokering.

The middle-agents in our taxonomy are 'pure', in that they are capable of only one variant of coordination. For example, a middle-agent is either a broker, a matchmaker, or a blackboard. This pure classification is useful for reference purposes. In practice, however, they do not need to be pure. They can be *static hybrids* and provide more than one variant of coordination, depending on the demands of the end-agents; or they can be *dynamic hybrids*, and change from one variant to another over time, depending on the environmental conditions. That is, static hybrids can, for instance, do both matchmaking and brokering at the same time, while dynamic hybrids would do only one at a time, depending on the conditions of the system (Decker, Sycara, & Williamson 1997).

We have a few directions for future work. We plan to investigate middle-agents in our taxonomy with respect to a number of characteristics including privacy, robustness, and load-balacing. We also plan to expand our taxonomy to include other types of mediating agents (inter-operators, for example, are a different type of mediating agents). Finally, we plan to implement a universal middle-agent - one that would provide all the services taken into account in our taxonomy.

References

Cox, B.; Tygar, J. D.; and Sirbu, M. 1995. Netbill security and transaction protocol. In *Proceedings fo the 1st USENIX Workshop on Electronic Commerce.*

Davis, R., and Smith, R. G. 1983. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence* 20(01):63–109.

Decker, K.; Sycara, K.; and Williamson, M. 1997. Middle-agents for the internet. In *Proc. IJCAI-97.*

Finin, T.; Fritzson, R.; McKay, D.; and McEntire, R. 1994. Kqml as an agent communication language. In *Proc. CIKM-94.* ACM Press.

FIPA. 1997. Fipa'97 specification part 2: Agent communication language. http://drogo.cselt.stet.it/fipa/.

Genesereth, M. 1997. An agent-based framework for interoperability. In Bradshaw, J. M., ed., *Software Agents.* AAAI Press. 317–345.

Lynch, N. 1996. *Distributed Algorithms.* Morgan Kaufmann.

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.

Sun. Jini connection technology. http://www.sun.com/jini/.

Sycara, K.; Pannu, A.; Williamson, M.; Zeng, D.; and Decker, K. 1996. Distributed intelligent agents. *IEEE Expert* 36–46.

Sycara, K.; Lu, J.; ; and Klusch, M. 1998. Interoperability among heterogeneous software agents on the internet. Technical Report CMU-CS-92-131, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA. Technical report.