Efficient Agent Communication in Multi-agent Systems

Myeong-Wuk Jang, Amr Ahmed, and Gul Agha

Department of Computer Science
University of Illinois at Urbana-Champaign,
Urbana IL 61801, USA
{mjang,amrmomen,agha}@uiuc.edu

Abstract. In open multi-agent systems, agents are mobile and may leave or enter the system. This dynamicity results in two closely related agent communication problems, namely, efficient message passing and service agent discovery. This paper describes how these problems are addressed in the *Actor Architecture* (*AA*). Agents in AA obey the operational semantics of actors, and the architecture is designed to support large-scale open multi-agent systems. Efficient message passing is facilitated by the use of dynamic names: a part of the mobile agent name is a function of the platform that currently hosts the agent. To facilitate service agent discovery, middle agents support application agent-oriented matchmaking and brokering services. The middle agents may accept search objects to enable customization of searches; this reduces communication overhead in discovering service agents when the matching criteria are complex. The use of mobile search objects creates a security threat, as codes developed by different groups may be moved to the same middle agent. This threat is mitigated by restricting which operations a migrated object is allowed to perform. We describes an empirical evaluation of these ideas using a large scale multi-agent UAV (Unmanned Aerial Vehicle) simulation that was developed using AA.

1 Introduction

In open agent systems, new agents may be created and agents may move from one computer node to another. With the growth of computational power and network bandwidth, large-scale open agent systems are a promising technology to support coordinated computing. For example, agent mobility can facilitate efficient collaboration with agents on a particular node. A number of multi-agent systems, such as EMAF [3], JADE [4], InfoSleuth [16], and OAA [8], support open agent systems. However, before the vision of scalable open agent systems can be realized, two closely related problems must be addressed:

– *Message Passing Problem*: In mobile agent systems, efficiently sending messages to an agent is not simple because they move continuously from one agent platform to another. For example, the *original agent platform* on which an agent is created should manage the location information about the agent.

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 236–253, 2005.

However, doing so not only increases the message passing overhead, but it slows down the agent's migration: before migrating, the agent's current host platform must inform the the original platform of the move and may wait for an acknowledgement before enabling the agent.

– *Service Agent Discovery Problem*: In an open agent system, the mail addresses or names of all agents are not globally known. Thus an agent may not have the addresses of other agents with whom it needs to communicate. To address this difficulty, middle agent services, such as *brokering* and *matchmaking* services [25], need to be supported. However, current middle agent systems suffer from two problems: *lack of expressiveness*–not all search queries can be expressed using the middle agent supported primitives; and *incomplete information*–a middle agent does not possess the necessary information to answer a user query.

We address the message passing problem for mobile agents in part by providing a richer name structure: the names of agents include information about their current location. When an agent moves, the location information in its name is updated by the platform that currently hosts the agent. When the new name is transmitted, the location information is used by other platforms to find the current location of that agent if it is the receiver of a message. We address the service agent discovery problem in large-scale open agent systems by allowing client agents to send search objects to be executed in the middle agent address space. By allowing agents to send their own search algorithms, this mitigates both the lack of expressiveness and incomplete information.

We have implemented these ideas in a Java-based agent system called the *Actor Architecture* (or *AA*). AA supports the *actor semantics* for agents: each agent is an autonomous object with a unique name (address), message passing between agents is asynchronous, new agents may be dynamically created, and agent names may be communicated [1]. AA has been designed with a modular, extensible, and application-independent structure. While AA is being used to develop tools to facilitate large-scale simulations, it may also be used for other large-scale open agent applications. The primary features of AA are: a light-weight implementation of agents, reduced communication overhead between agents, and improved expressiveness of middle agents.

This paper is organized as follows. Section 2 introduces the overall structure and functions of AA as well as the agent life cycle model in AA. Section 3 explains our solutions to reduce the message passing overhead for mobile agents in AA, while Section 4 shows how the search object of AA extends the basic middle agent model. Section 5 descries the experimental setting and presents an evaluation of our approaches. Related work is explained in Section 6, and finally, Section 7 concludes this paper with future research directions.

2 The Actor Architecture

AA provides a light-weight implementation of agents as active objects or actors [1]. Agents in AA are implemented as threads instead of processes. They

use object-based messages instead of string-based messages, and hence, they do not need to parse or interpret a given string message, and may use the type information of each field in a delivered message. The actor model provides the infrastructure for a variety of agent systems; actors are social and reactive, but they are *not* explicitly required to be "autonomous" in the sense of being proactive [28]. However, autonomous actors may be implemented in AA, and many of our experimental studies require proactive actors. Although the term agent has been used to mean proactive actors, for our purposes the distinction is not critical. In this paper, we use the terms 'agent' and 'actor' as synonyms.

The Actor Architecture consists of two main components:

- *AA platforms* which provide the system environment in which actors exist and interact with other actors. In order to execute actors, each computer node must have one AA platform. AA platforms provide actor state management, actor communication, actor migration, and middle agent services.
- *Actor library* which is a set of APIs that facilitate the development of agents on the AA platforms by providing the user with a high level abstraction of service primitives. At execution time, the actor library works as the interface between actors and their respective AA platforms.

An AA platform consists of eight components (see Fig. 1): Message Manager, Transport Manager, Transport Sender, Transport Receiver, Delayed Message Manager, Actor Manager, Actor Migration Manager, and ATSpace.

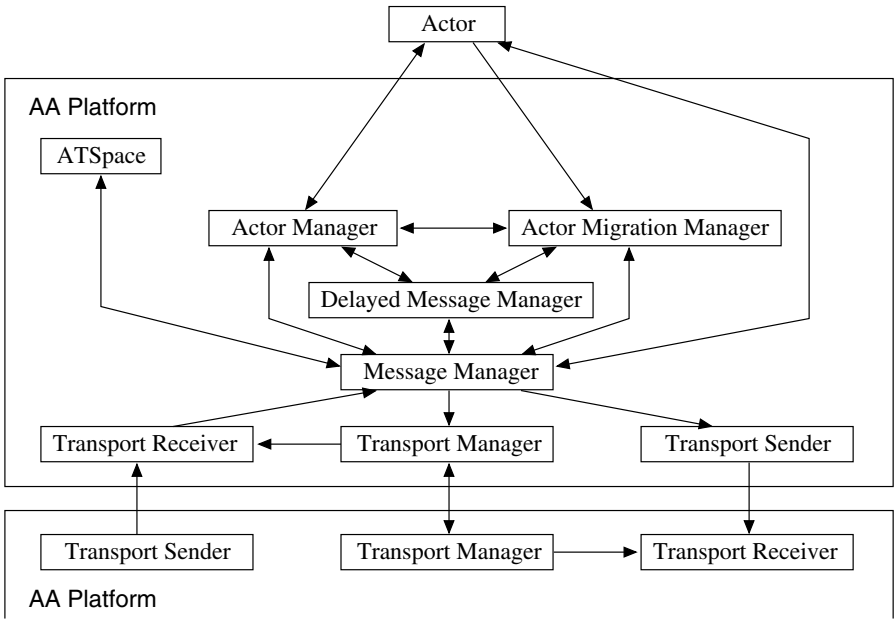


Fig. 1. Architecture of an AA Platform.

The *Message Manager* handles message passing between actors. Every message passes through at least one Message Manager. If the receiver actor of a message exists on the same AA platform, the Message Manager of that platform directly delivers the message to the receiver actor. However, if the receiver actor is not on the same AA platform, this Message Manager delivers the message to the Message Manager of the platform where the receiver currently resides, and finally that Message Manager delivers the message to the receiver actor. The *Transport Manager* maintains a public port for message passing between different AA platforms. When a sender actor sends a message to another actor on a different AA platform, the *Transport Sender* residing on the same platform as the sender receives the message from the Message Manager of that platform and delivers it to the *Transport Receiver* on the AA platform of the receiver. If there is no built-in connection between these two AA platforms, the Transport Sender contacts the Transport Manager of the AA platform of the receiver actor to open a connection so that the Transport Manager can create a Transport Receiver for the new connection. Finally, the Transport Receiver receives the message and delivers it to the Message Manager on the same platform.

The *Delayed Message Manager* temporarily holds messages for mobile actors while they are moving from one AA platform to another. The *Actor Manager* of an AA platform manages the state of actors that are currently executing as well as the locations of mobile actors created on this platform. The *Actor Migration Manager* manages actor migration.

The *ATSpace* provides middle agent services, such as matchmaking and brokering services. Unlike other system components, an ATSpace is implemented as an actor. Therefore, any actor may create an ATSpace, and hence, an AA platform may have more than one ATSpaces. The ATSpace created by an AA platform is called the *default ATSpace* of the platform, and all actors can obtain the names of default ATSpaces. Once an actor has the name of an ATSpace, the actor may send the ATSpace messages in order to use its services for finding other actors that match a given criteria.

In AA, actors are implemented as active objects and are executed as threads; actors on an AA platform are executed with that AA platform as part of one process. Each actor has one actor life cycle state on one AA platform at any time (see Fig. 2). When an actor exists on its original AA platform, its state information appears within only its original AA platform. However, the state of an actor migrated from its original AA platform appears both on its original AA platform and on its current AA platform. When an actor is ready to process a message its state becomes `Active` and stays so while the actor is processing the message. When an actor initiates migration, its state is changed to `Transit`. Once the migration ends and the actor restarts, its state becomes `Active` on the new AA platform and `Remote` on the original AA platform. Following a user request, an actor in the `Active` state may move to the `Suspended` state.

In contrast to other agent life cycle models (e.g. [10, 18]), the AA life cycle model uses the `Remote` state to indicate that an actor that was created on the current AA platform is working on another AA platform.

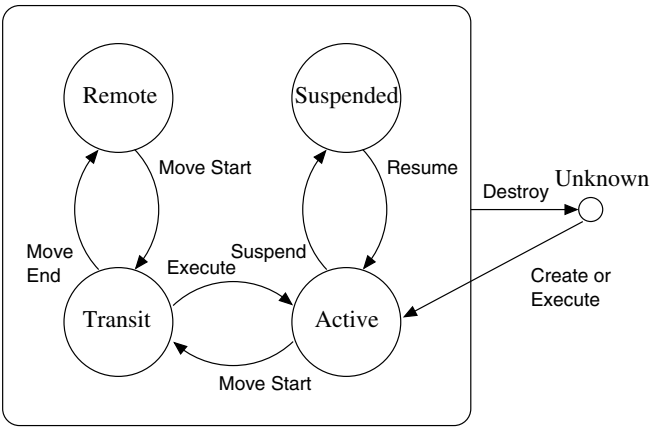


Fig. 2. Actor Life Cycle Model.

3 Optimized Message Delivery

We describe the message delivery mechanisms used to support inter-actor communications. Specifically, AA uses two approaches to reduce the communication overhead for mobile actors that are not on their original AA platforms: *location-based message passing* and *delayed message passing*.

3.1 Location-Based Message Passing

Before an actor can send messages to other actors, it should know the names of the intended receiver actors. In AA, each actor has its own unique name called *UAN* (*Universal Actor Name*). The UAN of an actor includes the *location information* and the *unique identification number* of the actor as follows:

```
uan://128.174.245.49:37
```

From the above name, we can infer that the actor exists on the host whose IP address is `128.174.245.49`, and that the actor is distinguished from other actors on the same platform with its unique identification number `37`.

When the *Message Manager* of a sender actor receives a message whose receiver actor has the above name, it checks whether the receiver actor exists on the same AA platform. If they are on the same AA platform, the Message Manager finds the receiver actor on this AA platform and directly delivers the message. Otherwise, the Message Manager of the sender actor delivers the message to the Message Manager of the receiver actor. In order to find the AA platform where the Message Manager of the receiver actor exists, the location information `128.174.245.49` in the UAN of the receiver actor is used. When the Message Manager on the AA platform with IP address `128.174.245.49` receives the message, it finds the receiver actor there and delivers the message.

The above actor naming and message delivery scheme works correctly when all actors are on their original AA platforms. However, because an actor may

migrate from one AA platform to another, we extend the basic behavior of the Message Manager with a *forwarding* service: when a Message Manager receives a message for an actor that has migrated, it delivers the message to the current AA platform of the mobile actor. To facilitate this service, each AA platform maintains the current locations of actors that were created on it, and updates the location information of actors that have come from other AA platforms on their original AA platforms.

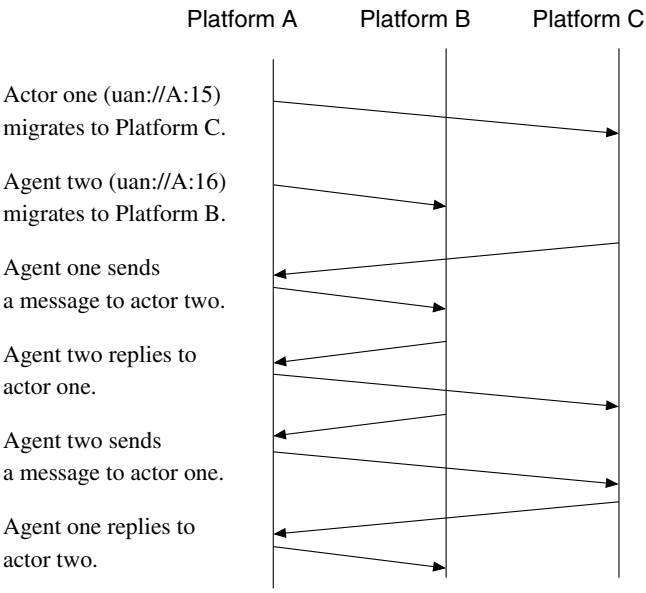
The problem with using only universal actor names for message delivery is that every message for a migrated actor still has to pass through the original AA platform in which the actor was created (Fig. 3.a). This kind of blind indirection may happen even in situations where the receiver actor is currently on an AA platform that is near the AA platform of the sender actor. Since message passing between actor platforms is relatively expensive, AA uses *Location-based Actor Name* (*LAN*) for mobile actors in order to generally eliminate the need for this kind of indirection. Specifically, the LAN of an actor consists of its current location and its UAN as follows:

```
lan://128.174.244.147//128.174.245.49:37
```

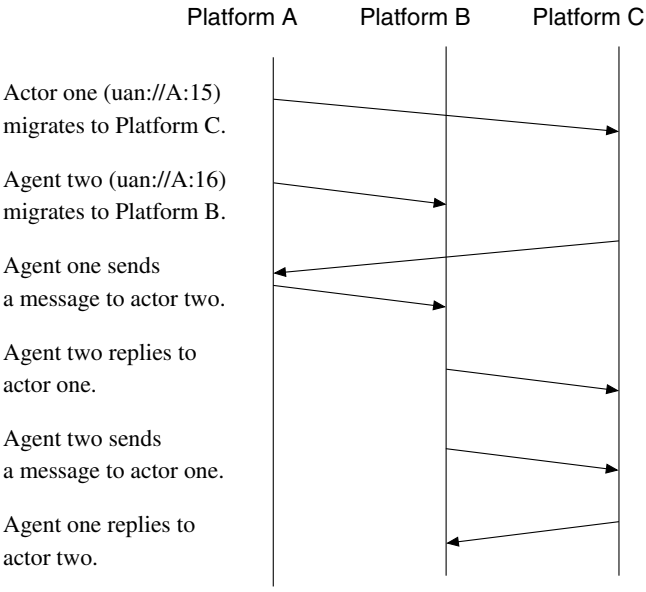
The current location of a mobile actor is set by an AA platform when the actor arrives on the AA platform. If the current location is the same as the location where an actor was created, the LAN of the actor does not have any special information beyond its UAN.

Under the location-based message passing scheme, when the Message Manager of a sender actor receives a message for a remote actor, it extracts the current location of the receiver actor from its LAN and delivers the message to the AA platform where the receiver actor exists. The rest of the procedure for message passing is similar to that in the UAN-based message passing scheme. Fig. 3.b shows how the location-based message passing scheme works. Actor *one* with `ual://C//A:15` sends its first message to actor *two* through the original AA platform of actor *two* because actor *one* does not know the location of actor *two*. This message includes the location information about actor *one* as the sender actor. Therefore, when actor *two* receives the message, it knows the location of actor *one*, and it can now directly send a message to actor *one*. Similarly, when actor *one* receives a message from actor *two*, it learns the location of actor *two*. Finally, the two actors can directly communicate with each other without mediation by their original AA platforms.

In order to use the LAN address scheme, the location information in a LAN should be recent. However, mobile actors may move repeatedly, and a sender actor may have old LANs of mobile actors. Thus a message for a mobile actor may be delivered to its *previous AA platform* from where the actor left. This problem is addressed by having the old AA platform deliver the message to the original AA platform where the actor was created; the original platform always manages the current addresses of its actors. When the receiver actor receives the message delivered through its original AA platform, the actor may send a null



a. UAN–based Message Passing



b. Location–based Message Passing

Fig. 3. Message Passing between Mobile Actors.

message with its LAN to update its location at the sender actor. Therefore, the sender actor can use the updated information for subsequent messages.

3.2 Delayed Message Passing

While a mobile actor is moving from one AA platform to another, the current AA platform of the actor is not well defined. In AA, because the location information of a mobile actor is updated after it finishes migration, its original AA platform thinks the actor still exists on its old AA platform during migration. Therefore, when the Message Manager of the original AA platform receives a message for a mobile actor, it sends the message to the Message Manager of the old AA platform thinking that it is still there. After the Message Manager of the old AA platform receives the message, it forwards the message to the Message Manager of the original AA platform. Thus, a message is continuously passed between these two AA platforms until the mobile actor updates the Actor Manager of its original AA platform with its new location.

In order to avoid unnecessary message thrashing, we use the *Delayed Message Manager* in each AA platform. After the actor starts its migration, the Actor Manager of the old AA platform changes its state to be `Transit`. From this moment, the Delayed Message Manager of this platform holds messages for this mobile actor until the actor reports that its migration has ended. After the mobile actor finishes its migration, its new AA platform sends its old AA platform and its original AA platform a message to inform them that the migration process has ended. When these two AA platforms receive this message, the original AA platform changes the state of the mobile actor from `Transit` to `Remote` while the old AA platform removes all information about the mobile actor, and the Delayed Message Manager of the old AA platform forwards the delayed messages to the Message Manager of the new AA platform of the actor.

4 Active Brokering Service

An ATSpace supports *active brokering services* by allowing agents to send their own search algorithms to be executed in the ATSpace address space [14]. We compare this service to current middle agent services.

Many middle agents are based on *attribute-based communication*. Service agents register themselves with the middle agent by sending a tuple whose attributes describe the service they advertise. To find the desired service agents, a client agent supplies a tuple template with constraints on attributes. The middle agent then tries to find service agents whose registered attributes match the supplied constraints. Systems vary more or less according to the types of constraints (primitives) they support. Typically, a middle agent provides exact matching or regular expression matching [2, 11, 17]. As we mentioned earlier, this solution suffers from a lack of expressiveness and incomplete information.

For example, consider a middle agent with information about seller agents. Each service agent (seller) advertises itself with the following attributes `<actor`

`name, seller city, product name, product price>`. A client agent with the following query is stuck:

Q1: *What are the **best two** (in terms of price) sellers that offer computers and whose locations are roughly **within 50 miles of me**?*

Considering the current tuple space technology, the operator "best two" is clearly not supported (expressiveness problem). Morever, the tuple space does not include distance information between cities (incomplete information problem). Faced with these difficulties, a user with this complex query Q1 has to transform it into a simpler one that is accepted by the middle agent which retrieves a superset of the data to be retrieved by Q1. In our example, a simpler query could be:

Q2: *Find all tuples about sellers that sell computers.*

An apparent disadvantage of the above approach is the movement of a large amount of data from the middle agent space to the buyer agent, especially if Q2 is semantically distant from Q1. In order to reduce communication overhead, ATSpace allows a sender agent to send its own search algorithm to find service agents, and the algorithm is executed in the ATSpace. In our example, the buyer agent would send a search object that would inspect tuples in the middle agent and select the best two sellers that satisfy the buyer criteria.

4.1 Security Issues

Although active brokering services mitigate the limitations of middle agents, such as brokers or matchmakers, they also introduce the following security problems in ATSpaces:

- *Data Integrity*: A search object may not modify tuples owned by other actors.
- *Denial of Service*: A search object may not consume too much processing time or space of an ATSpace, and a client actor may not repeatedly send search objects to overload an ATSpace.
- *Illegal Access*: A search object may not carry out unauthorized accesses or illegal operations.

We address the first problem by preventing the search object from modifying tuple data of other actors. This is done by supplying methods of the search object with a copy of the data in the ATSpace. However, when the number of tuples in the ATSpace is large, this solution requires extra memory and computation resources. Thus the ATSpace supports the option of delivering a shallow copy of the original tuples to the search object at the risk of data being changed by search objects as such scheme may compromise the data integrity.

To prevent malicious objects from exhausting the ATSpace computational resource, we deploy user-level thread scheduling as depicted in Fig. 4. When a search object arrives, the object is executed as a thread and its priority is

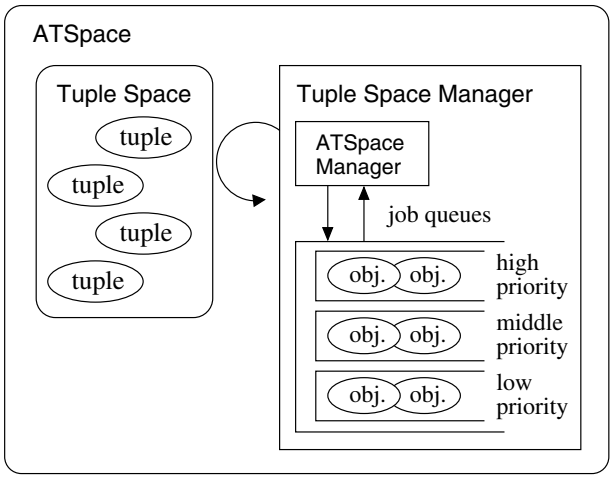


Fig. 4. Architecture of an ATSpace.

set to high. If the thread executes for a long time, its priority is continuously downgraded. Moreover, if the running time of a search object exceeds a certain limit, it may be destroyed by the tuple space manager.

To prevent unauthorized accesses, if the ATSpace is created with an access key, then this key must accompany every message sent from client actors. In this case, actors are allowed only to modify their own tuples. This prevents removal or modification of tuples by unauthorized actors.

5 Experiments and Evaluation

The AA platforms and actors have been implemented in Java language to support operating system independent actor mobility. The Actor Architecture is being used for large-scale UAV (Unmanned Aerial Vehicle) simulations. These simulations investigate the effects of different collaborative behaviors among a large number of micro UAVs during their surveillance missions over a large number of moving targets [15]. For our experiments, we have tested more than 1,000 actors on four computers: 500 micro UAVs, 500 targets, and other simulation purpose actors are executed. The following two sections evaluate our solutions.

5.1 Optimized Message Delivery

According to our experiments, the location-based message passing scheme in AA reduces the number of hops (over AA platforms) that a message for a mobile actor goes through. Since an agent has the location information about its collaborating agents, the agent can carry this information when it moves from one AA platform to another. With location-based message passing, the system is more fault-tolerant; since messages for a mobile actor need not pass through the original AA platform of the actor, the messages may be correctly delivered to the actor even when the actor's original AA platform is not working correctly.

Moreover, delayed message passing removes unnecessary message thrashing for moving agents. When delayed message passing is used, the old AA platform of a mobile actor needs to manage its state information until the actor finishes its migration, and the new platform of the mobile actor needs to report the migration state of the actor to its old AA platforms. In our experiments, this overhead is more than compensated; without delayed message passing the same message may get delivered seven or eight times between the original AA platform and the old AA platform while a mobile actor is moving. If a mobile actor takes more time for its migration, this number may be even greater.

5.2 Active Brokering Service

The performance benefit of ATSpace can be measured by comparing its active brokering services with the data retrieval services of a template-based general middle agent supporting the same service along four different dimensions: the number of messages, the total size of messages, the total size of memory space on the client and middle agent AA platforms, and the computation time for the whole operation. To analytically evaluate ATSpaces, we will use the scenario mentioned in section 4 where a service requesting agent has a complex query that is not supported by the template-based model.

First, with the template-based service, the number of messages is $n+2$ where n is the number of service agents that satisfy a complex query. This is because the service requesting agent has to first send a message to the middle agent to bring a superset of its final result. This costs two messages: a service request message to the middle agent ($\texttt{Service\_Request}_{\texttt{template}}$) that contains Q2 and a reply message that contains agent information satisfying Q2 ($\texttt{Service\_Reply}_{\texttt{template}}$). Finally, the service requesting agent sends n messages to the service agents that match its original criteria. With the active brokering service, the total number of messages is $n+1$. This is because the service requesting agent need not worry about the complexity of his query and only sends a service request message ($\texttt{Service\_Request}_{\texttt{ATSpace}}$) to the ATSpace. This message contains the code that represents its criteria along with the message that should be sent to the agents which satisfy these criteria. The last n messages have the same explanation as in the template-based service.

While the number of messages in the two approaches does not differ that much, the total size of these messages may have a huge difference. In both approaches, a set of n messages needs to be sent to the agents that satisfy the final matching criteria. Therefore, the question of whether or not active brokering services result in bandwidth saving depends on the relative size of the other messages. Specifically the difference in bandwidth consumption (DBC) between the template-based middle agent and the ATSpace is given by the following equation:

$$\begin{aligned} DBC = [&size(\texttt{Service\_Request}_{\texttt{template}}) - \\ &size(\texttt{Service\_Request}_{\texttt{ATSpace}})] + \\ &size(\texttt{Service\_Reply}_{\texttt{template}}) \end{aligned}$$

In general, since the service request message in active brokering services is larger as it has the search object, the first component is negative. Therefore, active brokering services will only result in a bandwidth saving if the increase in the size of its service request message is smaller than the size of the service reply message in the template-based service. This is likely to be true if the original query (Q1) is complex such that turning it into a simpler one (Q2) to retrieve a superset of the result would incur a great semantic loss and as such would retrieve much extra agent information from the middle agent.

Third, the two approaches put a conflicting requirement on the amount of space needed on both the client and middle agent machines. In the template-based approach the client agent needs to provide extra space to store the tuples returned by Q2. On the hand, the ATSpace needs to provide extra space to store copies of tuples given to search objects. However, a compromise can be made here as the creator of the ATSpace can choose to use the shallow copy of tuples.

Fourth, the difference in computation times of the whole operation in the two approaches depends on two factors: the time for sending messages and the time for evaluating queries on tuples. The tuples in the ATSpace are only inspected once by the search object sent by the service requesting agent. However, in the template-based middle agent, some tuples are inspected twice. First, in order to evaluate Q2, the middle agent needs to inspect all the tuples that it has. Second, these tuples that satisfy Q2 are sent back to the service requesting agent to inspect them again and retain only those tuples that satisfy Q1. If Q1 is complex then Q2 will be semantically distant from Q1, which in turns has two ramifications. First, the time to evaluate Q2 against all the tuples in the middle agent is small relative to the time needed to evaluate the search object over them. Second, most of the tuples on the middle agent would pass Q2 and be sent back to be re-evaluated by the service requesting agent. This reevaluation has nearly the same complexity as running the search object code. Thus we conclude that when the original query is complex and external communication cost is high, the active brokering service will result in time saving.

Apart from the above analytical evaluation, we have run a series of experiments on the UAV simulation to substantiate our claims. (Interested readers may refer to [13] for more details.) Fig. 5 demonstrates the saving in computational time of an ATSpace compared to a template-based middle agent that provides data retrieval services with the same semantic. Fig. 6 shows the wall clock time ratio of a template-based middle agent to an ATSpace. In these experiments, UAVs use either active brokering services or data retrieval services to find their neighboring UAVs. In both cases, the middle agent includes information about locations of UAVs and targets. In case of the active brokering service, UAVs send search objects to an ATSpace while the UAVs using data retrieval service send tuple templates. The simulation time for each run is around 35 minutes, and the wall clock time depends on the number of agents. When the number of agents is small, the difference between the two approaches is not significant. However, as the number of agents is increased, the difference becomes large.

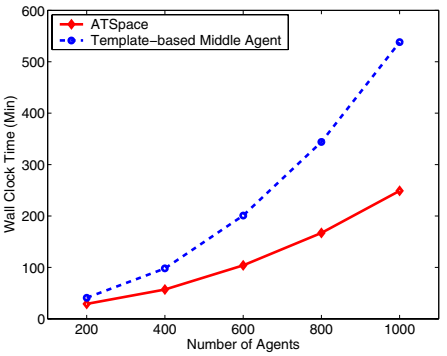


Fig. 5. Wall Clock Time (Min) for ATSpace and Template-based Middle Agent.

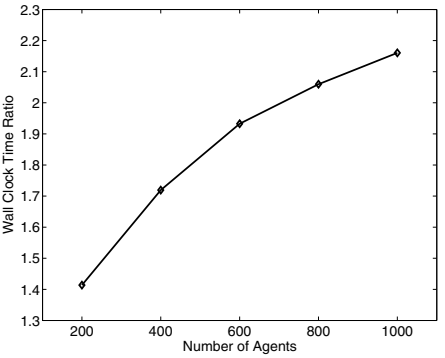


Fig. 6. Wall Clock Time Ratio of Template-based Middle Agent-to-ATSpace.

Fig. 7 depicts the number of messages required in both cases. The number of messages in the two approaches is quite similar but the difference is slightly increased according to the number of agents. Note that the messages increase almost linearly with the number of agents, and that the difference in the number of messages for a template-based middle agent and an ATSpace is small; it is in fact less than 0.01% in our simulations.

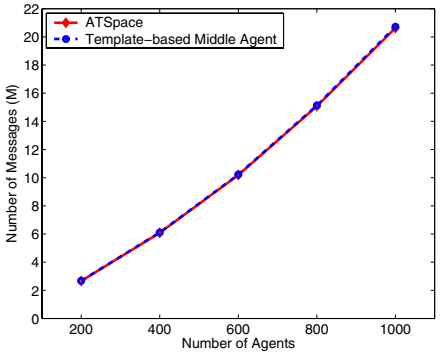


Fig. 7. The Number of Messages for ATSpace and Template-based Middle Agent.

Fig. 8 shows the total message size required in the two approaches, and Fig. 9 shows the total message size ratio. When the search queries are complex, the total message size in the ATSpace approach is much less than that in the template-based middle agent approach. In our UAV simulation, search queries are rather complex and require heavy mathematical calculations, and hence, the ATSpace approach results in a considerable bandwidth saving. It is also interesting to note the relationship between the whole operation time (as shown in Fig. 5) and the bandwidth saving (as shown in Fig. 8). This relationship supports our claim

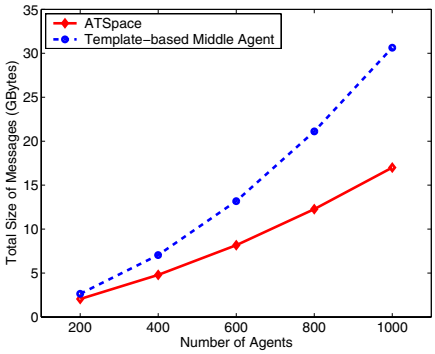


Fig. 8. Total Message Size (GBytes) for ATSpace and Template-based Middle Agent.

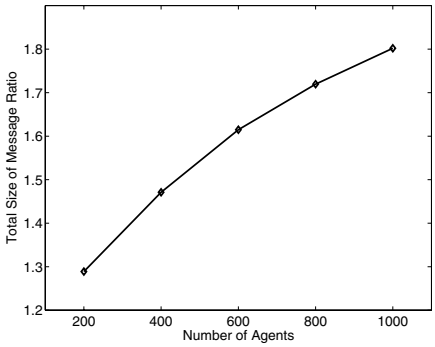


Fig. 9. Total Message Size Ratio for Template-based Middle Agent-to-ATSpace.

that the saving in the total operation time by the ATSpace is largely due to its superiority in efficiently utilizing the bandwidth.

6 Related Work

The basic mechanism of location-based message passing is similar to the message passing in *Mobile IP* [20], although its application domain is different. The original and current AA platforms of a mobile actor correspond to the home and foreign agents of a mobile client in Mobile IP, and the UAN and LAN of a mobile actor are similar to the home address and care-of address of a mobile client in Mobile IP. However, while the sender node in Mobile IP manages a binding cache to map home addresses to care-of addresses, the sender AA platform in AA does not have a mapping table. Another difference is that in mobile IP, the home agent communicates with the sender node to update the binding cache. However, in AA this update can be done by the agent itself when it sends a message that contains its address.

The LAN (Location-based Actor Name) may also be compared to UAL (Universal Actor Locator) in *SALSA* [27]. In SALSA, UAL represents the location of an actor. However, SALSA uses a middle agent called Universal Actor Naming Server to locate the receiver actor. SALSA's approach requires the receiver actor to register its location at a certain middle agent, and the middle agent must manage the mapping table.

The ATSpace approach, which is based on the tuple space model, is related to *Linda* [6]. In the Linda model, processes communicate with other processes through a shared common space called a blackboard or a tuple space without considering references or names of other processes [6, 21]. This approach was used in several agent frameworks, for example *EMAF* [3] and *OAA* [8]. However, these models support only primitive features for pattern-based communication among processes or agents. From the middle agent perspective, *Directory Facilitator* in

the *FIPA* platform [10], *ActorSpace* [2], and *Broker Agent* in *InfoSleuth* [16] are related to our research. However, these systems do not support customizable matching algorithms.

From the expressiveness perspective, some work has been done to extend the matching capability of the basic tuple space model. *Berlinda* [26] allows a concrete entry class to extend the matching function, and *TS* [12] uses policy closures in a Scheme-like language to customize the behavior of tuple spaces. However, these approaches do not allow the matching function to be changed during execution time. At the other hand, *OpenSpaces* [9] provides a mechanism to change matching polices during the execution time. OpenSpaces groups entries in its space into classes and allows each class to have its individual matching algorithm. A manager for each class of entries can change the matching algorithm during execution time. All agents that use entries under a given class are affected by any change to its matching algorithm. This is in contrast to the ATSpace where each agent can supply its own matching algorithm without affecting other agents. Another difference between OpenSpaces and ATSpaces is that the former requires a registration step before putting the new matching algorithm into action, but ATSpace has no such requirement.

Object Space [22] allows distributed applications implemented in the C++ programming language to use a matching function in its template. This matching function is used to check whether an object tuple in the space is matched with the tuple template given in `rd` and `in` operators. However, in the ATSpace the client agent supplied search objects can have a global overview of the tuples stored in the shared space and hence can support global search behavior rather than the one tuple based matching behavior supported in Object Space. For example, using the ATSpace a client agent can find the best ten service agents according to its criteria whereas this behavior cannot be achieved in Object Space.

TuCSoN [19] and *MARS* [5] provide programmable coordination mechanisms for agents through Linda-like tuple spaces to extend the expressive power of tuple spaces. However, they differ in the way they approach the expressiveness problem; while TuCSoN and MARS use reactive tuples to extend the expressive power of tuple spaces, the ATSpace uses search objects to support search algorithms defined by client agents. A reactive tuple handles a certain type of tuples and affects various clients, whereas a search object handles various types of tuples and affects only its creator agent. Therefore, while TuCSoN and MARS extend the general search ability of middle agents, ATSpace supports application agent-oriented searching on middle agents.

Mobile Co-ordination [23] allows agents to move a set of multiple tuple space access primitives to a tuple space for fault tolerance. In *Jada* [7], one primitive may use multiple matching templates. In *ObjectPlaces* [24], dynamic objects are used to change their state whenever corresponding objecplace operations are being called. Although these approaches improve the searching ability of tuple spaces with a set of search templates or dynamic objects, ATSpace provides more flexibility to application agents with their own search code.

7 Conclusion and Future Work

In this papers we addressed two closely related agent communication issues: efficient message delivery and service agent discovery. Efficient message delivery has been addressed using two techniques. First, the agent naming scheme has been extended to include the location information of mobile agents. Second, messages whose destination agent is moving are postponed by the Delayed Message Manager until the agent finishes its migration. For efficient service agent discovery, we have addressed the ATSpace, Active Tuple Space. By allowing application agents to send their customized search algorithms to the ATSpace, application agents may efficiently find service agents. We have synthesized our solutions to the mobile agent addressing and service agent discovery problems in a multi-agent framework.

The long term goal of our research is to build an environment that allows for experimental study of various issues that pertains to message passing and service agent discovery in open multi-agent systems and provide a principled way of studying possible tensions that arise when trying to simultaneously optimize each service. Other future directions include the followings: for efficient message passing, we plan to investigate various trade-offs in using different message passing schemes for different situations. We also plan to extend the Delayed Message Manager to support mobile agents who are contiguously moving between nodes. For service agent discovery, we plan to elaborate on our solutions to the security issues introduced with active brokering services.

Acknowledgements

The authors would like to thank the anonymous reviewers and Naqeeb Abbasi for their helpful comments and suggestions. This research is sponsored by the Defense Advanced Research Projects Agency under contract number F30602-00-2-0586.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
2. G. Agha and C.J. Callsen. ActorSpaces: An Open Distributed Programming Paradigm. In *Proceedings of the 4th ACM Symposium on Principles and Practice of Parallel Programming*, pages 23–32, May 1993.
3. S. Baeg, S. Park, J. Choi, M. Jang, and Y. Lim. Cooperation in Multiagent Systems. In *Intelligent Computer Communications (ICC '95)*, pages 1–12, Cluj-Napoca, Romania, June 1995.
4. F. Bellifemine, A. Poggi, and G. Rimassa. JADE - A FIPA-compliant Agent Framework. In *Proceedings of Practical Application of Intelligent Agents and Multi-Agents (PAAM '99)*, pages 97–108, London, UK, April 1999.
5. G. Cabri, L. Leonardi, and F. Zambonelli. MARS: a Programmable Coordination Architecture for Mobile Agents. *IEEE Computing*, 4(4):26–35, 2000.

6. N. Carreiro and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.
7. P. Ciancarini and D. Rossi. Coordinating Java Agents over the WWW. *World Wide Web*, 1(2):87–99, 1998.
8. P.R. Cohen, A.J. Cheyer, M. Wang, and S. Baeg. An Open Agent Architecture. In *AAAI Spring Symposium*, pages 1–8, March 1994.
9. S. Ducasse, T. Hofmann, and O. Nierstrasz. OpenSpaces: An Object-Oriented Framework for Reconfigurable Coordination Spaces. In A. Porto and G.C. Roman, editors, *Coordination Languages and Models, LNCS 1906*, pages 1–19, Limassol, Cyprus, September 2000.
10. Foundation for Intelligent Physical Agents. *SC00023J: FIPA Agent Management Specification*, December 2002. http://www.fipa.org/specs/fipa00023/.
11. N. Jacobs and R. Shea. The Role of Java in InfoSleuth: Agent-based Exploitation of Heterogeneous Information Resources. In *Proceedings of Intranet-96 Java Developers Conference*, April 1996.
12. S. Jagannathan. Customization of First-Class Tuple-Spaces in a Higher-Order Language. In *Proceedings of the Conference on Parallel Architectures and Languages - Vol. 2, LNCS 506*, pages 254–276. Springer-Verlag, 1991.
13. M. Jang, A. Ahmed, and G. Agha. A Flexible Coordination Framework for Application-Oriented Matchmaking and Brokering Services. Technical Report UIUCDCS-R-2004-2430, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004.
14. M. Jang, A. Abdel Momen, and G. Agha. ATSpace: A Middle Agent to Support Application-Oriented Matchmaking and Brokering Services. In *IEEE/WIC/ACM IAT(Intelligent Agent Technology)-2004*, pages 393–396, Beijing, China, September 20-24 2004.
15. M. Jang, S. Reddy, P. Tosic, L. Chen, and G. Agha. An Actor-based Simulation for Studying UAV Coordination. In *15th European Simulation Symposium (ESS 2003)*, pages 593–601, Delft, The Netherlnds, October 26-29 2003.
16. R.J. Bayardo Jr., W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments. *ACM SIGMOD Record*, 26(2):195–206, June 1997.
17. D.L. Martin, H. Oohama, D. Moran, and A. Cheyer. Information Brokering in an Agent Architecture. In *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 467–489, London, April 1997.
18. D.G.A. Mobach, B.J. Overeinder, N.J.E. Wijngaards, and F.M.T. Brazier. Managing Agent Life Cycles in Open Distributed Systems. In *Proceedings of the 2003 ACM symposium on Applied Computing*, pages 61–65, Melbourne, Florida, 2003.
19. A. Omicini and F. Zambonelli. TuCSoN: a Coordination Model for Mobile Information Agents. In *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, Pisa, Italy, June 1998.
20. C.E. Perkins. Mobile IP. *IEEE Communications Magazine*, 35:84–99, May 1997.
21. K. Pfleger and B. Hayes-Roth. An Introduction to Blackboard-Style Systems Organization. Technical Report KSL-98-03, Stanford Knowledge Systems Laboratory, January 1998.
22. A. Polze. Using the Object Space: a Distributed Parallel make. In *Proceedings of the 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 234–239, Lisbon, September 1993.

23. A. Rowstron. Mobile Co-ordination: Providing Fault Tolerance in Tuple Space Based Co-ordination Languages. In *Proceedings of the Third International Conference on Coordination Languages and Models*, pages 196–210, 1999.
24. K. Schelfthout and T. Holvoet. ObjectPlaces: An Environment for Situated Multi-Agent Systems. In *Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3 (AAMAS'04)*, pages 1500–1501, New York City, New York, July 2004.
25. K. Sycara, K. Decker, and M. Williamson. Middle-Agents for the Internet. In *Proceedings of the 15th Joint Conference on Artificial Intelligences (IJCAI-97)*, pages 578–583, 1997.
26. R. Tolksdorf. Berlinda: An Object-oriented Platform for Implementing Coordination Language in Java. In *Proceedings of COORDINATION '97 (Coordination Languages and Models), LNCS 1282*, pages 430–433. Pringer-Verlag, 1997.
27. C.A. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIGPLAN Notices: OOPSLA 2001 Intriguing Technology Track*, 36(12):20–34, December 2001.
28. M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, Ltd, 2002.