A Flexible Coordination Framework for Application-Oriented Matchmaking and Brokering Services

Myeong-Wuk Jang, Amr M.E. Ahmed, Gul Agha
Open Systems Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign
{mjang, amrmomen, agha}@uiuc.edu

Abstract

An important problem for agents in open multiagent systems is how to find agents that match certain criteria. A number of middle agent services, such as matchmaking and brokering services, have been proposed to address this problem. However, the search capabilities of such services are relatively limited since the match criteria they use are relatively inflexible. We propose ATSpace, *a middle agent to support application-oriented matchmaking and brokering services. Application agents in ATSpace deliver their own search algorithms to a public tuple space which holds agent property data; the tuple space executes the search algorithms on this data. We show how the ATSpace model increases the dynamicity and flexibility of middle agent services. Unfortunately, the model also introduces security threats: the data and access control restrictions in ATSpace may be compromised, and system availability may be affected. We describe some mechanisms to mitigate these security threats.*

Keywords: Agent Coordination, Agent Interaction, Middle Agents, Brokering Services, Matchmaking Services.

1. Introduction

In multiagent systems, agents need to communicate with each other to accomplish their goals. An important problem in open multiagent systems is the *connection problem*: how to find agents that match given criteria [Dav83]. When agents are designed or owned by the same organization,

developers may be able to design agents which explicitly know the names of other agents that they need to communicate with. However in *open systems*, because different agents may dynamically enter or leave a system, it is generally not feasible to let agents know the names of all other agents that they need to communicate with at some point.

For solving the connection problem, Decker classifies middle agent services as either *matchmaking* (also called *Yellow Page*) services or *brokering* services [Dec96, Syc97]. Matchmaking services (e.g. Directory Facilitator in FIPA platforms [Fip02]) are passive services whose goal is to provide a client agent with a list of names of agents whose properties match its supplied criteria. The agent may later contact the matched agents to request services. On the other hand, brokering services (e.g. ActorSpace [Cal94]) are active services that directly deliver a message (or a request) to the relevant agents on their clients' behalf.

In both types of services, an agent advertises itself by sending a message which contains its name and a description of its characteristics to a *middle agent*. A middle agent may be implemented on top of a tuple space model such as Linda [Car89]; this involves imposing constraints on the format of the stored tuples and using Linda-supported primitives. Specifically, to implement matchmaking and brokering services on top of Linda, a tuple template may be used by the client agent to specify the matching criteria. However, the expressive power of a template is very limited; it consists of value constraints for its actual parameters and type constraints for its formal parameters. In order to overcome this limitation, Callsen's ActorSpace implementation used regular expressions in its search template [Agh93, Cal94]. Even though this implementation increased expressivity, its capability is still limited by the power of its regular expressions.

We propose *ATSpace*[1] (Active Tuple Spaces) to empower agents with the ability to provide arbitrary application-oriented search algorithms to a middle agent for execution on the tuple space. While ATSpace increases the dynamicity and flexibility of the tuple space model, it also introduces some security threats as codes developed by different groups with different interests are executed in the same space. We will discuss the implication of these threats and how they may be mitigated.

This paper is organized as follows. Section 2 explains the ATSpace architecture and introduces its primitives. Section 3 describes security threats occurred in ATSpace and addresses how to resolve them. Section 4 illustrates the power of the new primitives by describing experiments with using ATSpace on UAV (Unmanned Aerial Vehicle) simulations. Section 5 evaluates the

[1] We will use *ATSpace* to refer the model for a middle agent to support application-oriented service, while we use an *atSpace* to refer an instance of ATSpace.

performance of ATSpace and compares it with a general middle agent. Section 6 discusses related work, and finally, we conclude this paper with a summary of our research and future work.

2. ATSpace

2.1 A MOTIVATIVE EXAMPLE

We present a simple example to motivate the ATSpace model. In general, a tuple space user with a complex matching query is faced with the following two problems:

1. *Expressiveness problem* because a matching query cannot be expressed using the tuple space primitives.
2. *Incomplete information problem* because evaluating a matching query requires information which is not available for a tuple space manager.

For example, assume that a tuple space has information about seller agents and the prices of the products they sell; each tuple has the following attributes `(seller name, seller city, product name, product price)`. Buyer agents can access the tuple space in order to find seller agents that sell, for instance, computers or printers. Also, a buyer agent wants to execute the following query:

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

A generic tuple space may not support the request of this buyer agent because, firstly, it may not support the “best two” primitive (problem 1), and secondly, it may not have information about the distance between cities (problem 2). Faced with these difficulties the buyer agent with the query Q1 has to transform it to a tuple template style query (Q2) to be accepted by the general tuple space. This query Q2 will retrieve a superset of the data that should have been retrieved by Q1.

Q2: *Find all tuples about seller agents that sell computers.*

The buyer agent then evaluates its own search algorithm on the returned data to find tuples that

satisfy Q1. In our example, the buyer agent would first filter out seller agents whose locations are less than 50 miles from the location of its user, and then choose the best two sellers from the remaining ones. To select seller agents located within 50 miles, the buyer agent has a way of estimating roughly distances between cites. Finally, it should send these seller agents a message to start the negotiation process.

An apparent disadvantage of the above approach is the movement of large amount of data from the tuple space to the buyer agent. When the tuple space includes large amount of tuples related to computer sellers, the size of the message to be delivered is also large. In order to reduce communication overhead, ATSpace allows a client agent to send an object containing its own search algorithm, instead of a tuple template. In our example, the buyer agent would send mobile code that inspects tuples in the tuple space and selects the best two sellers that satisfy the buyer criteria; the mobile code also carries information about distances to the near cities.

In Figure 1, the seller agents with AN2 and AN3 names are selected by the search algorithm, and the atSpace agent delivers `sendComputerBrand` message to them as a brokering service. Finally, the seller agents send information about brand names of their computers to the buyer agent.

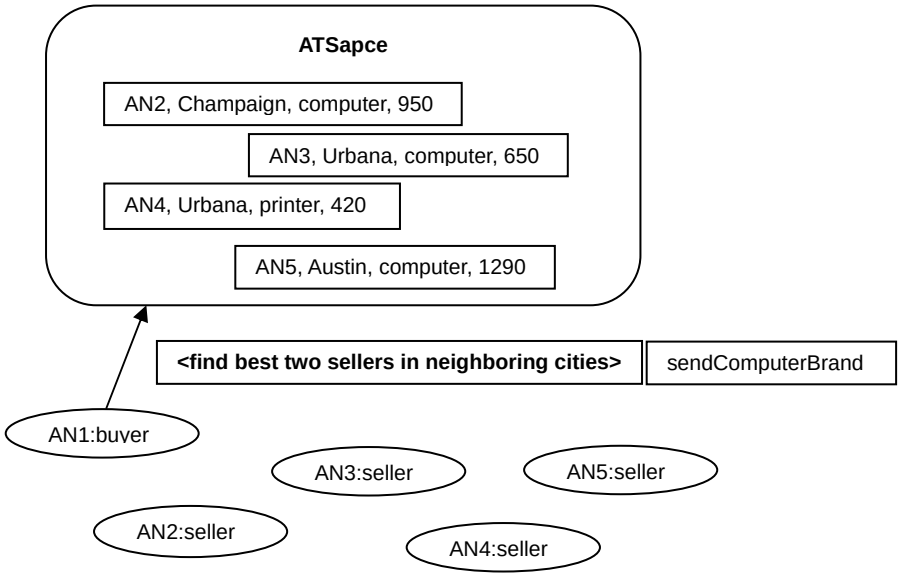

Figure 1: An Example of ATSpace

2.2 OVERALL ARCHITECTURE

ATSpace consists of three components: a tuple space, a message queue, and a tuple space manager (see Figure 2).

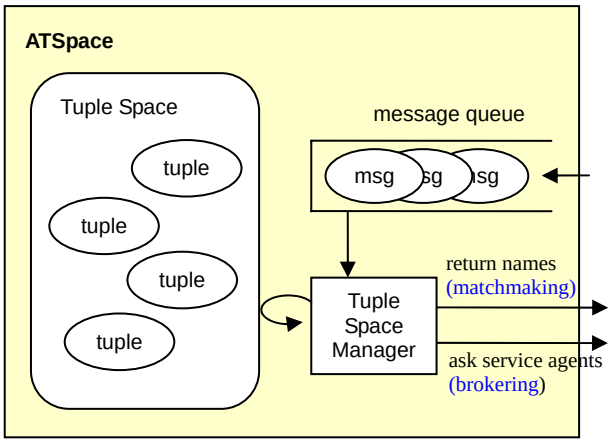

Figure 2: Basic Architecture of ATSpace

The tuple space is used as a shared pool for *agent tuples*, $\langle a, p_1, p_2, ..., p_n \rangle$, which consists of a name field, a, and a property part, $P = p_1, p_2, \ldots, p_n$ *where* $n \geq 1$; each tuple represents an agent whose name is given by the first field and whose characteristics are given by the subsequent fields. ATSpace enforces the rule that there cannot be more than one agent tuples whose agent names and property fields are identical. However, an agent may register itself with different properties (multiple tuples with the same name field), and different agents may register themselves with the same property fields (multiple tuples with the same property part).

$$\forall t_i, t_j : i \neq j \rightarrow [(t_i.a = t_j.a) \rightarrow (t_i.P \neq t_j.P) \,\&\&\, (t_i.P = t_j.P) \rightarrow (t_i.a \neq t_j.a)]$$

The message queue contains input messages that are received from other agents. Messages are classified into two types: *data input messages* and *service request messages*. A data input message includes a new agent tuple for insertion into the tuple space. A service request message includes either a tuple template or a mobile object. The template (or, alternately, the object) is used to search for agents with the appropriate agent tuples. A service request message may optionally contain another field, called the *service call message* field, to facilitate the brokering service. A *mobile*

object is an object that is provided by a service-requesting agent or client agent; such objects have pre-defined public methods, such as `find`. The `find` method is called by the tuple space manager with tuples in its atSpace as a parameter, and this method returns names of agents selected by the search algorithm specified in the mobile object.

The tuple space manager retrieves names of service agents whose properties match a tuple template or which are selected by a mobile object. In case of a matchmaking service, it returns the names to the client agent. In case of a brokering service, it forwards the service call message supplied by the client agent to the service agents.

2.3 OPERATION PRIMITIVES

General Tuple Space Primitives

The ATSpace model supports three basic primitives: `write`, `read`, and `take`. `write` is used to register an agent tuple into an atSpace, `read` is used to retrieve an agent tuple that matches a given criteria, and `take` is used to retrieve a matched agent tuple and remove it from the atSpace. When there are more than one agent tuples whose properties are matched with the given criteria, one of them is randomly selected by the agent tuple manager. When there is no a matched tuple, these primitives return immediately with an exception. In order to retrieve all agent tuples that match a given criteria, `readAll` or `takeAll` primitives should be used. The format[2] of these primitives is as follows:

```
void write(AgentName anATSpace, TupleData td);
AgentTuple read(AgentName anATSpace, TupleTemplate tt);
AgentTuple take(AgentName anATSpace, TupleTemplate tt);
AgentTuple[] readAll(AgentName anATSpace, TupleTemplate tt);
AgentTuple[] takeAll(AgentName anATSpace, TupleTemplate tt);
```

where `AgentName`, `TupleData`, `AgentTuple`, and `TupleTemplate` are data objects defined in ATSpace. A *data object* denotes an object that includes only methods to set and retrieve its member variables. When one of these primitives is called in an agent, the agent class handler creates a corresponding message and sends it to the atSpace specified as the first parameter, `anATSpace`. The `write` primitive causes a data input message while the others cause service request messages. Note that the `write` primitive does not include an agent tuple but a *tuple* that contains only the agent's

[2] The current ATSpace implementation is developed in the Java programming language, and hence, we use the Java syntax to express primitives.

properties. This is to avoid the case where an agent tries to register a property using another agent name to an atSpace. This tuple is then converted to an agent tuple with the name of the sender agent before the agent tuple is inserted to an atSpace.

In some applications, updating agent tuples happens very often. For such applications, *availability* and *integrity* are of great importance. Availability insures that at least one agent tuple exist at any time whereas integrity insures that old and new agent data do not exist simultaneously in an atSpace. Implementing the update request using two tuple space primitives, `take` and `write`, could result in one of these properties not being satisfied. If `update` is implemented using `take` followed by `write`, then availability is not met. On the other hand, if `update` is implemented using `write` followed by `take`, integrity is violated for a small amount of time. Therefore, ATSpace provides the `update` primitive to insure that `take` and `write` operations are performed as one atomic operation.

```
void update(AgentName anATSpace, TupleTemplate tt, TupleData td);
```

Matchmaking and Brokering Service Primitives

In addition, ATSpace also provides primitives for middle agent services: `searchOne` and `searchAll` for matchmaking services, and `deliverOne` and `deliverAll` for brokering services. Primitives for matchmaking are as follows:

```
AgentName searchOne(AgentName anATSpace, TupleTemplate tt);
AgentName searchOne(AgentName anATSpace, MobileObject ao);
AgentName[] searchAll(AgentName anATSpace, TupleTemplate tt);
AgentName[] searchAll(AgentName anATSpace, MobileObject ao);
```

The `searchOne` primitive is used to retrieve the name of a service agent that satisfies a given criteria, whereas the `searchAll` primitive is used to retrieve all names of service agents that match a given property.

Primitives for brokering service are as follows:

```
void deliverOne(AgentName anATSpace, TupleTemplate tt, Message msg);
void deliverOne(AgentName anATSpace, MobileObject ao, Message msg);
void deliverAll(AgentName anATSpace, TupleTemplate tt, Message msg);
void deliverAll(AgentName anATSpace, MobileObject ao, Message msg);
```

The `deliverOne` primitive is used to forward a specified service call message `msg` to the service agent that matches the given criteria, whereas the `deliverAll` primitive is used to send this message to all such service agents.

Note that our matchmaking and brokering service primitives allow agents to use *mobile objects* to support application-oriented search algorithm. We call matchmaking or brokering services used with mobile objects *active matchmaking* or *brokering services*. `MobileObject` is an abstract class that defines the interface methods between a mobile object and an atSpace. One of these methods is `find`, which may be used to provide the search algorithm to an atSpace. The format of the `find` method is defined as follows:

```
AgentName[] find(final AgentTuple[] ataTuples);
```

Service Specific Request Primitive

One drawback of the previous brokering primitives (`deliverOne` and `deliverAll`) is that they cannot support service-specific call messages. In some situations, a client agent cannot supply an atSpace with a service call message to be delivered to a service agent beforehand because it needs to examine the service agent properties first. Another drawback of the `deliverAll` primitive is that it stipulates that the same message should be sent to all service agents that match the supplied criteria. In some situations a client agent needs to send different messages to each service agent, depending on the service agent's properties. A client agent with any of the above requirements can use neither brokering services with tuple templates nor active brokering services with mobile objects. Therefore, the agent has to use the `readAll` primitive to retrieve relevant agent tuples and then create appropriate service call messages to send service agents selected. However, this approach suffers from the same problems as a general tuple space does.

To address the above shortcomings, we introduce the `exec` primitives. This primitive allows a client agent to supply a mobile object to an atSpace; the supplied mobile object has to implement the `doAction` method. When the method is called by an atSpace with agent tuples, it examines the properties of agents using the client agent application logic, creates different service call messages according to the agent properties, and then returns a list of *agent messages* to the atSpace to deliver the service call messages to the selected agents. Note that each agent message consists of the name of a service agent as well as a service call message to be delivered to the service agent. The formats of `exec` primitive and the `doAction` method are as follows.

```
void exec(AgentName anATSpace, MobileObject ao);
AgentMessage[] doAction(AgentTuple[] ataTuples);
```

3. SECURITY ISSUES

By allowing a mobile object to be supplied by an application agent, ATSpace supports application-oriented matchmaking and brokering services, which increases the flexibility and dynamicity of the tuple space model. However, it also introduces new security threats; we address some of these security threats and describe some ways to mitigate them. There are three important types of security issues for ATSpace:

- **Data Integrity**: A mobile object may not modify tuples owned by other agents.
- **Denial of Service**: A mobile object may not consume two much processing time or space of an atSpace, and a client agent may not repeatedly send mobile objects, thus overloading an atSpace.
- **Illegal Access**: A mobile object may not carry out unauthorized access or illegal operations.

We address the data integrity problem by blocking attempts to modify tuples. When a mobile object is executed by a tuple space manager, the manager makes a deep copy of tuples and then sends the copy to the `find` or `doAction` method of the mobile object. Therefore, even when a malicious agent changes some tuples, the original tuples are not affected by the modification. However, when the number of tuples in a tuple space is very large, this solution requires extra memory and computational resources. For better performance, the creator of an atSpace may select the option of delivering to mobile objects a shallow copy of the original tuples instead of a deep copy, although this will violate the integrity of tuples if an agent tries to delete or change tuples. We are currently investigating under what conditions a use of a shallow copy may be sufficient.

To address denial of service by consuming all processor cycles, we deploy user-level thread scheduling. Figure 3 depicts the extended architecture of ATSpace. When a mobile object arrives, the object is executed as a thread, and its priority is set to high. If the thread executes for a long time, its priority is continually downgraded. Moreover, if the running time of a mobile object exceeds a certain limit, it may be destroyed by the Tuple Space Manager; in this case, a message is sent to the sender agent of the mobile object to inform it about the destruction of the object. To incorporate these restrictions, we have extended the architecture of ATSpace by implementing job queues--thus making their semantics similar to that of actors. Other denial of service issues are still our on-going research.

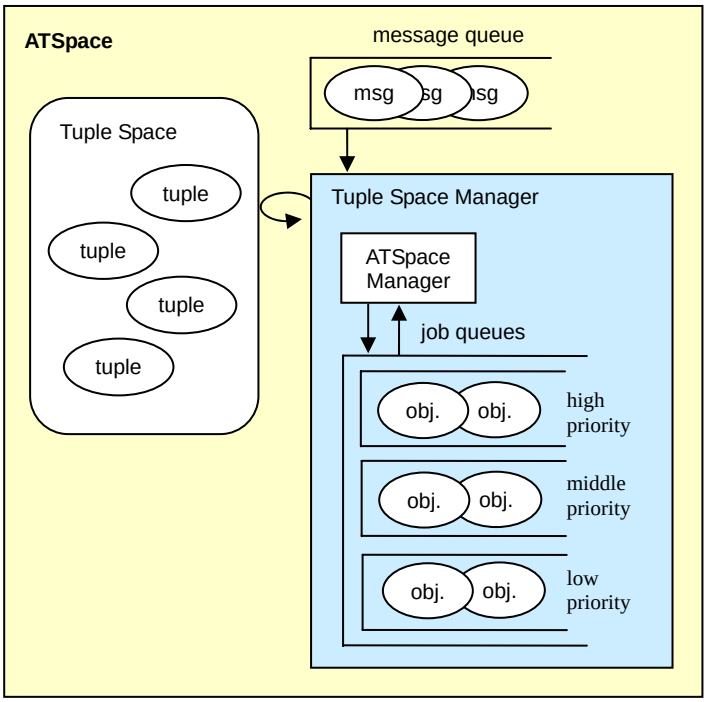

Figure 3: Extended Architecture of ATSpace

To prevent unauthorized access, an atSpace may be created with an *access key*; if an atSpace is created with an *access key*, then this key must accompany every message sent from service requester agents. Also, an atSpace may limit agents to modify only their own tuples.

4. Experiments

We have applied the ATSpace model in a UAV (Unmanned Aerial Vehicle) application which simulates the collaborative behavior of a set of UAVs in a surveillance mission [Jan03]. During the mission, a UAV needs to communicate with other neighboring UAVs within its local communication range (see Figure 4). We use the brokering primitives of ATSpace to accomplish this broadcasting behavior. Every UAV updates information about its location on an atSpace at every simulation step using the `update` primitive. When local broadcast communication is needed, the sender UAV (considered a client agent from the ATSpace perspective) uses the `deliverAll`

primitive and supplies as a parameter a mobile object[3] that contains its location and communication range. When this mobile object is executed in the atSpace, the `find` method is called by the tuple space manager to find relevant receiver agents. The `find` method computes distances between the sender UAV and other UAVs to find neighboring ones within the given communication range. When the tuple space manager receives names of service agents, neighboring UAVs in this example, from the mobile object, it delivers the service call message given by the client agent--the sender UAV in this example--to them.

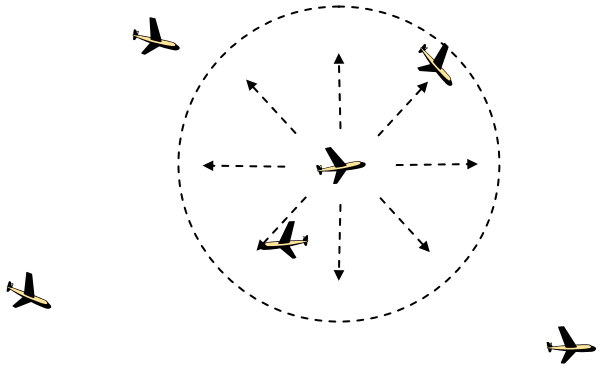

Figure 4: Simulation of Local Broadcast Communication

We also use ATSpace to simulate the behavior of UAV radar sensors. Each UAV should detect targets within its sensing radar range (see Figure 5). The *SensorSimulator*, which is the simulator component responsible for accomplishing this behavior, uses the `exec` primitive to implement this task. The mobile object[4] supplied with the `exec` primitive computes distances between UAVs and targets, and decides neighboring targets for each UAV. It then creates messages each of which consists of the name of its receiver UAV and a service call message to be sent its receiver UAV agent. This service call message is simply the environment model around this UAV (neighboring targets in our domain). Finally, the mobile object returns these set of messages to the tuple space manager which in turn sends them to respective agents.

[3] The code for this mobile object is in Appendix A.
[4] The code for this mobile object is in Appendix B.

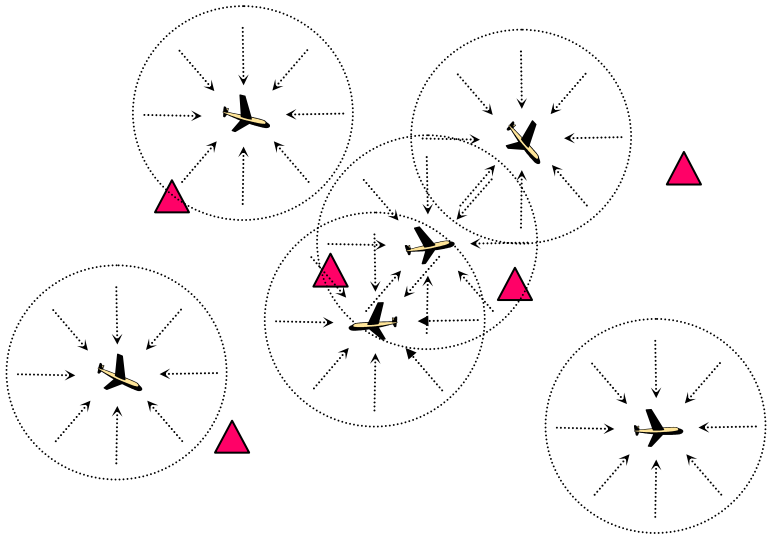

Figure 5: Simulation of Radar Sensor

5. Evaluation

The performance benefit of ATSpace can be measured by comparing its active brokering service with the data retrieval service of the template-based tuple space model along four different dimensions: the number of messages, the total size of messages, the total size of memory space on the clients' and middle agents' computers, and the time for the entire computation. To analytically evaluate ATSpace, we use the scenario described in section 2.1 where a service-requesting agent has a complex query that is not supported by the template-based model.

Let the number of service agents that satisfy this complex query be *n*. In the template-based tuple space model, the number of messages is *n* + 2. The details are as follows:

- `Service_Request`$_{\texttt{template}}$: a template-based service request message that includes Q2. A service-requesting agent sends this message to a tuple space to bring a superset of its final result.
- `Service_Reply`$_{\texttt{template}}$: a reply message that contains agent tuples satisfying Q2.
- *n* `Service_Call`: *n* service call messages to be delivered by the service-requesting agent to the agents that match its original criteria Q1.

In ATSpace, 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

($\texttt{Service\_Rquest}_{\texttt{ATSpace}}$) to an atSpace. This message contains the code that represents its criteria along with a service call message which should be sent the agents that satisfy the criteria. The last *n* messages have the same explanation as in the template based model except that the sender is the atSpace instead of the service-requesting agent.

While the difference in the number of messages delivered in the two approaches is comparatively small, the difference in the total size of these messages may be huge. Specifically, the difference in bandwidth consumption (*BD*: Bandwidth Difference) between the template-based model and the ATSpace one is given by the following equation:

$$BD = [\text{size}(\texttt{Service\_Request}_{\texttt{template}}) - \text{size}(\texttt{Service\_Request}_{\texttt{ATSpace}})\,] + \text{size}(\texttt{Service\_Reply}_{\texttt{template}})$$

In general the ATSpace service request message is larger, as it has the matching code, and thus the first component is negative. As such, ATSpace 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 approach. 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 a lot of the tuples from the template-based tuple space manager.

The amounts of the storage space used on the client agent's and middle agent's computers are similar in both cases. In the general tuple space, a copy of the tuples exists in the client agent, and an atSpace also requires a copy of the data for the mobile object to address the data integrity issue. However, if a creator of an atSpace opts to use a shallow copy of the data, the size of such a copy in the atSpace is much less than that of the copy in the client agent.

The difference in computation times of the entire operation in the two models depends on two factors: the time for sending messages and the time for evaluating queries on tuples. As we explained before, ATSpace will usually reduce the total size of messages so that the time for sending messages is in favor of ATSpace. Moreover, the tuples in the ATSpace are only inspected once by the mobile object sent by the service-requesting agent. However, in the template-based approach, some tuples are inspected twice: first, in order to evaluate Q2, the template-based tuple space manager needs to inspect all the tuples that it has, and second, the service-requesting agent inspects these tuples that satisfy Q2 to retain the tuples that also satisfy Q1. If Q1 is complex then Q2 may not filter tuples properly. Therefore, even though the time to evaluate Q2 against the entire tuples in the tuple space is smaller than the time needed to evaluate them by the mobile object, most

of the tuples on the tuple space manager may pass Q2 and be re-evaluated again by the service-requesting agent. This re-evaluation may have nearly the same complexity as running the mobile object code. Thus we can conclude that when the original query is complex and external communication cost is high, ATSpace will result in time saving.

Apart from the above analytical evaluation, we also evaluated the saving in computational time resulting from using the ATSpace in the UAV domain using the settings mentioned in section 4. Figure 6 shows the benefit of ATSpace compared to a general tuple space that provides the same semantic in the UAV simulation. In these experiments, UAVs use either active brokering service or data retrieval service 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 mobile objects to the middle agent while UAVs using data retrieval service send tuple templates. The simulation time for each run is around 40 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.

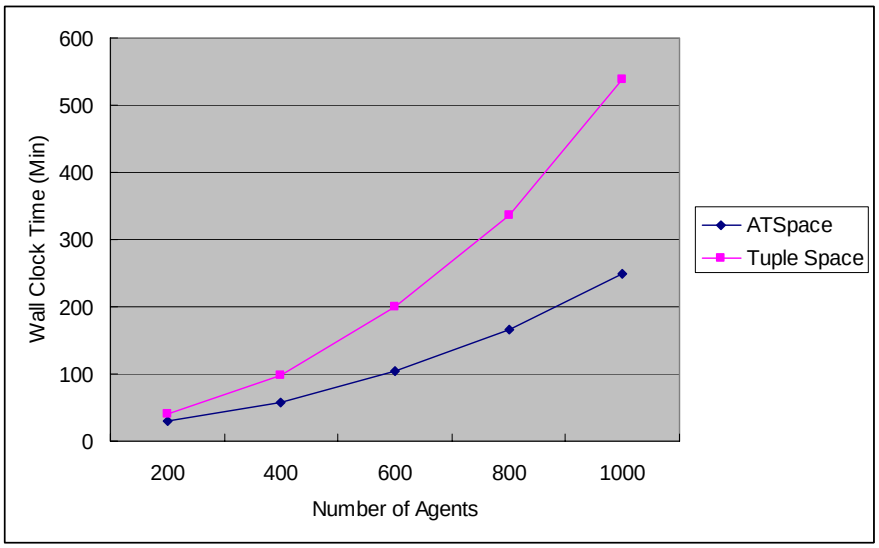

Figure 6: Wall Clock Time for ATSpace and Tuple Space

Figure 7 shows the number of messages, and Figure 8 shows the total size of messages in the two approaches, although the number of messages required is similar in both cases. However, a general tuple space requires more data movement than ATSpace, the shapes of these two lines in Figure 8 is similar to those in Figure 6. Therefore, we can hypothesize that there are strong

relationship between the total size of messages and the wall clock time of simulations.

Figure 7: The Number of Messages for ATSpace and Tuple Space

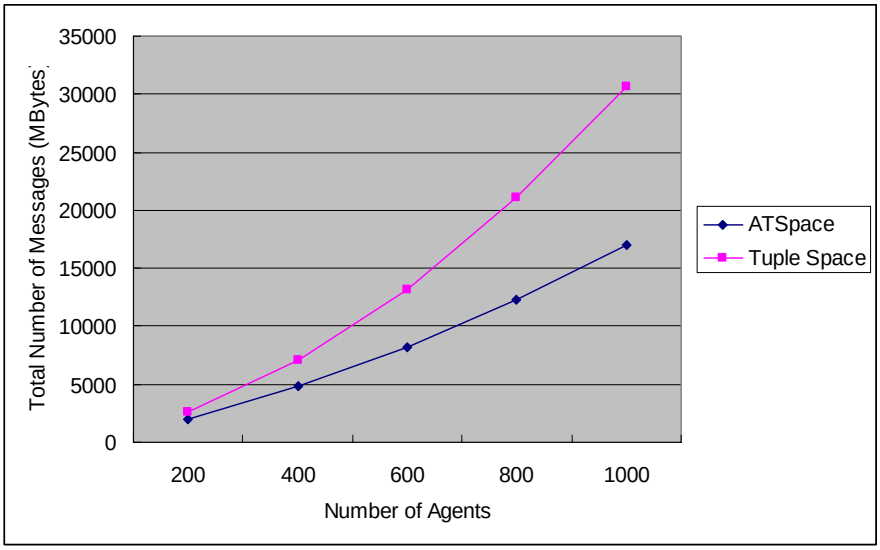

Figure 8: The Total Size of Messages for ATSpace and Tuple Space

6. Related Work

In this section we compare our ATSpace model with three related approaches: Other tuple space models, the Java Applet model, and finally mobile agents.

6.1 ATSpace Vs. Other Tuple Space Models

Our work is related to *Linda* [Car89, Gel85] and its variants, such as *JavaSpaces* and *TSpaces* [Leh99, Sun03]. In these models, 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 [Car89, Pfl98]. This approach was used in several agent frameworks, for example OAA and EMAF [Bae95, Mar97]. However, these models support only primitive features for anonymous communication among processes or agents.

From the middle agent perspective, *Directory Facilitator* in the *FIPA platform* and *Broker Agent* in *InfoSleuth* are related to our research [Fip02, Jac96]. However, these systems do not support customizable matching algorithm.

Some work has been done to extend the matching capability in the tuple space model. *Berlinda* allows a concrete entry class to extend the matching function [Tol97], and $\mathcal{TS}$ uses policy closures in a Scheme-like language to customize the behavior of tuple spaces [Jag91]. However, these approaches do not allow the matching function to be changed during execution. *OpenSpaces* provides a mechanism to change matching polices during execution [Duc00]. 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. All agents that use entries under a given class are affected by any change to its matching algorithm. This is in contrast to ATSpace where each agent can supply its own matching algorithm without affecting other agents. Another difference between OpenSpaces and ATSpace is that the former requires a registration step before putting a new matching algorithm into action. *Object Space* allows distributed applications implemented in the C++ programming language to use a matching function in its template [Pol93]. 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 ATSpace, the client agent supplied mobile objects can have a global overview of the tuples stored in the shared space, and hence, it can support global search behavior rather than one tuple based matching behavior supported in Object Space. For example, using 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 and *MARS* provide programmable coordination mechanisms for agents through Linda-

like tuple spaces to extend the expressive power of tuple spaces [Cab00, Omi98]. 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, ATSpace uses mobile objects to support search algorithms defined by client agents. A reactive tuple handles a certain type of tuples and affects various clients, whereas a mobile object handles various types of tuples and affects only its creator agent. Also, these approaches do not provide an execution environment for client agents. Therefore, these may be considered as orthogonal approaches and can be combined with our approach.

6.2 The ATSpace Model vs. the Applet Model

ATSpace allows the movement of a mobile object to the ATSpace manager, and thus it can be confused with the Applet model. However, a mobile object in ATSpace quite differs from a Java applet: a mobile object moves from a client computer to a server computer while a Java applet moves from a server computer to a client computer. Also, the migration of a mobile object is initiated by its owner agent on the client computer, but that of a Java applet is initiated by the request of a client Web browser. Another difference is that a mobile object receives a method call from an atSpace agent after its migration, but a Java applet receives parameters and does not receive any method call from processes on the same computer.

6.3 Mobile Objects vs. Mobile Agents

A mobile object in ATSpace may be considered as a mobile agent because it moves from a client computer to a server computer. However, the behavior of a mobile object differs from that of a mobile agent. First of all, the behavior of objects in general can be compared with that of agents as follows:

- An object is *passive* while an agent is *active*, i.e., a mobile object does not initiate activity.
- An object does not have the *autonomy* that an agent has: a mobile object executes its method whenever it is called, but a mobile agent may ignore a request received from another agent.
- An object does not have a *universal name* to communicate with other remote objects; therefore, a mobile object cannot access a method on the remote object, but a mobile agent can communicate with agents on other computers. However, note that some object-based middleware may provide such functionality: e.g., objects in CORBA or DCOM [Vin97, Tha99]

may refer remote objects.

- The method interface of an object is precisely predefined, and this interface is directly used by a calling object.[5] On the other hand, an agent may use a general communication channel to receive messages. Such messages require marshaling and unmarshaling, and have to be interpreted by receiver agents to activate the corresponding methods.
- While an object is executed as a part of a processor or a thread, an agent is executed as an independent entity; mobile objects may share references to data, but mobile agents do not.
- An object may use the reference passing in a method call, but an agent uses the value passing; when the size of parameters for a method call is large, passing the reference to local data is more efficient than passing a message, because the value passing requires a deep copy of data.

Besides the features of objects, we impose additional constraints on mobile objects in ATSpace:

- A mobile object can neither receive a message from an agent nor send a message to an agent.
- After a mobile object finishes its operation, the mobile object is destroyed by its current middle agent; a mobile object is used exactly once.
- A mobile object migrates only once; it is prevented from moving again.
- The identity of the creator of a mobile object is separated from the code of the mobile agent. Therefore, a middle agent cannot send a mobile object to another middle agent with the identity of the original creator of the object. Thus, even if the code of a mobile object is modified by a malicious server program, the object cannot adversely affect its creator. Moreover, since a mobile object cannot send a message to another agent, a mobile object is more secure than a mobile agent.[6] However, a mobile object raises the same security issues for the server side.

In summary, a mobile object loses some of the flexibility of a mobile agent, but this loss is compensated by increased computational efficiency and security.

7. Conclusion and Future Work

In this technical report we presented ATSpace, Active Tuple Space, which works as a common shared space to exchange data among agents, a middle agent to support application-oriented

[5] Methods of a Java object can be detected with the Java reflection mechanism. Therefore, the predefined interface is not necessary to activate a method of a Java object.
[6] [Gre98] describes security issues of mobile agents in detail.

brokering and matchmaking services, and an execution environment for mobile objects utilizing data on its space. Our experiments with UAV surveillance simulations show that the model may be effective in reducing coordination costs. We have described some security threats that arise when using mobile objects for agent coordination, along with some mechanisms we use to mitigate them. We are currently incorporating memory use restrictions into the architecture and considering mechanisms to address denial of service attacks that may be caused by flooding the network [Shi02]. We also plan to extent ATSpace to support multiple tuple spaces distributed across the Internet (a feature that some Linda-like tuple spaces [Omi98, Sny02] already support).

Acknowledgements

This research is sponsored by the Defense Advanced Research Projects Agency under contract number F30602-00-2-0586.

References

[Agh93] G. Agha and C.J. Callsen, "ActorSpaces: An Open Distributed Programming Paradigm," *Proceedings of the 4th ACM Symposium on Principles & Practice of Parallel Programming*, pp. 23-32, May 1993.

[Bae95] S. Baeg, S. Park, J. Choi, M. Jang, and Y. Lim, "Cooperation in Multiagent Systems," *Intelligent Computer Communications* (*ICC '95*), Cluj-Napoca, Romania, pp. 1-12, June 1995.

[Cab00] G. Cabri, L. Leonardi, F. Zambonelli, "MARS: a Programmable Coordination Architecture for Mobile Agents," *IEEE Computing*, Vol. 4, No. 4, pp. 26-35, 2000.

[Cal94] C. Callsen and G. Agha, "Open Heterogeneous Computing in ActorSpace," *Journal of Parallel and Distributed Computing*, Vol. 21, No. 3, pp. 289-300, 1994.

[Car89] N. Carreiro, and D. Gelernter, "Linda in context," *Communications of the ACM*, Vol. 32, No. 4, pp. 444-458, 1989.

[Dav83] R. Davis and R.G. Smith, "Negotiation as a Metaphor for Distributed Problem Solving," *Artificial Intelligence*, Vol. 20, No. 1, pp. 63-109, January 1983.

[Dec96] K. Decker, M. Williamson, and K. Sycara, "Matchmaking and Brokering," *Proceedings of the Second International Conference on Multi-Agent Systems* (*ICMAS-96*), Kyoto, Japan, December, 1996.

[Duc00] S. Ducasse, T. Hofmann, and O. Nierstrasz, “OpenSpaces: An Object-Oriented Framework for Reconfigurable Coordination Spaces,” In A. Porto and G.C. Roman, (Eds.), *Coordination Language and Models, LNCS 1906*, Limassol, Cyprus, pp. 1-19, September 2000.

[Fip02] Foundation for Intelligent Physical Agents, *SC00023J: FIPA Agent Management Specification*, December 2002. http://www.fipa.org/specs/fipa00023/

[Gel85] D. Gelernter, “Generative Communication in Linda,” *ACM Transactions on Programming Language and Systems*, Vol. 7, No. 1, pp. 80-112, January 1985.

[Gre98] M.S. Greenberg, J.C. Byington, and D.G. Harper, “Mobile Agents and Security,” *IEEE Communications Magazine*, Vol. 36, No. 7, pp. 76-85, July 1998.

[Jac96] N. Jacobs and R. Shea, “The Role of Java in InfoSleuth: Agent-based Exploitation of Heterogeneous Information Resources,” *Proceedings of Intranet-96 Java Developers Conference*, April 1996.

[Jag91] S. Jagannathan, “Customization of First-Class Tuple-Spaces in a Higher-Order Language,” *Proceedings of the Conference on Parallel Architectures and Languages - Vol. 2, LNCS 506*, Springer-Verlag, pp. 254-276, 1991.

[Jan03] M. Jang, S. Reddy, P. Tosic, L. Chen, and G. Agha. “An Actor-based Simulation for Studying UAV Coordination,” *Proceedings of the 15th European Simulation Symposium* (*ESS 2003*), pp. 593-601, October 2003

[Leh99] T.J. Lehman, S.W. McLaughry, and P. Wyckoff, “TSpaces: The Next Wave,” *Proceedings of the 32nd Hawaii International Conference on System Sciences* (*HICSS-32*), January 1999.

[Mar97] D.L. Martin, H. Oohama, D. Moran, and A. Cheyer, “Information Brokering in an Agent Architecture,” *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, London, pp. 467-489, April 1997.

[Omi98] A. Omicini and F. Zambonelli, “TuCSoN: a Coordination Model for Mobile Information Agents,” *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, Pisa, Italy, June 1998.

[Pfl98] K. Pfleger and B. Hayes-Roth, *An Introduction to Blackboard-Style Systems Organization*, KSL Technical Report KSL-98-03, Stanford Knowledge Systems Laboratory, January 1998.

[Pol93] A. Polze, “Using the Object Space: a Distributed Parallel make,” *Proceedings of the 4th*

IEEE Workshop on Future Trends of Distributed Computing Systems, Lisbon, pp. 234-239, September 1993.

[Shi02] C. Shields, "What do we mean by Network Denial of Service?," *Proceedings of the 2002 IEEE Workshop on Information Assurance and Security*, United States Military Academy, West Point, NY, pp. 17-19, June 2002.

[Sny02] J. Snyder and R. Menezes, "Using Logical Operators as an Extended Coordination Mechanism in Linda," In F. Arbab and C. Talcott (Eds.), *Coordination 2002, LNCS 2315*, Springer-Verlag, Berlin Heidelberg, pp. 317-331, 2002.

[Sun03] Sun Microsystems, *JavaSpaces™ Service Specification*, Ver. 2.0, June 2003. http://java.sun.com/products/jini/specs

[Syc97] K. Sycara, K. Decker, and M. Williamson, "Middle-Agents for the Internet," *Proceedings of the 15th Joint Conference on Artificial Intelligences* (*IJCAI-97*), pp. 578-583, 1997.

[Tha99] T.L. Thai, Learning DCOM, O'Reilly & Associates, 1999.

[Tol97] R. Tolksdorf, "Berlinda: An Object-oriented Platform for Implementing Coordination Language in Java," *Proceedings of COORDINATION '97* (*Coordination Languages and Models*), *LNCS 1282*, Pringer-Verlag, pp. 430-433, 1997.

[Vin97] S. Vinoski, "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments," IEEE Communications, Vol. 14, No. 2, pp. 46-55, Feb 1997.

Appendix A: Mobile Object for Local Broadcast Communication

```
public class CommunicationMobileObject implements MobileObject
{
   protected final static double BROADCAST_RANGE = 50000.0;
               //  range for broadcast communication

   private Point m_poPosition;
               //  location of the current location of a UAV

   /**
    *  Creates a mobile object with the location of the caller UAV agent.
    */
   public CommunicationMobileObject(Point p_point)
   {
      m_poPosition = p_point;
   }

    /**
    *  Defines the 'find' method.
    */
   public AgentName[] find(AgentTuple[] p_ataTuples)
   {
      double dEWDistance = m_poPosition.getX();
      double dNSDistance = m_poPosition.getY();

      LinkedList llReceivers = new LinkedList();

      for (int i=0; i<p_ataTuples.length; i++) {
         if (p_ataTuples[i].sizeOfElements() == 1) {
            Object objItem = p_ataTuples[i].getElement(0);

            try {
               //
               //  check the type of a field of a tuple.
               //
               if ( (Class.forName("app.task.uav.Point").isInstance(objItem)) ) {
                  Point poObject = (Point) objItem;
                  double dDistance =
                     Math.sqrt( Math.pow((poObject.getX() - dEWDistance), 2.0) +
                     Math.pow((poObject.getY() - dNSDistance), 2.0) );

                  //
                  //  compute the distance between the caller UAV and another.
                  //
                  if ( dDistance <= BROADCAST_RANGE) {
                     llReceivers.add(p_ataTuples[i].getAgentName());
                  }
               }
            } catch (ClassNotFoundException e) {
               System.err.println(">> Investigator.search: " + e);
            }
         }
      }

      //
      //  return the names of neighboring UAV agents.
      //
      AgentName anaReceivers[] = new AgentName[llReceivers.size()];
      llReceivers.toArray(anaReceivers);
      return anaReceivers;
   }
}
```

Appendix B: Mobile Object for Sensors of UAVs.

```
public class SensorMobileObject implements MobileObject
{
   public final static double RADAR_SENSOR_RANGE = 25000.0;
                //  range for radar sensing

   private final static double RADAR_SENSOR_ALTITUDE = 2000.0;
                //  minimum altitude of a UAV to detect an object by a radar

   private ObjectInfo[] m_oiaNeighboringObject;

   /**
    * Defines the 'doAction' method.
    */
   public AgentMessage[] doAction(final AgentTuple[] p_ataAgentTuples)
   {
      LinkedList llMsgs = new LinkedList();

      //
      //  classify the tuples into UAVs or Targets.
      //
      LinkedList llUAVs = new LinkedList();
      LinkedList llTargets = new LinkedList();

      for (int i=0; i<p_ataAgentTuples.length; i++) {
         if (p_ataAgentTuples[i].sizeOfElements() == 1) {
            try {
               Object objItem = p_ataAgentTuples[i].getElement(0);

               //
               //  check the type of a field of a tuple.
               //
               if ( (Class.forName("app.task.uav.Point").isInstance(objItem))) {
                  //
                  //  if a UAV is lower than the predefined minimum altitude,
                  //     then ignore the UAV.
                  //
                  Point poUAV = (Point) objItem;
                  if (poUAV.getZ() >= RADAR_SENSOR_ALTITUDE) {
                     llUAVs.add(p_ataAgentTuples[i]);
                  }
               } else if (Class.forName("app.task.uav.ObjectInfo").isInstance(objItem)) {
                  ObjectInfo oiTarget = (ObjectInfo) objItem;
                  llTargets.add(oiTarget);
               }
            } catch (ClassNotFoundException e) {
               System.err.println(">>SensorMobileObject.doAction: " + e);
            }
         }
      }

      //
      //  change LinkedList-type data to Array-type data.
      //
      AgentTuple[] ataUAVs = new AgentTuple[llUAVs.size()];
      llUAVs.toArray(ataUAVs);

      ObjectInfo[] oiaTargets = new ObjectInfo[llTargets.size()];
      llTargets.toArray(oiaTargets);

      //
      //  compute horizontal distance and vertical distance among UAVs.
      //
      m_oiaNeighboringObject = new ObjectInfo[oiaTargets.length];
```

```
      for (int i=0; i<ataUAVs.length; i++) {
         //
         //  collect neighboring objects, such as targets.
         //
         int iNumNeighboringObjects = 0;

         Point pointUAV = (Point) ataUAVs[i].getElement(0);
         double dX = pointUAV.getX();
         double dY = pointUAV.getY();

         for (int j=0; j<oiaTargets.length; j++) {
            double dDistance =
               java.lang.Math.sqrt(
                  Math.pow(dX - oiaTargets[j].getEWDistance(), 2.0) +
                  Math.pow(dY - oiaTargets[j].getNSDistance(), 2.0) );

            if ( (i != j) &&
               (dDistance < RADAR_SENSOR_RANGE) ) {
               oiaTargets[j].setHDistance(dDistance);
               oiaTargets[j].setVDistance(0);

               m_oiaNeighboringObject[iNumNeighboringObjects++] = oiaTargets[j];
            }
         }

         //
         //  if there are more than one neighboring objects,
         //     then create a message to send information about them to the UAV.
         //
         if (iNumNeighboringObjects > 0) {
            ObjectInfo[] oiaObjectDetected = new ObjectInfo[iNumNeighboringObjects];

            System.arraycopy(m_oiaNeighboringObject, 0,
                             oiaObjectDetected, 0,
                             iNumNeighboringObjects);

            Object[] objaArgs = { oiaObjectDetected };

            llMsgs.add(createAgentMessage(ataUAVs[i].getAgentName(), "alarm", objaArgs));
         }
      }

      //
      //  return agent messages to ATSpace.
      //
      AgentMessage anaMsgs[] = new AgentName[llMsgs.size()];
      llReceivers.toArray(anaMsgs);
      return anaMsgs;
   }
}
```