ATSpace: A Middle Agent to Support Application Oriented Matchmaking and Brokering Services

Myeong-Wuk Jang, Amr Abdel Momen, Gul Agha
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
{*mjang, amrmomen, agha*}*@uiuc.edu*
http://osl.cs.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 model 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 a middle agent service. 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 address these security threats.*

1 Introduction

In multiagent systems, agents need to communicate with each other to accomplish their goals, and an important problem in open multiagent systems is the problem of finding other agents that match given criteria, called the *connection problem* [4]. 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 an agent may be implemented by different groups, it is not feasible to let agents know the names of all agents that they may at some point need to communicate with.

Decker classifies middle agent services as either *matchmaking* (also called *Yellow Page*) services or *brokering* services [5]. Matchmaking services (e.g. Directory Facilitator in FIPA platforms [7]) 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 [1]) are active services that directly deliver a message (or a request) to the relevant agents on their client's 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 *service manager*. A service manager may be implemented on top of a tuple space model such as Linda [3]; 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 [1]. 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 the tuple space manager 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.

[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.

2 ATSpace

2.1 A Motivating Example

We present a simple example to motivate the ATSpace model. Assume that a tuple space has information about seller agents (e.g., city and name) and the prices of the products they sell. A buyer agent wants to contact the two "best" seller agents who offer computers and whose location is within 50 miles of his city. A brokering service supplied by a generic tuple space implementation may not support the request of the buyer agent because, firstly, it may not support the "best two" primitive, and secondly, it may not store distance information between cites. The buyer agent is now opt to retrieve from the tuple space the complete tuples that are related to computer sellers, and then execute their own search algorithm on them. However, this approach entails the movement of large amount of data. In order to reduce communication overhead, ATSpace allows a sender agent to send its own search algorithm which may, for example, carry information about distances to the nearest cities.

2.2 Overall Architecture

ATSpace consists of three components: a tuple space, a message queue, and a tuple space manager (see Figure 1).

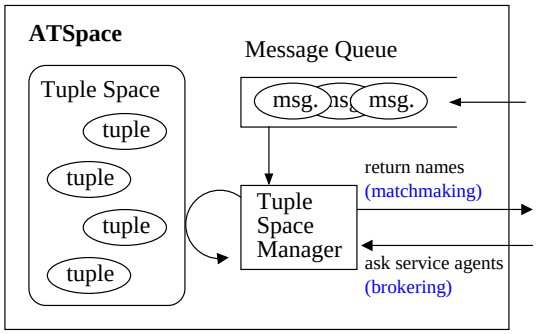

Figure 1. Basic Architecture of ATSpace

The tuple space is used as a shared pool for agent tuples, $\langle a, p_1, p_2, \ldots, p_n \rangle$, which consists of a name field, *a*, and a property part, $P = p_1, p_2, \cdots, p_n$ 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 tuple whose agent names and property fields are identical at any time. However, an agent may register itself with different properties, and different agents may register themselves with the same property fields.

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

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 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 request agent; such objects have a pre-defined public method called `find`. The `find` method is called by the tuple space manager with tuples in this atSpace as a parameter; it 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 the mobile object. In case of a matchmaking service, it returns the names to the client agent. In case of brokering service, it forwards the service call message supplied by the client agent to the agents.

2.3 Operation Primitives

The ATSpace model supports the three general tuple space primitives: `write`, `read`, and `take`. In addition, ATSpace also provides primitives for the matchmaking and brokering services. The `searchOne` primitive is used to retrieve the name of a service agent that satisfies a given property, whereas the `searchAll` primitive is used to retrieve the names of all service agents that satisfy the property. The `deliverOne` primitive is used to forward a specified message to a service agent that matches the property, whereas the `deliverAll` is used to send this message to all such service agents. These matchmaking and brokering service primitives allow client agents to use mobile objects to support application-oriented search algorithm as well as a tuple template. `MobileObject` is used as an abstract class that defines the interface methods between a mobile object and the ATSpace. One of these methods is `find`, which may be used to provide the search algorithm to an atSpace.

When a client agent requires information about specific properties of service agents stored in an atSpace to make service call messages, the above matchmaking or brokering service primitives cannot be used. The `exec` primitive within a mobile object provides this service. The supplied mobile object has to implement the `doAction` method which when called by the atSpace with agent tuples, examines the properties of agents using the client agent application logic, creates different service call messages according to the properties, and then returns a list of agent messages

to the atSpace for delivery to the selected agents.

3 Security Issues

There are three important security problems in ATSpace.

Data Integrity A mobile object may not modify tuples owned by other agents.

Denial of Service A mobile object may not consume too much processing time or space of an atSpace and a client agent may not send repeatedly mobile objects to overload an atSpace.

Illegal Access A mobile object may not carry out unauthorized accesses 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 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 the 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 to deliver a shallow copy of the original tuples to mobile objects instead of a deep copy, although this will violate the integrity of tuples if an agent tries to delete or change its tuple. 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 2 depicts the extension of the tuple space manager to achieve this. 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 its supplier informing it of the destruction. To incorporate these restrictions, we have extended the architecture of ATSpace by implementing job queues.

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

4 Evaluation

Figure 3 shows the advantage of ATSpace compared to a matchmaking service which provides the same semantic

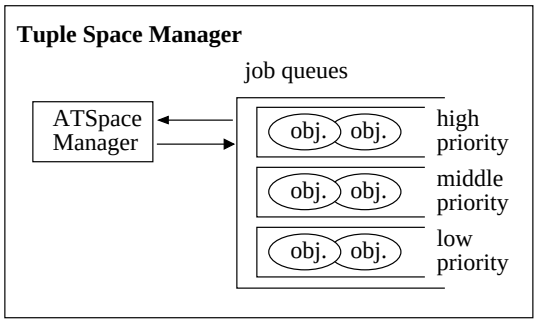

Figure 2. Extended Architecture of ATSpace

in UAV simulation (see [9] for details of this simulation). In these experiments, the UAVs use either an active brokering service or a matchmaking service to find their neighboring UAVs. In both cases, the middle agent includes information about the locations of UAVs. In case of the active brokering service, UAVs send mobile objects to the middle agent while UAVs using matchmaking service send tuple templates. The simulation time for each run is around 35 minutes.

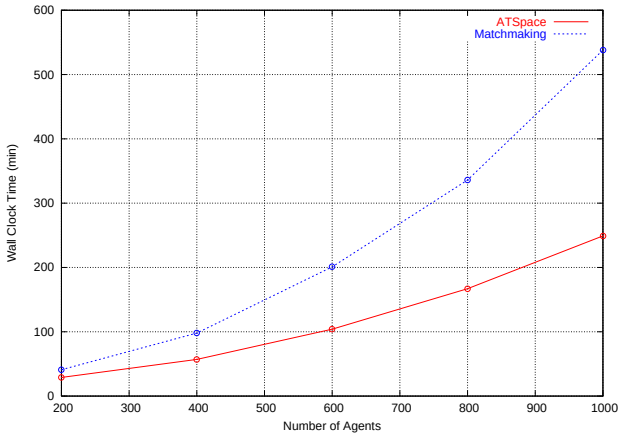

Figure 3. Wall Clock Time for ATSpace and Matchmaking Services

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 significant.

5 Related Work

Our work is related to *Linda* [3] and its variations, such as *JavaSpaces* and *TSpaces* [10, 13]. In these models, processes communicate with other processes through a shared common space called a *tuple space* without considering ref-

erences or names of other processes. From the middle agent perspective, *Directory Facilitator* in the *FIPA platform* and *Broker Agent* in *InfoSleuth* are related to our research [7, 8]. 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 [14]. However, this approach does not allow the matching function to be changed during execution time. *OpenSpaces* provides a mechanism to change matching polices during the execution time [6]. 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 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 the new matching algorithm into action.

TuCSoN and *MARS* provide programmable coordination mechanisms for agents through Linda-like tuple spaces to extend the expressive power of tuple spaces [2, 11]. 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 algorthms 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 envrionment for client agents. Therefore, these may be considered as orthogonal approaches and can be combined with our approach together.

6 Conclusion

ATSpace works as a common shared space to exchange data among agents, a middle agent to support matchmaking and brokering services, and an execution environment for mobile objects utilizing data on its space. Our experiments with a UAV surveillance task show that the model may be effective in reducing coordination costs. We 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 [12].

Acknowledgements

This research is sponsored by the DARPA ITO under contract number F30602-00-2-0586.

References

[1] G. Agha and C. Callsen. ActorSpaces: An Open Distributed Programming Paradigm. In *Proceedings of the 4th ACM Symposium on Principles & Practice of Parallel Programming*, pages 23–32, May 1993.

[2] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: a Programmable Coordination Architecture for Mobile Agents. *IEEE Computing*, 4(4):26–35, 2000.

[3] N. Carreiro and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[4] R. Davis and R. Smith. Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence*, 20(1):63–109, January 1983.

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

[6] S. Ducasse, T. Hofmann, and O. Nierstrasz. OpenSpaces: An Object-Oriented Framework for Reconfigurable Coordination Spaces. In A. Porto and G. Roman, editors, *Coordination Languages and Models, LNCS 1906*, pages 1–19, Limassol, Cyprus, September 2000.

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

[8] 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.

[9] M. Jang, A. A. Momen, 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.

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

[11] 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.

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

[13] Sun Microsystems. *JavaSpaces^{TM} Service Specification*, ver. 2.0, June 2003. http://java.sun.com/products/jini/specs.

[14] 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.