EFFICIENT COMMUNICATION AND COORDINATION FOR LARGE-SCALE MULTI-AGENT SYSTEMS

BY

MYEONG-WUK JANG

B.S., Korea University, 1990
M.S., Korea Advanced Institute of Science and Technology, 1992

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

The growth of the computational power of computers and the speed of networks has made large-scale multi-agent systems a promising technology. As the number of agents in a single application approaches thousands or millions, distributed computing has become a general paradigm in large-scale multi-agent systems to take the benefits of parallel computing. However, since these numerous agents are located on distributed computers and interact intensively with each other to achieve common goals, the agent communication cost significantly affects the performance of applications. Therefore, optimizing the agent communication cost on distributed systems could considerably reduce the runtime of multi-agent applications. Furthermore, because static multi-agent frameworks may not be suitable for all kinds of applications, and the communication patterns of agents may change during execution, multi-agent frameworks should adapt their services to support applications differently according to their dynamic characteristics.

This thesis proposes three adaptive services at the agent framework level to reduce the agent communication and coordination cost of large-scale multi-agent applications. First, communication locality-aware agent distribution aims at minimizing inter-node communication by collocating heavily communicating agents on the same platform and maintaining agent group-based load sharing. Second, application agent-oriented middle agent services attempt to optimize agent interaction through middle agents by executing application agent-supported search algorithms on the middle agent address space. Third, message passing for mobile agents aims at reducing the time of message delivery to mobile agents using location caches or by extending the agent address scheme with location information. With these services, we have achieved very impressive experimental results in large-scale UAV simulations including up to 10,000 agents. Also, we have provided a formal definition of our framework and services with operational semantics.

To my family, for their love, patience, and encouragement.

Acknowledgments

First of all, I would like to thank Professor Gul Agha for his intellectual guidance, for his generous advice, and for providing a research environment. During my studies, he has been my mentor as well as the supervisor of my research. He has taught me the attitude for living as well as researching.

I would also like to thank my committee members, Professor Geneva Belford, Professor Les Gasser, and Professor Indranil Gupta, for their invaluable comments and advice on my thesis research. I thank Professor Geneva Belford for her generous advice and comments, which improved the quality of my thesis. I thank Professor Les Gasser for many enjoyable discussions about multi-agent systems and for continually providing new perspectives and suggestions. I thank Professor Indranil Gupta for his incisive questions and thoughtful suggestions for my thesis. Also, I especially thank Professor William Pottenger for advising me when I had a difficult time. His endless enthusiasm about his research encouraged me to do my best in my own research.

I would like to thank past and present members of the OSL group, who made my time as a PhD student so enjoyable and memorable: Amr Ahmed, Mark Astley, Tom Brown, Po-Hao Chang, Liping Chen, Joshua Chia, MyungJoo Ham, Nadeem Jamali, WooYoung Kim, Nirman Kumar, YoungMin Kwon, Soham Mazumdar, Kirill Mechitov, Smitha Reddy, Shangping Ren, Koushik Sen, Sudarshan Srinivasan, Sameer Sundresh, Prasanna Thati, Predrag Tosic, Sandeep Uttamchandani, Abhay Vardhan, and Carlos Varela. I would also like to thank our departmental assistant Andrea Whitesell, who was always there and eager to help.

I would like to thank my parents, brothers, and sisters for their constant support throughout my studies. Because of their belief and encouragement, this research has been possible for me. Most of all, I would like to thank my wife, Young-Soon Lee, and daughter, Oo-Hyung Jang, whose love and support have been invaluable.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Problem Description

Parallel and distributed systems provide better execution environments for large-scale applications. By utilizing the benefits of parallel computation, they reduce the response or turnaround time of a given large-scale application. However, the development of parallel and distributed applications is more complex when designing the execution and interaction control of concurrent processes, for example, how to handle shared variables and deadlocks. This complexity sometimes considerably increases the development and testing time of parallel and distributed systems, especially when the systems are large scale.

The actor model [2, 5] provides a better programming environment for large-scale parallel and distributed systems. Actors manage their control flows as well as their states independently and perform operations by interacting with others using asynchronous message passing. Because they do not share variables, and their interaction is based on asynchronous message passing, application developers can avoid the fundamental problems of concurrent processes. Functions related to the coordination of concurrent processes (such as actor addressing, message passing, coordination protocol, etc.) are provided by middleware and are hidden to application actors (and actor developers). Hence, application developers can focus on the functions of their applications. Also, because the actor model supports the bottom-up development approach as well as the top-down development approach, a larger system can easily be built by merging existing smaller systems. Thus, the actor model contributes to the developers' productivity when developing parallel or distributed applications.

Many definitions of agents are based on the actor model, which takes an agent interaction and execution control point of view [45]. Asynchronous message passing in the actor model becomes the basis for the autonomy and social ability properties of agents. Multi-agent systems consist of applications and frameworks. Agent applications provide functions for certain specific purposes, and agent frameworks provide general distributed system environments for agent execution and interaction. To maximize the benefits of the concurrent and parallel execution of agents, load balancing or load sharing mechanisms may be used. They are especially effective with processing-oriented agent applications. However, for communication-oriented agent applications, the communication patterns of agents should be considered more seriously than balancing the workload. This is because load balancing may increase the agent communication cost by distributing strongly related agents on multiple computer nodes and thus decrease the overall performance of the system. This thesis focuses on mechanisms to be used for the efficient communication and coordination of large-scale communication-oriented multi-agent applications. We have analyzed the agent communication models generally used by agent frameworks and provided efficient mechanisms to support the communication models. The proposed mechanisms are based on the characteristics of multi-agent systems.

1.1.1 Two Models for Agent Communication

Agent communication models are generally categorized as *point-to-point communication* and *middle agent-based communication*. In the point-to-point communication model, a message is delivered from its sender agent to its receiver agent. Using the name or address[1] of the receiver agent, the message is eventually delivered to the receiver agent. If the receiver agent is mobile, the message may be passed through multiple computer nodes. In the middle agent-based communication model, matchmaking or brokering services are generally used. Middle agent services allow agents to communicate with others using the attributes of receiver agents instead of their names. By using the attributes of receiver agents instead of their names, middle agent services provide one-to-N communication and anonymous communication.

The detailed mechanisms of these agent communication models are hidden to application agents.

[1]The *address* of an agent is given by its agent platform, whereas the *name* of an agent is given by a programmer or user [98]. However, in this thesis, we use these two terms interchangeably.

Therefore, agent application developers need to understand the difference of these two communication models and know the usage of the communication primitives provided for agent communication. Mechanisms for message delivery between agents are part of agent middleware, and they are selected and developed by agent framework developers. The same model can be implemented differently, and different implementations perform differently. If we utilize the characteristics of large-scale multi-agent systems, the performance of agent coordination in large-scale multi-agent systems could be improved.

1.2 Efficient Communication and Coordination

1.2.1 Fundamental Ideas

According to the location of the receiver agent of a message, agent communication is classified into *inter-node communication* and *intra-node communication*. When the sender and receiver agents are located on the same computer node, message passing between these two agents is called intra-node communication. When they are located on two different computer nodes, messages between these two agents are delivered through networks, and this kind of message passing is called inter-node communication. Although the speed of the Internet has increased significantly, inter-node communication still takes much longer than intra-node communication. Therefore, if we can change inter-communication to intra-node communication, the communication cost between agents will be reduced.

In mobile agent systems, agents can move to find better system resources, and agents can be moved by agent frameworks for load sharing and fault tolerance. Thus, the locations of agents are changed, and sender agents cannot always know the locations of receiver agents. Although forwarding or agent locating mechanisms provide solutions to deliver messages to the current computer nodes of receiver agents, they increase the number of message hops to deliver messages. This is because these mechanisms do not deliver a message directly from the computer node of its sender agent to the computer node of its receiver agent: the message is delivered to the current node of the receiver agent through multiple computer nodes. If agent middleware can deliver a message directly to the computer node of the receiver agent, the communication cost for mobile agents will

be reduced.

Middle agent-based communication includes either brokers or matchmakers. Brokers forward messages given by sender agents to the computer nodes where receiver agents are located, whereas matchmakers only give the locations of receiver agents to the requesting agents that will send messages; in the brokering protocol, the final message is delivered by brokers, whereas in the matchmaking protocol it is delivered by sender agents. When we consider the number of message hops, brokering services are more efficient than matchmaking services; brokering services generally require two message hops, and matchmaking services require three message hops. When the messages of sender agents are small, brokering services are generally more efficient than matchmaking services. Therefore, if agents can use brokering services instead of matchmaking services, the communication cost for middle agent-based communication will be reduced.

1.2.2 Proposed Mechanisms

To improve the performance of agent communication, the characteristics of large-scale multi-agent systems should be utilized. For example, many agent frameworks provide agent mobility, and many agents are location-independent. Therefore, even though an agent framework moves an agent from one computer node to another, it does not affect the behaviors of agent applications. Also, agents use a given naming scheme without considering the internal details of agent names or addresses, and the changes in the names or internal mechanisms for sending messages do not influence the operations of agent applications. Using these characteristics of multi-agent systems, we can reduce the communication and coordination cost in large-scale applications.

First, agent frameworks can distribute agents to collocate a group of agents that have intensive intra-group communication. Because agents interact with others intensively to achieve common goals, and the patterns of agent interaction change continuously, dynamic agent distribution is more effective than static agent distribution. Therefore, we have presented dynamic agent distribution mechanisms. Also, because a centralized component for agent distribution becomes the bottleneck of the whole system and a point of failure easily, the mechanisms are developed as distributed algorithms. Our target applications include a large number of agents. Thus, handling agents individually may increase the cost of maintaining information about the communication of agents

and deciding the new distribution of agents. Therefore, the proposed dynamic agent distribution mechanisms use coarse-grained information about the patterns of agent communication instead of fine-grained information such as the individual communication patterns of agents. Also, the mechanisms form groups of agents for the analysis of communication patterns and maintain the agent groups for negotiation and distribution.

Second, to send messages directly from sender agents to receiver agents, agent frameworks can use information about the locations of agents. This information may be stored in the address (or name) of an agent, or agent frameworks may use caches to map the addresses of agents to their current locations. If agents move frequently, information about the locations of agents will easily become outdated, and messages may not be delivered directly to receiver agents. However, if agents do not move frequently and communicate with other agents intensively, this mechanism increases the possibility of sending messages directly from sender agents to receiver agents, and reduces the total number of message hops for agent interaction.

Thus far, brokering and matchmaking services should be selected by agents, because they use different primitives for these services. These two types of services have their own advantages: brokering services are more efficient, whereas matchmaking services are more flexible. Some complex search criteria that can be used with matchmaking services to find service agents cannot be used with brokering services. However, the expressiveness power of a selected primitive may give agents more chances to use brokering services. To improve this expressiveness power, agent frameworks may allow agents to send their own search algorithm to middle agents. The search algorithms that are moved from client agents to middle agents may reduce the amount of data to be moved from agents to middle agents by performing operations locally in the middle agents, thus reducing the communication cost.

1.2.3 Inter-relationship between Proposed Mechanisms

These three approaches to reduce the agent communication and coordination cost do not require any change in the basic agent communication models. However, they reduce the communication cost and thus improve the overall performance of large-scale multi-agent systems. Also, these approaches are interrelated. For example, dynamic agent distribution moves agents from their birth agent platforms

to others, thus increasing the communication overhead of mobile agents. Therefore, the efficiency of message passing for mobile agents influences the performance of dynamic agent distribution. Also, in distributed implementations of middle agent services, how to locate agent attribute data is an important issue, because these data change the communication patterns between middle agents and application agents. Therefore, in order to change inter-node communication to intra-node communication, agent attribute data can be moved. How to select either data movement or agent migration is still a complex research problem. In brokering services, brokers forward messages to receiver agents. If the receiver agent of a message is mobile, the message can be delivered to the current location of the receiver agent through other computer nodes. Thus, efficient message passing also positively influences the performance effects of brokering services.

1.3 Contributions

This research makes five main contributions:

1. *Actor-based Agent Framework for Large-Scale Multi-Agent Systems*

   We have developed an actor-based agent framework called the *Adaptive Actor Architecture* (*AAA*), and we have synthesized the proposed mechanisms in AAA. This actor-based agent framework has been tested with large-scale UAV simulations including 10,000 agents on a set of computers connected by a Giga-bit switch. With these simulations, we were able to investigate the effects of these new mechanisms concretely at the agent framework level in the context of large-scale multi-agent applications.

2. *Dynamic Agent Distribution to Reduce Inter-node Communication*

   Our communication locality-aware agent distribution mechanisms reduce the amount of inter-node communication and decrease the workload of overloaded agent platforms. These mechanisms are based on the communication dependencies between agents and agent platforms and the dependencies between agent groups and agent platforms, but not on the dependencies among individual agents. The granularity of negotiation is a group of agents instead of an individual agent, and the groups are dynamically formed according to the communication patterns of the agents. Moreover, these agent distribution mechanisms are developed as fully

distributed algorithms and are transparent to agent applications.

3. *Application Agent-oriented Middle Agent Services*

   A new active middle agent model allows the application agents to send their own search algorithms to a middle agent, and the delivered algorithms are executed on the middle agent address space on behalf of the application agents without affecting other agents. This approach allows the agents to use brokering services even though the search algorithms are more complex than any combination of the middle agent-supported primitives. Therefore, this model can effectively reduce an amount of communication between sender agents and a middle agent.

4. *Efficient Message Passing for Mobile Agents*

   The location-based message passing mechanisms reduce the number of hops required for message passing for mobile agents that do not exist on their birth agent platforms. These message passing mechanisms attempt to use recent information about the locations of other agents in their names or in the location cache. Because the location address in the name of an agent or in the location cache changes dynamically, this adaptive approach works more effectively and efficiently.

5. *Operational Semantics for Multi-Agent Systems*

   This research includes three approaches to reduced the communication cost. The correctness of these approaches and base frameworks can be verified by actor semantics. For this purpose, we have modified and extended the operational semantics proposed in the actor model [2, 5]. With this operational semantics, we can precisely understand and analyze the problems of our base agent frameworks based on this operational semantics. For example, with this operational semantics, we were able to find two message looping problems caused by asynchronous message passing in our agent framework and correct them.

1.4 Thesis Outline

This thesis is structured as follows. Chapter 2 explains our agent framework based on the actor model, with which the proposed mechanisms are integrated. In chapter 3, we describe two dynamic

agent distribution mechanisms: one mechanism aims at minimizing the agent communication cost by changing inter-node communication to intra-node communication, and the other mechanism attempts to prevent overloaded computer nodes from negatively affecting the overall performance. Chapter 4 presents application agent-oriented middle agent services to improve the performance of middle agent-based communication by extending the expressiveness power of middle agents: this middle agent model allows agents to send mobile search objects to a middle agent, and the objects perform search operations to find service agents on the middle agent address space. In chapter 5, we explain optimized message passing mechanisms that allow mobile agents to deliver messages directly to receiver agents: these mechanisms use the location information in the names of agents or the platform cache. Chapter 6 describes the operational semantics of our agent framework and analyzes message passing procedures in our agent framework using the semantics. Finally, in the last chapter, we discuss the limitations of the new services and include our future work.

Chapter 2

Adaptive Actor Architecture

2.1 Introduction

The proposed framework-level services described in this thesis are independent from specific agent frameworks. However, they need a framework to evaluate the proposed services. For our experiments, we developed an actor-based agent framework called the *Adaptive Actor Architecture* (*AAA*), and integrated the services in AAA. AAA is an extended version of the *Actor Architecture* (*AA*) [63, 65, 68]. Although agents on AA and AAA are autonomous, social, proactive, and reactive like agents on other agent frameworks, they also have the properties of actors; they can create other actors, communicate with other actors by asynchronous message passing, and change their own behaviors [2].

AAA provides an efficient execution environment for agents. In actor-based agent frameworks, agents are implemented as threads instead of processes. The communication between threads is more efficient than that between processes. Moreover, agents in many actor frameworks use object-based messages instead of string-based messages. Therefore, they neither need to convert object data to string messages nor parse incoming messages. Moreover, they can use type information about objects directly. However, with this approach, we cannot use heterogeneous agent platforms developed in different programming languages. Therefore, the communication paradigm should be extended to choose either an object-based message or a string-based message according to the characteristics of a destination agent platform.

2.2 Adaptive Actor Architecture

AAA provides development and execution environments for multi-agent systems. Agents in AAA follow the actor model [2], and the behaviors of agents are based on actor primitives. AAA consists of two main parts:

- *AAA platforms*, which provide the execution environment in which agents exist and interact with other agents. To execute agents, each computer node must have one AAA platform. AAA platforms provide agent state management, agent communication, agent migration, dynamic agent distribution, and middle agent services.
- *Actor library*, which includes a set of APIs (Application Programming Interfaces) that facilitate the development of agents on the AAA platforms by providing developers with a high-level abstraction of service primitives. At execution time, the actor library works as the interface between agents and their respective AAA platforms.

2.2.1 Architecture of an AAA Platform

An AAA platform consists of 10 components (see Figure 2.1): Message Manager, Transport Manager, Transport Sender, Transport Receiver, Delayed Message Manager, Actor Manager, Actor Migration Manager, Actor Allocation Manager, System Monitor, and ATSpace.

The *Message Manager* (MM) handles message passing between agents. Every message passes through at least one MM. If the receiver agent of a message exists on the same AAA platform as the sender agent, the MM of the platform delivers the message directly to the receiver agent. However, if the receiver agent is not on the same AAA platform, this MM delivers the message to the MM of the platform where the receiver currently resides, and the MM finally delivers the message to the receiver. The *Transport Manager* (TM) maintains a public port for message passing between different AAA platforms. When a sender agent sends a message to a receiver agent on a different AAA platform, the *Transport Sender* (TS) residing on the same platform as the sender receives the message from the MM of the sender agent and delivers it to the *Transport Receiver* (TR) on the AAA platform of the receiver. If there is no built-in connection between these two AAA platforms, the TS contacts the TM of the AAA platform of the receiver agent to open a connection so that

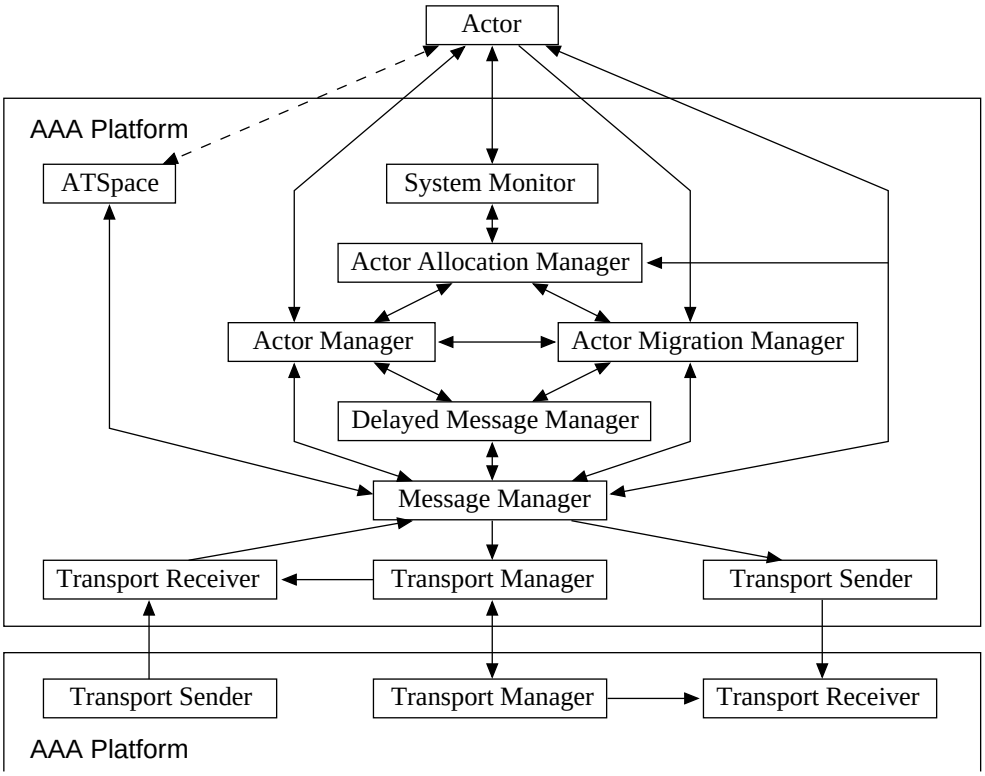


Figure 2.1: Architecture of an AAA Platform

the TM creates a TR for the new connection. Finally, the TR receives the message and delivers it to the MM on the same platform.

The *Delayed Message Manager* (DMM) temporarily holds messages for mobile agents while they are moving from their AAA platforms to other AAA platforms. The *Actor Manager* (AM) manages the states of the agents that are currently executing and the locations of the mobile agents created on the AAA platform. The *Actor Migration Manager* (AMM) manages agent migration.

The *System Monitor* (SM) periodically checks the workload of its computer node, and an *Actor Allocation Manager* (AAM) analyzes the communication pattern of agents. With the collected information, the AAM decides for either agents or agent groups to deliver them to other AAA platforms with the help of the Actor Migration Manager. The AAM negotiates with other AAMs to check the feasibility of migrations before starting agent migration.

The *ATSpace* provides middle agent services, such as matchmaking and brokering services. Unlike other system components, the ATSpace is implemented as an agent. Therefore, any agent can create an ATSpace, and an AAA platform may have more than one ATSpace. The ATSpace

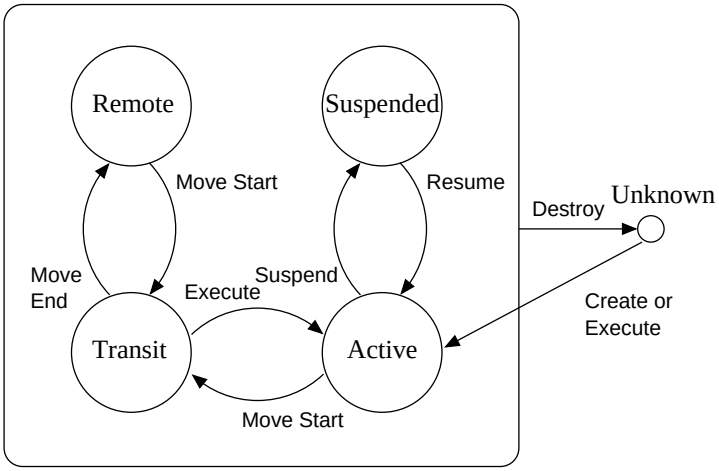


Figure 2.2: Agent Life Cycle Model

created by an AAA platform is called the *default ATSpace* of the platform, and all agents can obtain the agent names of default ATSpaces. Once an agent has the name of an ATSpace, the agent may send the ATSpace messages to use its services, and the messages are delivered through the Message Manager.

2.2.2 Agent Life Cycle in AAA

Each agent in AAA has one agent life cycle state at any time (see Figure 2.2). When an agent exists on its birth AAA platform, its state information appears only within its birth AAA platform. However, the state of an agent migrated from its birth AAA platform appears both on its birth AAA platform and on its current AAA platform. When an agent is ready to process a message, its state becomes `Active` and stays so while the agent is processing the message. When an agent initiates migration, its state is changed to `Transit`. Once the migration ends and the agent restarts, its state becomes `Active` on the new AAA platform and `Remote` on the birth AAA platform. Following a user request, an agent in the `Active` state may move to the `Suspended` state. In contrast to other agent life cycle models (e.g., [45, 81]), the life cycle model of AAA uses the `Remote` state to indicate that an agent that was created on the current AAA platform is working on another AAA platform.

2.2.3 Agent Primitives

In AAA, an agent uses three fundamental actor primitives. The semantics of these primitives is described in chapter 6.

- `create`: creates an agent with specified behavior.
- `send`: delivers a message to a specified receiver agent. This operator provides asynchronous communication between agents.
- `become`: creates an anonymous actor to carry out the rest of the current computation, alters the behavior of the agent executing this `become` operator, and frees the agent to accept another message.

In addition, an agent uses the following extended primitives. The semantics of primitive `migrate` is described in chapter 6.

- `createRemote`: creates an agent with specified behavior on a remote agent platform.
- `call`: sends a message synchronously to a specified receiver agent. This operator blocks the execution of the caller agent until the return value has arrived.
- `migrate`: moves the agent executing this operator from one agent platform to another.

2.2.4 Agent Communication Messages

Agent Communication Messages are categorized as *application agent-level messages* and *platform-level messages*. Application agent-level messages are created by application agents using actor primitives such as `send`, `call`, etc., and they are used for inter-agent communication. Platform-level messages are created by agent platforms, and they are used to inform the state of an agent to another agent platform. Although they have different purposes, both are delivered through the same agent components in AAA.

The attributes of agent communication messages are as follows:

- Sender Agent Name: the name of the sender actor.

- Receiver Agent Name: the name of the receiver actor.
- Message (Method) Name: a message name.
- Arguments: arguments for this message.
- Reply With: an identifier of this message.
- Reply To: an identifier that denotes what this message replies to.
- Return Request Flag: a flag that denotes whether this message requires a reply.
- Return Flag: a flag that denotes whether this message is a reply..
- Error Message Flag: a flag that denotes whether this message is an error handling message.

2.3 Related Work

2.3.1 Actors and Agents

The term *actor* was first introduced in 1971 as an active entity that triggers its activity according to pattern matching [57]. The actor model was published as a mathematical model for concurrent computation in 1973 by Carl Hewitt [58]. The actor model greatly influenced work on distributed and parallel computing, and it also provided the fundamental and common behavior of a variety of agents. Although agents are defined differently according to the research area or the researchers' background, actors have a commonly accepted definition. Actors are concurrent objects that manage their own execution flows as well as their states and that communicate with other actors using asynchronous message passing. An actor uses three fundamental operators: creating another actor, sending a message to another actor, and changing its behavior. Messages in the actor systems are eventually delivered, but the delivery time is not bounded. The actor model has been formulated by operational semantics and is used to check the equivalence of expressions [5].

With our agent point of view, agents are based on the concept of actors, and they may be considered as intelligent and mobile actors. As described in this thesis, actors can migrate from one actor platform to another, and our actor framework supports not only actor behaviors but also agent services such as middle agent services. Therefore, although the term agent has been used

with different meanings in other papers, as far as our work is concerned, this distinction [18, 116] is not critical. In this thesis, the terms "agent" and "actor" are used interchangeably.

2.3.2 Actor Frameworks

Several attempts have been made to provide actor runtime environments and actor programming languages, such as *Broadway* [101], *Actor Foundry* [11], *JMAS* [25], *SALSA* [111], *JavAct* [9], and *CyberOrgs* [62]. However, these frameworks focus on the fundamental services for actor systems; thus, they are not concerned with open multi-agent systems, middle agent services, or large-scale systems. Moreover, they do not seriously consider the efficiency of agent communication in the context of large-scale multi-agent systems.

Broadway [101] provides a runtime system for actors and a set of system actors as a C++ library. This actor framework uses meta-level architecture with policies that specify the interaction among agents. These policies are described by the high-level language Dil. This meta-level architecture allows meta-level objects to customize actor communication, and the meta-level objects can manipulate actor states using meta-level primitives. By separating actor interactions from actors, this actor framework can easily adapt to changing requirements and can reuse meta-level objects for other actors.

The Actor Foundry [11] is an actor framework that consists of a runtime system and a programming package. The runtime system works as middleware to support message passing among actors and to manage actors. The programming package helps programmers to develop actors more easily and works as a bridge between user-defined actors and the runtime system during execution. The runtime system and the programming package are developed in Java programming language. The Actor Foundry supports all actor operators except `become`. The main purpose of the `become` operator is to prepare an actor to process another message with a new behavior or state. However, since objects can manage their own states without an additional operator, and objects can automatically be prepared to process the next message after finishing the current message, this operator has been omitted. In addition to the basic actor behavior, The Actor Foundry supports actor mobility, synchronous message passing, and the `destroy` operator for garbage collection. The Actor Foundry was designed according to a modular component approach. Therefore, each part

may easily be replaced by a user-defined component, and new services can be attached without the modification of the runtime system.

JMAS (Java-based Mobile Actor System) [25] is a distributed computing framework for executing mobile actors. The actor execution and communication environment in JMAS is provided by the Distributed Run-Time Manager (D-RTM). JMAS supports not only three active primitives, such as `create`, `send`, and `become`, but also $\texttt{create}_{\texttt{remote}}$ and $\texttt{become}_{\texttt{remote}}$ primitives for mobile actors. Moreover, JMAS provides load balancing based on the CPU market strategy proposed in [29]. For an actor to send a message to another mobile actor, JMAS platforms forward the messages to the current location of the receiver actor.

SALSA (Simple Actor Language System and Application) [111] is an actor-based programming language used to develop actor applications. SALSA programs are converted into Java source code by a SALSA preprocessor, and finally, its Java bytecodes are executed on top of a Java Virtual Machine with the SALSA actor library. In addition to the fundamental operators of actors, SALSA supports actor migration, universal actor locators, and continuation-style control mechanisms. SALSA has shown good performance using a preprocessor and a runtime system, compared with the Actor Foundry.

JavAct [9] is an API for distributed and mobile actors implemented in Java programming language. JavAct uses Java RMI (Remote Message Invocation) for message passing. JavAct supports `createOn` and `becomeOn` primitives for mobile actors. JavAct uses reflection and Meta-Object Protocols (MOPs) to program concurrent object scheduling strategies and customize security [8, 22]. For an actor to send a message to another mobile actor, JavAct forwards the messages to the current location of the receiver actor.

CyberOrgs (Cyber Organizations) [62] has been proposed as a hierarchical model for resource control. With limited resources, the actors developed by different groups should compete with each other and should negotiate with actor platforms for shared resources. A cyberorg acts as the manager of a set of actors and resources. Actors in a cyberorg are executed within the boundary of resources purchased from another cyberorg with eCash. CyberOrgs have a hieratical structure such as a tree; a cyberorg may include another cyberorg as well as actors. The prototype of this model has been implemented using the Actor Foundry.

2.4 Discussion

This chapter explains our agent framework called the *Adaptive Actor Architecture* (*AAA*). AAA was implemented in Java programming language except for the System Monitor. The System Monitor uses system calls to check the status of the CPU and memory, and these functions are implemented in C. The system call functions and other components in the System Monitor are connected by Java *JNI* (*Java Native Interface*) [52]. To reduce the inter-node communication cost, our agent platforms use TCP instead of Java *RMI* (*Remote Method Invocation*) [55]. AAA has similar agent architecture and agent state management as FIPA [45]. Also, AAA provides middle agent services using ATSpace, like Directory Facilitator in FIPA. AAA uses Java object-based messages instead of the Agent Communication Language (ACL). Agents on AAA have the properties of actors, and they use actor primitives. Also, they are autonomous, social, proactive, and reactive, like agents on other agent frameworks. AAA is still an evolving agent framework, and thus, the format of agent communication messages and its platform-level services may be changed and may be extended in the future.

Chapter 3

Dynamic Agent Distribution

3.1 Introduction

In large-scale multi-agent systems, distributed computing may be necessary, especially when the application requires too many agents to execute all of them on a single computer node. Moreover, because distributed computing facilitates parallel execution, it may reduce the execution time of an application considerably. However, distributed computing incurs overhead (i.e., an inter-node communication cost), which does not occur in stand-alone computing. When two agents that communicate with each other run on two different computer nodes, they must use a communication channel through a network. Because agents perform their missions by continuously interacting with other agents to achieve common goals, their inter-node communication cost may significantly reduce the benefit of parallel computing. Although the communication speed of networks has grown considerably in the last decade, inter-node communication is still much more expensive than intra-node communication at the speed of message passing. Therefore, if we can reduce the amount of inter-node communication by changing it to intra-node communication, it will significantly improve the overall performance of the entire system.

To change inter-node communication to intra-node communication, agents can be distributed according to their communication patterns. Since the communication patterns of agents may change continuously, agent distribution should be dynamic; the best distribution of agents at the initial execution time of an application may be their worst distribution some time later. Therefore, we propose an agent distribution mechanism to allocate agents dynamically according to the changes in their communication localities. However, when agents are automatically distributed on multiple

computer nodes only according to their communication patterns, some agent platforms may be overloaded with numerous migrated agents. Therefore, our dynamic agent distribution mechanism for a communication locality is complemented by another agent distribution mechanism for load balancing.

For the scalability of this service, agent distribution mechanisms should not be controlled by a single server. In large-scale multi-agent systems, a centralized server should collect information about the communication patterns of numerous agents through networks and require a considerably longer processing time to analyze the patterns and determine agent distribution. This will reduce the effect of dynamic agent distribution mechanisms. Therefore, our proposed mechanisms have been developed as fully distributed algorithms.

Our dynamic agent distribution mechanisms improve the performance of large-scale multi-agent systems by changing inter-node communication to intra-node communication. However, these mechanisms also introduce the overhead for monitoring, decision making, and agent migration. Therefore, an important concern of this service in large-scale multi-agent systems is to minimize this overhead while simultaneously maximizing its benefit. By using detailed information, the mechanisms can make better decisions for agent distribution, but it takes longer and thus reduces the effect of dynamic agent distribution in large-scale multi-agent applications. Therefore, our proposed mechanisms are based on coarse-grained information for agent distribution instead of fine-grained information. In this chapter, we consider two agent distribution mechanisms to improve the performance of multi-agent systems with less overhead cost for this service.

3.2 Adaptive Agent Distribution Mechanisms

This section presents two mechanisms for dynamic agent distribution: a *communication localizing mechanism*, which collocates agents that communicate intensively with each other, and a *load sharing mechanism*, which moves agent groups from heavily loaded agent platforms to lightly loaded agent platforms.[1] Although the purpose of these two mechanisms is different, the mechanisms consist of similar process phases and share the same components. Note that these components are

[1]In our agent framework, each computer node has an *agent platform*, which provides a local agent execution environment within which agents operate and interact with other agents on the same or another agent platform.

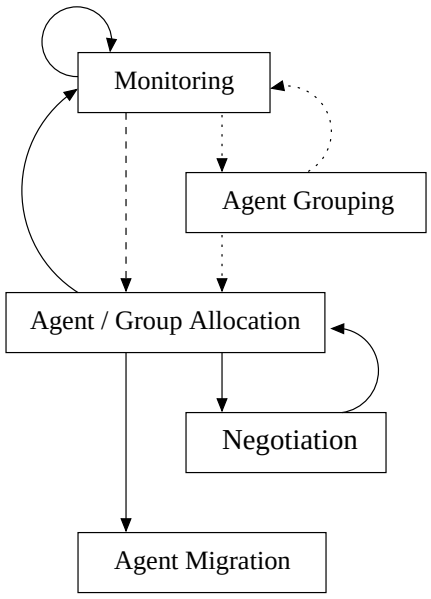


Figure 3.1: State Transition Diagram for Dynamic Agent Distribution. The solid lines indicate use by both mechanisms, the dashed line indicates use by only the communication localizing mechanism, and the dotted lines indicate use by only the load sharing mechanism.

replicated on all agent platforms. Figure 3.1 shows a state transition diagram for these two agent distribution mechanisms.

For dynamic agent distribution services, four primary system components are used in our agent framework:

- The *Message Manager* takes charge of message passing between agents.
- The *System Monitor* periodically checks the workload of its computer node.
- The *Actor Allocation Manager* is responsible for dynamic agent distribution.
- The *Actor Migration Manager* moves agents to other agent platforms.

A detailed explanation of these system components has been described in [64] and chapter 2.

3.2.1 Agent Distribution for the Communication Locality

The *communication localizing mechanism* handles dynamic changes in the communication patterns of agents. As time passes, the communication localities of agents change following changes in the agents' "social" behavior. By analyzing messages delivered between agents, agent platforms may

decide on which agent platform an agent should be located. Because an agent platform can neither estimate the future communication patterns of agents nor know whether agents on other platforms may migrate, the local decisions of an agent platform cannot be perfect. However, our experiments show that in the case of our particular applications, using the recent communication history to estimate the future communication patterns of agents works sufficiently well so that considerable performance improvement can be achieved. The communication localizing mechanism consists of four phases: *monitoring*, *agent distribution*, *negotiation*, and *agent migration* (see Figure 3.1).

Monitoring Phase

The Actor Allocation Manager (AAM) on each agent platform checks the communication patterns of agents on it with the assistance of its Message Manager. Specifically, the AAM uses information about both the sender agent and the agent platform of the receiver agent for each message.

The AAM periodically computes the communication dependencies $C_{ij}(t)$ at time t between agent i and agent platform j using

$$C_{ij}(t) = \alpha \left(\frac{M_{ij}(t)}{\sum_k M_{ik}(t)} \right) + (1 - \alpha) C_{ij}(t-1)$$

where $M_{ij}(t)$ is the number of messages sent from agent i to agent platform j during the t-th time step, and α is a coefficient representing the relative importance of recent information compared with old information. Coefficient α is used to ignore temporary intensive communication with agents on a certain agent platform.

To analyze the communication patterns of agents, the agents are classified into two types: *movable* and *unmovable*. Any agent can decide to move itself, even though it is either movable or unmovable. However, the AAM does not consider unmovable agents as candidates for dynamic agent distribution; an agent platform can migrate only movable agents. These types of agents are initially decided by programmers and may change by themselves during execution, but not by their agent platforms.

Agent Distribution Phase

After a certain number of repeated monitoring phases, the AAM computes the communication dependency ratio between an agent's current agent platform n and all other agent platforms. The communication dependency ratio R_{ij} between agent i and platform j is defined using

$$R_{ij} = \frac{C_{ij}}{C_{in}}, \quad j \neq n$$

When the maximum value of the communication dependency ratio of an agent is larger than a predefined threshold θ, the AAM assigns the agent to a *virtual agent group*, which represents a remote agent platform:

$$k = \arg\max_{j}(R_{ij}) \ \wedge \ (R_{ik} \ > \ \theta) \ \rightarrow \ a_i \in G_k$$

where a_i represents agent i, and G_k denotes virtual agent group k.

After the AAM checks all agents and assigns some of them to virtual agent groups, it starts the negotiation phase.

Negotiation Phase

Before the sender agent platform P_1 moves the agents assigned to a given virtual agent group to destination agent platform P_2, the AAM of P_1 communicates with the AAM of P_2 to check the feasibility of this migration. The AAM of agent platform P_2 checks the current memory usage, the recent CPU usage pattern, and the number of agents on this agent platform with the assistance of the System Monitor. If agent platform P_2 has sufficient system resources for the new agents, the AAM accepts the request. Otherwise, the AAM of P_2 responds with the number of agents that it can accept. In this case, P_1 moves only a subset of the virtual agent group.

Agent Migration Phase

Based on the response of a destination agent platform, the AAM of the sender agent platform initiates migration of all or part of the agents in the selected virtual agent group. When the destination agent platform has accepted part of the agents in the virtual agent group, the agents

to be moved are selected according to their communication dependency ratios. After the current operation of a selected agent ends, the Actor Migration Manager moves the agent to its destination agent platform. After the agent has migrated, it carries out its remaining operations.

3.2.2 Agent Distribution for Load Sharing

The agent distribution mechanism for the communication locality handles the dynamic change in the communication patterns of the agents, but this mechanism may overload some agent platforms once numerous agents are moved to these platforms. Before accepting the agents, the destination agent platform estimates its future workload to avoid becoming overloaded. However, this estimation is based on the current agents, and the workload of a system is continuously changing even though its local agents are the same. Thus, if the number of agents on an agent platform is close to the threshold of its overloaded workload state, the possibility of being overloaded is also relatively high. Therefore, we provided a load sharing mechanism to redistribute agents from heavily loaded agent platforms to lightly loaded agent platforms.

When an agent platform is overloaded, the System Monitor detects this condition and activates the agent redistribution procedure. Since agents had been assigned to their current agent platforms according to their most recent communication localities, choosing agents randomly for migration to lightly loaded agent platforms may result in cyclical migrations: the moved agents may still have a high communication rate with agents on their previous agent platform. Therefore, this mechanism intends to migrate a group of agents that communicate more intensively with each other. This load sharing mechanism consists of five phases: *monitoring*, *agent grouping*, *group distribution*, *negotiation*, and *agent migration* (see Figure 3.1).

Monitoring Phase

The System Monitor of each agent platform periodically checks the state of its agent platform; it obtains information about the current processor and memory usage of its computer node by accessing system call functions, and it maintains the number of agents on its agent platform. When the System Monitor decides that its agent platform is overloaded, it activates an agent distribution procedure. When the AAM is notified by the System Monitor, it starts monitoring

the local communication patterns of agents in order to partition them into *local agent groups*. For this purpose, at the beginning of this phrase, an agent that was not previously assigned to a local agent group is randomly assigned to one.

To check the local communication patterns of agents, the AAM uses information about the sender agent, the agent platform of the receiver agent, and the local agent group of the receiver agent of each message. (If the receiver agent is located on a different agent platform from that of the sender agent, information about the local agent group of the receiver agent is not used.) After a predetermined time interval, the AAM updates the communication dependencies between agents and local agent groups on the same agent platform. The $c_{ij}(t)$, the communication dependency between agent i and local agent group j at time step t, is defined as

$$c_{ij}(t) = \beta \left(\frac{m_{ij}(t)}{\sum_k m_{ik}(t)} \right) + (1 - \beta) c_{ij}(t-1)$$

where $m_{ij}(t)$ is the number of messages sent from agent i to agents in local agent group j during the t-th time step, and β is a coefficient used to decide the relative importance of recent information and old information. Note that in this case, $\sum_k m_{ik}(t)$ represents the number of messages sent by agent i to any agent in its current agent platform and is equal to M_{in}, where n is the index of the current agent platform.

Agent Grouping Phase

After a certain number of repeated monitoring phases, each agent i is reassigned to a local agent group whose index j^* is decided by

$$j^* = \arg\max_j (c_{ij}(t)) \;\; \rightarrow \;\; a_i \in A_{j^*}$$

where A_{j^*} denotes local agent group j^*.

Since the initial group assignment of agents may not be well organized, the monitoring and agent grouping phases are repeated several times. After each agent grouping phase, information about the local communication dependencies of agents is reset.

During the agent grouping phase, the number of local agent groups can be changed. When

two groups have much smaller populations than others, these two groups may be merged into one group. When one group has a much larger population than others, the agent group may be split into two groups. The minimum and maximum number of groups are predefined.

Group Distribution Phase

After a certain number of repeated monitoring and agent grouping phases, the AAM decides to move an agent group to another agent platform. The group selection is based on the communication dependencies between agent groups and agent platforms. The communication dependency D_{ij} between local agent group i and agent platform j is decided by summing the communication dependencies between all agents in the local agent group and the agent platform.

$$D_{ij}(t) = \sum_{k \in A_i} C_{kj}(t)$$

where A_i is the set of indices of all agents in local agent group i and $C_{kj}(t)$ is the communication dependency between agent k and agent platform j at time t.

Agent group i^*, which has the least dependency on the current agent platform, is selected using

$$i^* = \arg\max_i \left(\frac{\sum_{j, j \neq n} D_{ij}}{D_{in}} \right)$$

where n is the index of the current agent platform.

Destination agent platform j^* of selected agent group i is decided by the communication dependency between the agent group and agent platforms using

$$j^* = \arg\max_j \left(D_{ij} \right), \quad j \neq n$$

where n is the index of the current agent platform.

Negotiation Phase

Once the destination agent platform of a agent group is decided, the AAM of the sender agent platform communicates with the corresponding AAM of the destination agent platform. If the

destination AAM accepts all agents in the group, the AAM of the sender agent platform starts the migration phase. Otherwise, the sender AAM communicates successively with the AAM of the next best destination agent platform until it finds an available destination agent platform, or it checks the feasibility of all other agent platforms. If no agent platform can accept this agent group, this negotiation fails, and after a certain time period, the sender's agent distribution mechanism restarts this process.

The negotiation phase of our load sharing mechanism is similar to that of the communication localizing mechanism. However, the granularity of negotiation for these two mechanisms is different: the communication localizing mechanism is at the level of an agent, whereas the load sharing mechanism is at the level of an agent group. If the destination agent platform has sufficient system resources for all agents in the selected local agent group, the AAM of the destination agent platform can accept the request for the agent group migration. Otherwise, the destination agent platform refuses the request; it cannot accept part of a local agent group.

Agent Migration Phase

When the sender agent platform receives the acceptance reply from the receiver agent platform, the AAM of the sender agent platform initiates the migration of all agents in the selected local agent group. The following procedure for this phase in the load sharing mechanism is the same as that in the communication localizing mechanism.

3.3 Characteristics

3.3.1 Transparent Distributed Algorithm

The proposed agent distribution mechanisms have been developed as fully distributed algorithms; each agent platform performs its agent distribution mechanisms independently according to information about its workload and the communication patterns of agents on it. There are no centralized components to manage the overall procedure of agent distribution. Moreover, these mechanisms are transparent to multi-agent applications. The only requirement for application developers is to declare candidate agents for agent distribution as movable.

3.3.2 Stop-and-Repartitioning vs. Implicit Agent Distribution

Some dynamic partitioning systems require global synchronization phases for object reallocation [113]. This kind of approach is called *stop-and-repartitioning* [14]. Our agent distribution mechanisms are executed concurrently with applications. The monitoring and decision-making procedures do not interrupt the execution of application agents.

3.3.3 Load Balancing vs. Load Sharing

The second agent distributed mechanism is not a load balancing scheme but a load sharing scheme; it does not attempt to balance the workload of computer nodes participating in an application. The purpose of our agent distribution mechanisms is to reduce the runtime of applications by maximizing intra-node communication instead of inter-node communication. Therefore, only overloaded agent platforms perform the second agent distribution mechanism to move agents from there to lightly loaded agent platforms.

3.3.4 Fine-Grained Approach vs. Coarse-Grained Approach

The proposed agent distribution mechanisms are based on a coarse-grained analysis for the communication patterns of agents. For example, the communication localizing mechanism uses information about the communication dependencies from agents to agent platforms instead of the dependencies among individual agents. Also, the load sharing mechanism uses information about the communication dependencies from agents to agent groups and the dependencies from agent groups to agent platforms.

If our agent distribution mechanisms use fine-grained information, such as the communication dependencies among individual agents, the decision for agent migration would be more accurate. However, the mechanisms require more process power and memory, and take more computational time. The purpose of our mechanisms is to reduce the runtime of an application. If they require considerable system resources, they increase the runtime of the application. Also, if they take more time, the decision made from the given information may no longer be accurate for the current situation.

3.3.5 Individual Agent-based Distribution vs. Agent Group-based Distribution

With the agent group-based distribution mechanism, a local communication locality problem may be solved naturally. For example, when two agents on the same agent platform communicate intensively with each other but not with other agents on the same platform, these agents may continue to stay on the current agent platform even though they have a large amount of communication with agents on another agent platform. If these two agents can move together to the remote agent platform, the overall performance can be improved. However, an individual agent-based distribution mechanism may not handle this situation. Moreover, individual agent distribution may require much platform-level message passing among agent platforms for the negotiation. For example, to send agents to other agent platforms, the agent platforms should negotiate with each other to avoid sending too many agents to a certain agent platform, thus overloading the agent platform. If two agent platforms negotiate with each other for a set of agents at one time, the agent platforms may reduce the number of negotiation messages and the negotiation time.

3.3.6 Stabilization Property

In our dynamic agent distribution mechanisms, when a message from agent i to agent j is passed, variable M_{ij} about this message is only updated; the agent platform of the sender agent updates only this variable for the sender agent. If this variable on both the sender and receiver agent platforms is updated for the sender and receiver agents, these two agents may be switched on their migration phases. To reduce this possibility, agent platforms attempt to start their agent distribution mechanisms at different times. However, because our distribution mechanisms are distributed algorithms, this approach cannot overcome this problem completely. Because this problem also can occur in a circular pattern, it is hard to show the *stabilization property* of our communication localizing mechanism. However, if we change our scheme for this variable, agent migration for agent communication localities will be eventually stopped when the communication patterns of agents are not changed. For example, assume that whenever a message is delivered, the agent platform of the agent with the higher identification number between the names of the sender and receiver agents updates this variable. If the communication patterns of agents are no longer

changed, the agent platforms of agents will be fixed in the order of agents from those with the highest identification numbers to those with the lowest identification numbers. Therefore, agent migration by our communication localizing mechanism will be stopped eventually.

3.3.7 Size of a Time Step

In the monitoring phase, the size of each time step may be fixed to support many kinds of applications. However, if the size of the step is dynamically adjusted by an agent application, the agent distribution mechanisms will adapt more effectively to the behavioral change of the application. For example, in multi-agent simulations, the size of this step may be the same as that of a simulation time. Thus, the size of each time step may vary according to the workload of each simulation step and the processor power. To use a dynamic step size, our agent system has a *reflective mechanism* [101]; agents in the applications are affected by multi-agent platform services, and the services of the multi-agent platform may be controlled by agents in the applications.

3.4 Experimental Results

For our experiments, we used four computers (3.4 GHz Intel CPU and 2 GB main memory) connected by a Giga-bit switch. The agent distribution mechanisms were developed and evaluated on our multi-agent framework called the *Adaptive Actor Architecture* (*AAA*) (see chapter 2 and [64, 66]). AAA is implemented in Java programming language except for the System Monitor. The System Monitor uses system calls to check the status of the CPU and memory, and these functions are implemented in C. The system call functions and the other components in the System Monitor are connected by Java *JNI* (*Java Native Interface*) [52]. To reduce the inter-node communication cost, our agent platforms use TCP instead of Java *RMI* (*Remote Method Invocation*) [55]. Detailed information about AAA is given in chapter 2.

To evaluate the benefit of our agent distribution mechanisms, we conducted micro UAV (Unmanned Aerial Vehicle) simulations on AAA. These simulations included from 2,000 to 10,000 agents; half of them were UAVs, and the others were targets. The micro UAVs performed a surveillance mission on a predefined mission area by collaborating with each other. When a UAV detected a target, the UAV approached the target to investigate it. If a UAV detected more than one target,

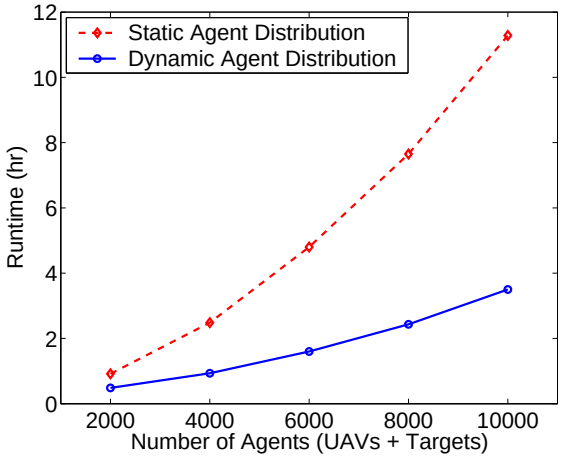


Figure 3.2: Runtimes for Static and Dynamic Agent Distributions

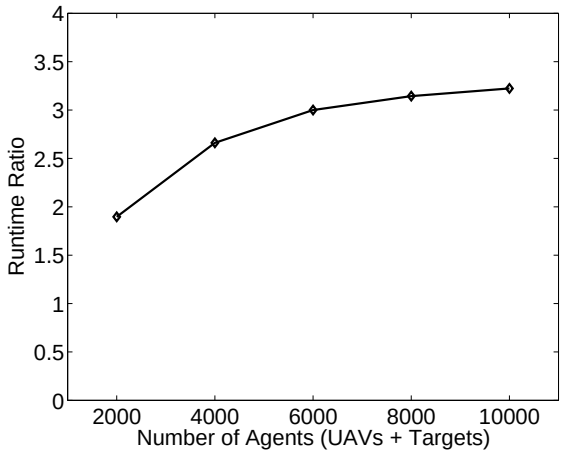


Figure 3.3: Ratios of Static Agent Distribution Runtimes to Dynamic Agent Distribution Runtimes

the UAV handled the targets detected with its neighboring UAVs. Therefore, during the mission time, UAVs continuously communicated with their neighboring UAVs to perform their shared tasks.

These UAV simulations were based on the *agent–environment interaction model* [20]; all UAVs and targets were implemented as intelligent agents, and the navigation space and radar censors of all UAVs are implemented as environment agents. To remove the centralized components in distributed computing, each environment agent was responsible for a certain navigation area. UAVs might communicate directly with each other using their agent names and indirectly through environment agents without agent names. Detailed information about the UAV simulations is given in Appendix A.

Figure 3.2 depicts the difference in runtimes of te simulations in two cases, static and dynamic agent distribution, and Figure 3.3 shows the runtime ratios in both cases. These two figures show the potential performance benefit of dynamic agent distribution. In our particular example, as the number of agents increased, the ratio also generally increased. With 10,000 agents, a simulation using our dynamic agent distribution was 3.2 times faster than a simulation using a static agent distribution. The environment for both cases was implemented as four environment agents, and the simulation time of each run was approximately 22 minutes.

This difference was mainly caused by the difference in the speed of intra-node versus inter-node communication. In our experimental environments, intra-node communication was 600 times faster

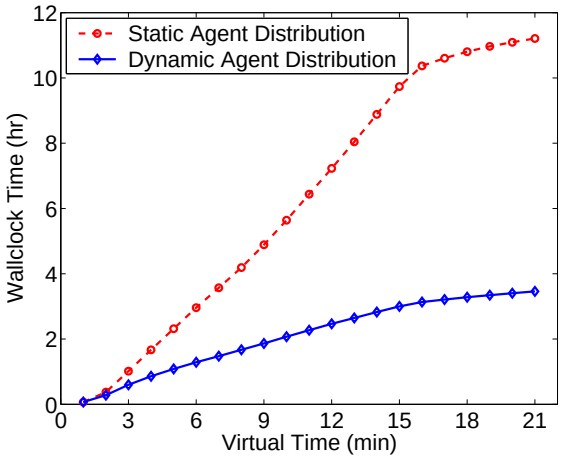


Figure 3.4: Relation between Virtual Time and Wallclock Time

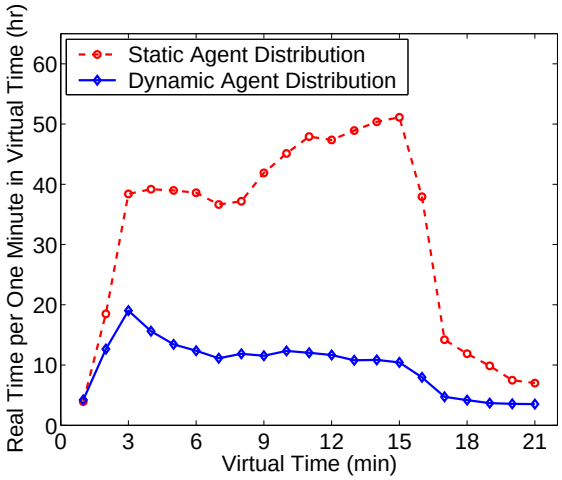


Figure 3.5: The Amount of Real Time Required per One Minute in Virtual Time

than inter-node communication. (When the size of a message was 1,099 bytes, the time required for intra-node message passing was 173 μs, and that of inter-node message passing is 109 ms. These values were computed by dividing the time required to toss a message 20,000 times by 20,000.) When 5,000 UAVs and 5,000 targets were normally distributed in a given mission area, a UAV agent detected around four or five targets at any time, and there were approximately 18 UAVs within its communication range. (Since UAVs started their mission together from the left side to the right side of the mission area, the number of neighboring UAVs in the communication range was greater than this number.)

Figures 3.4 and 3.5 show the properties of our simulations, specifically the relation between wallclock time (or real time) and virtual time in a simulation. Figure 3.4 depicts the relation between virtual time and wallclock time for a UAV simulation including 10,000 agents. In our simulations, each virtual time step was half a second. Figure 3.5 illustrates how much real time was required per 120 time steps (one minute in virtual time). At the beginning of the simulations, each virtual time step required more real time than did the virtual time minute steps after 15 minutes. The reason is that after 15 minutes, some UAVs had finished their missions, and they did not change their locations and ignore other UAVs and targets detected. Therefore, the slope of the line in figure 3.4 becomes smaller after that time. When all UAVs had finished their missions, the UAV simulation ended.

In figure 3.5, at the beginning of these simulations, the real time required per one minute in virtual time increased, because the UAVs needed to communicate more with the neighboring UAVs and targets. Therefore, at approximately 15 minutes in virtual time, the real time consumed in a simulation with static agent distribution reached a maximum. However with dynamic agent distribution, the time consumed at 15 minutes was relatively lower than that at 3 minutes. This is because they needed to communicate more, but they were already properly distributed according to their communication localities. This graph illustrates that even though the amount of communication increased, if the agents were properly allocated according to their communication localities, the change in simulation time was relatively small.

For our experiments, the values of α and β were selected experimentally. As the values of α and β increased, the runtimes of UAV simulations were generally decreased. However, after certain values of α and β, the performance of UAV simulations are considerably similar. For example, with 10,000 agents, when the values of α and β were more than 0.5, the runtime was within the range between 3.48 and 3.52 hours. When their values were 0.1, the range was between 4.17 and 4.27 hours.

Figures 3.6 and 3.7 depict the overhead for monitoring and decision-making processes. The runtime of the decision making process includes the time for monitoring as well as that for decision making. In our experiments, the maximum overhead for monitoring and decision making was 3.1%, and that for monitoring was 2.4%.

We conducted UAV simulations with fine-grained information, communication independencies among individual agents. Figure 3.8 shows the runtimes of UAV simulations using three different agent distribution mechanisms, and Figure 3.9 depicts the runtime ratios of static agent distribution to dynamic agent distribution based on fine-grained information and those of static agent distribution to the dynamic agent distribution based on coarse-grained information. The runtimes of UAV simulations using the dynamic agent distribution based on fine-grained information are still better than those using static agent distribution. However, although with 4,000 agents the dynamic agent distribution based on fine-grained information was only 1.4 times faster than static agent distribution, with 10,000 agents the dynamic agent distribution based on coarse-grained information was 2.3 times faster than the dynamic agent distribution based on fine-grained information.

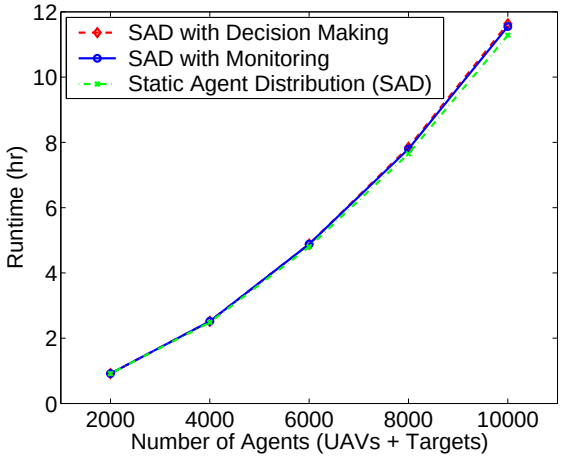


Figure 3.6: Runtimes for Static Agent Distribution with Monitoring and Decision-Making Overheads

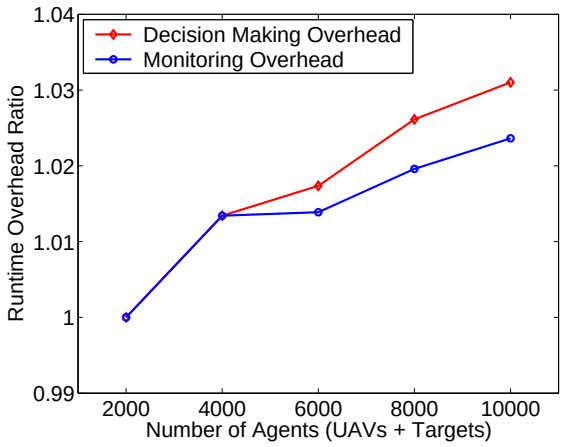


Figure 3.7: Runtime Overhead Ratios of Monitoring and Decision Making

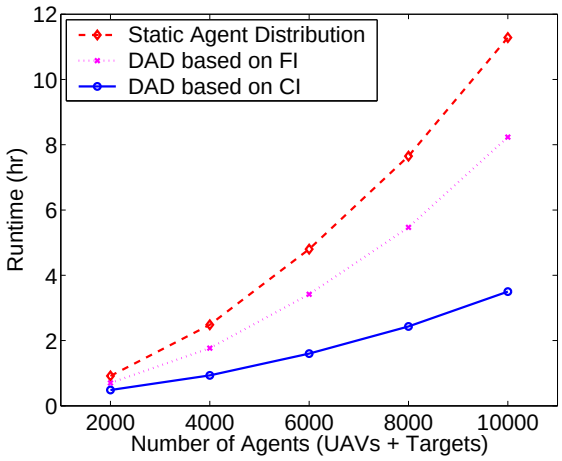


Figure 3.8: Runtimes of UAV Simulations (DAD: Dynamic Agent Distribution, FI: Fine-grained Information, and CI: Coarse-grained Information)

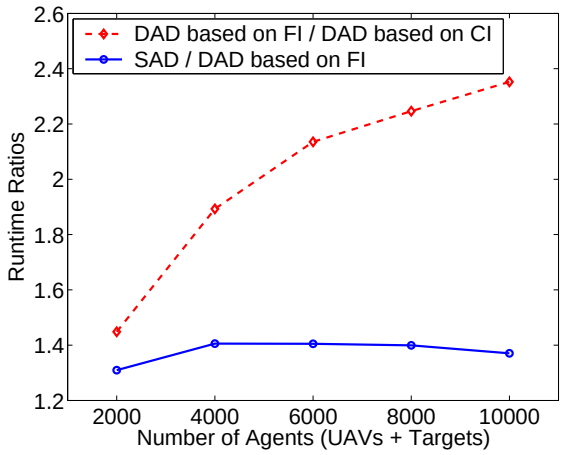


Figure 3.9: Runtime Ratios (SAD: Static Agent Distribution, DAD: Dynamic Agent Distribution, FI: Fine-grained Information, and CI: Coarse-grained Information

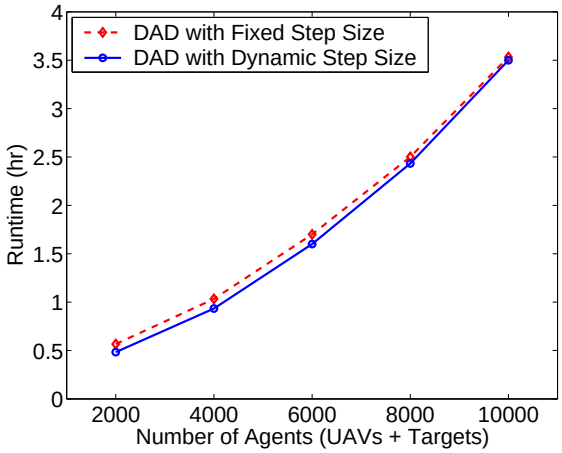


Figure 3.10: Runtimes for Dynamic Agent Distributions (DAD) with Fixed Step Size and Dynamic Step Size

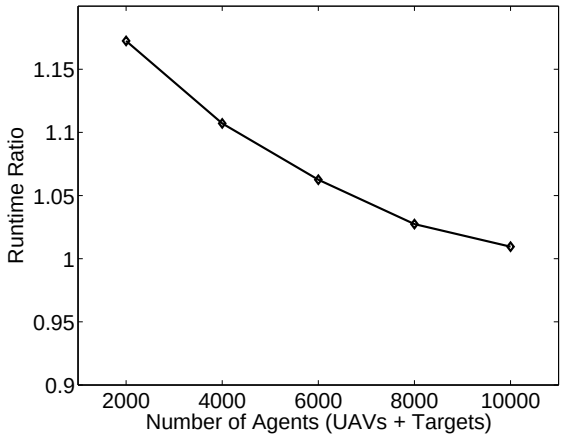


Figure 3.11: Runtime Ratios of Simulations with Fixed Step Size and Simulations with Dynamic Step Size

Figure 3.10 depicts the runtimes for dynamic agent distribution with different policies about the step sizes for agent distribution processes, and figure 3.11 shows the runtime ratios. For the fixed step size, 10 seconds in real time was used for one step, whereas for the dynamic step size, 5 seconds in virtual time was used. Table 3.1 shows the number of steps used for UAV simulations. The performance of dynamic agent distribution with dynamic step size was always better than that of dynamic agent distribution with fixed step size. However, as the number of agents increased, the runtime ratios decreased.

	2000	4000	6000	8000	10000
DAD with fixed step size	204	360	612	900	1272
DAD with dynamic step size	233	233	233	233	233

Table 3.1: Number of Steps for UAV Simulations. DAD represents dynamic agent distribution. The column headers represent the number of agents.

3.5 Related Work

Our dynamic agent distribution can be compared with three approaches: traditional load balancing and task assignment, application partitioning with object migration, and dynamic load balancing with agent migration.

3.5.1 Load Balancing and Task Assignment

To maximize the effect of parallel execution in distributed systems, load balancing is often used. Load balancing mechanisms can be broadly classified into *static* and *dynamic*. Static load balancing mechanisms decide on the assignment of processes (or tasks) to processors (or computer nodes) at compile time, whereas dynamic load balancing mechanisms decide on the assignment of processes at runtime. For static load balancing mechanisms to decide on the assignment of processes, they should use *a priori* knowledge about the processes and processors. Because it is difficult to predict the properties of processes at compile time, such as the execution time of a process and the communication time among processes, and because the properties of processes may be changed during execution, dynamic load balancing mechanisms can generally be used for many types of applications. However, dynamic agent distribution mechanisms require more overhead, such as for monitoring and process migration during execution [59, 95, 117]. Deciding on an optimal assignment of more than two processes is known as NP-hard [16, 17]. However, some heuristic mechanisms that are more efficient but that provide suboptimal assignments of processes have also been published [10, 21, 43, 77].

Because load balancing mechanisms attempt to equalize the workload among processors, their overheads are quite large, especially with a large number of computer nodes. Although load sharing mechanisms have the same goal as load balancing mechanisms, they attempt to prevent computer nodes from being idle while others are overloaded by moving processes from heavily loaded processors to lightly loaded processors [95]. Our agent distribution mechanism uses load sharing instead of load balancing. However, we consider the communication localities of processes as one important factor for distribution, and we use coarse-grained information instead of fine-grained information to minimize the overhead of monitoring and decision making.

Zoltan [40], *PREMA/ILB* [14], and *Charm++* [23] support dynamic load balancing with object migration. Zoltan [40] uses a loosely coupled approach between applications and load balancing algorithms by using an object-oriented callback function interface. However, this library-based load balancing approach depends on information given by the applications, and the applications activate object decomposition. Therefore, without the developers' thorough analysis of the applications, the change in dynamic access patterns of objects may not be detected correctly, and object decomposi-

tion may not be performed at the proper time. The ILB of PREMA [14] also interacts with objects using callback routines to collect information to be used for the load balancing decision making, and to pack and unpack objects. Charm++ [23] uses the Converse runtime system to maintain message passing among objects; hence, it may collect information to analyze communication dependencies among objects. However, this system also requires callback methods for packing and unpacking objects, as others do.

Our agent distribution mechanisms do not interact with agents, but in our agent platforms they receive information from message passing and system monitoring components to analyze the communication patterns of agents and the workload of the agent platforms. Therefore, developers do not need to define any callback method for load sharing. In addition, agent distribution is completely transparent to the application agents.

Grid computing is a distributed infrastructure for computing power and distributed data sharing [44, 70]. Because Grid computing handles large-scale scientific applications, load balancing is one of the major requirements of the Grid environment. However, load balancing in Grid computing involves job partitioning, resource identifying, and job queening procedures as the traditional load balancing distribution does. Genaud et al. [49] proposed load balancing scatter operations for Grid computing, and Banino et al. [13] proposed load balancing based on the master/slave paradigm. However, they are static load balancing. In dynamic load balancing, system load and communication time are important factors. However, Grid computing considers the communication time between hosts, such as the round trip time between client and server nodes, instead of the communication time between tasks (or objects) [85]. Cao et al. [30] used agent-based service discovery for load balancing in Grid computing, but their system focused on only the workload of computer nodes; availability, workload average, and idle time of each node.

Peer-to-Peer (*P2P*) systems are distributed systems for sharing resources, such bandwidth, computational power, and data, without a centralized server or authority. Although P2P systems have goals similar to the Grid's, P2P should handle instability, transient populations, fault tolerance, and self-adaptation, because computer nodes can arrive in and depart from the systems at any time [7]. An important problem in P2P is how to distribute data and find them among a large number of compute nodes, and this operation is based on the *Distributed Hash Table* (*DHT*)

[72]. Therefore, load balancing in P2P focuses mainly on data distribution [51, 72, 71] and system organizing [1].

The *High Level Architecture* (*HLA*) is software architecture used to build component-based distributed simulations [37, 75]. However, the *Run Time Infrastructure* (*RTI*) in HLA supports neither federate migration nor load balancing. Therefore, several research groups have proposed federate migration and load balancing services [27, 103]. However, these load balancing mechanisms focus on the balanced workload among computer nodes and do not consider the communication cost between federates. Pang and Weisz [84] proposed a load-balanced object placement mechanism to reduce the communication cost. This dynamic object placement mechanism handles objects individually, whereas our agent distribution mechanisms analyze the communication dependency, not between individual agents but between agents and agent groups or agent platforms. Our approach reduces the space and computational overhead of this service with a large number of agents.

3.5.2 Application Partition with Object Migration

J-Orchestra [107], *Addistant* [104], and *JavaParty* [89] are automatic application partitioning systems for Java applications. They transform input Java applications into distributed applications using a bytecode rewriting technique. They can move Java objects to take advantage of locality. However, they differ from our approach. First, although they move objects to take advantage of data locality, our approach moves agents to take advantage of communication locality. Second, the access pattern of an object differs from the communication pattern of an agent. For example, although a data object may be moved whenever it is accessed by other objects on different platforms, an agent cannot be migrated whenever it communicates with other agents on different platforms.

3.5.3 Load Balancing with Agent Migration

Many agent frameworks, such as *CORMAS* [19], *MACE3J* [47], *MadKit* [56], *XRaptor* [24], and *Zereal* [109] have been developed for large-scale agent applications, such as agent-based simulations. However, these agent frameworks do not trigger agent migration for communication locality. Thus, each agent should move itself so as to reduce its inter-node communication. For example, Zereal

[109] is a distributed agent framework used for large-scale simulations of Massively Multiplayer Online Games (MMOGs). In Zereal, agents migrate from one computer to another according to a world topology. Therefore, the migration of an agent is decided only by itself. In our agent framework, an agent migrates according to its own decision, but it is also moved by its agent platform according to its communication locality.

Some agent frameworks have considered dynamic agent distribution with mobile agents [31, 33, 38, 50, 73, 82, 115]. *FLASH* [82], *MALD* [31], *MATS* [50], and Traveler [115] mainly focused on only the workloads of computer nodes. Desic and Huljenic [39] used mobile agents for data collection, node monitoring, and task distribution. Although these authors considered the latency and bandwidth of client–server applications, they did not consider the communication patterns of agents. Keren and Barak [73] showed that dynamic agent distribution could outperform static agent distribution by 17–40%. They used information about both the resource utilization of each node and the communication patterns of individual agents. The *IO* of *SALSA* [38] provides various load balancing mechanisms for multi-agent applications, and the IO also analyzes the communication patterns among individual agents.

The *Comet* algorithm assigns agents to computer nodes according to their credit [33]. The credit of an agent is decided by its computational load, intra-communication load, and inter-communication load. Chow and Kwok [33] emphasized the importance of the relationship between the intra-communication and inter-communication of each agent. We used similar ideas, but with some important differences. The authors' system included a centralized component to decide on agent assignments, and their experiments included a small number of agents, i.e., 120 agents. In contrast, our agent distribution mechanisms were implemented as fully distributed algorithms, and we experimented with 10,000 agents. Because of the large number of agents, our agent distribution mechanisms did not analyze the communication dependencies of all individual agents, but only the dependencies between agents and agent platforms and between agent groups and agent platforms. As we showed in chapter 3, when dynamic agent distribution uses the communication dependencies of individual agents, its benefit is reduced considerably in large-scale multi-agent systems.

3.6 Discussion

This chapter explains two dynamic agent distribution mechanisms used in our multi-agent framework; these agent distribution mechanisms dynamically allocate agents according to their communication localities and the workloads of computer nodes. The main contribution of this research is to provide agent distribution mechanisms that focus on the communication localities of agents to collocate agents that communicate intensively with each other, and to move an agent group instead of individual agents in the case of agent distribution for load sharing. The proposed mechanisms consider the communication dependencies between agents and agent platforms and the dependencies between agent groups and agent platforms, instead of the communication dependencies among individual agents. These mechanisms are fully distributed algorithms and are transparent to agent applications. Our experimental results show that micro UAV simulations using dynamic agent distribution are approximately 3.2 times faster than those with static agent distribution.

To maximize the effect of our approach, several characteristics of the applications are assumed. First, an application includes a large number of agents so that more than one computer node is required for the application. Second, numerous agents communicate intensively with each other. Thus, the communication locality of each agent is one of the most important factors for the overall performance of the applications. Third, the communication patterns of agents are changing continuously. Hence, static agent distribution mechanisms may not work; initial optimized agent distribution will no longer be efficient after several processing steps. Under these constraints, the proposed agent distribution mechanisms may significantly improve the overall performance of the entire system.

Chapter 4

Application Agent-oriented Middle Agent Services

4.1 Introduction

In open multi-agent systems where agents may enter and leave at any time, an agent cannot know the names of all the other agents with whom it needs to communicate. In this environment, middle agent services, such as brokering and matchmaking services, are very effective to find service or receiver agents [35, 36]. Application agents can find the names of other agents (with *matchmaking services*) or send messages to other agents (with *brokering services*) using their attributes (such as methods, behaviors, characteristics, or aliases) instead of their names. Middle agent services are provided by either a component in the agent framework or a specialized agent called a *middle agent*. With a common shared space, called *tuple space*, which is managed by a middle agent, agents can register their attributes with their names on the space, and they can communicate with each other using attribute information from the space. Many middle agents are based on the Linda model [32, 48]. In Linda-like middle agent models, the registered data are called *tuples*, and the attributes given by an agent to retrieve tuples are represented by a *tuple template*. When a middle agent receives a tuple template, this middle agent attempts to find matched tuples with the given template. It then returns them to the sender agent or uses them to forward a message given by the sender agent to the retrieved service agents.

With matchmaking services, an application agent may retrieve the names of agents with certain attributes from a middle agent. When partial attributes are given by an application agent, the middle agent may return the names of agents with their fully registered attributes in the matched

tuples. Since the application agent finally selects receiver agents with their attributes retrieved, the agent does not need to deliver all of its interest to the middle agent and hence is not limited to the expressive power of the middle agent-supported primitives. With brokering services, an application agent should send all its interest information to a middle agent, and the middle agent can select the receiver agents of a message; hence, the sender agent is limited to the expressive power of the primitives. Therefore, matchmaking services may be more flexible than brokering services. However, an application agent using brokering services does not need to receive information about the names and attributes of selected agents. Since the amount of this information may be very large, brokering services are more efficient than matchmaking services. Moreover, by using the common attribute of several receiver or service agents, an application agent may send a message to multiple agents with one message-sending primitive.

Many middle agent models have improved the expressive ability of the tuple space model [26, 41, 83, 108]. However, previous research has focused on the ability of middle agents because middle agents possess both the data to be searched and the algorithms to be used for searching. In this chapter, we describe the *ATSpace* (*Active Tuple Space*) model to provide application agent-oriented middle agent services; a middle agent manages data but the middle agent may use search algorithms delivered from application agents. ATSpace increases the dynamism and flexibility of the tuple space model, but it also introduces some security threats, as codes developed by different groups with different interests are executed in the same space. We discuss the implications of these threats and our solutions to mitigate them.

4.1.1 A Motivating Example

A simple example may show the motivation of the ATSpace model that provides application agent-oriented middle agent services. In general, a middle agent user with a complex matching query is faced with the following two problems:

1. **Expressiveness Problem**: A matching query cannot be expressed using the middle agent primitives.

2. **Incomplete Information Problem**: Evaluating a matching query requires information that is not available to a middle agent.

For example, assume that a middle agent 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>`. A buyer agent can access the tuple space in a middle agent to find seller agents that sell, for instance, computers or printers and may execute the following query:

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

A template-based middle agent may not support the request of this buyer agent because, first, it may not support the "two best" primitive (expressiveness problem), and second, it may not have information about the distance between cities (incomplete information problem). Faced with these difficulties, the buyer agent with Q1 has to transform it to a tuple template-style query (Q2) to be accepted by a template-based middle agent. The 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 two best sellers from them. To select seller agents located within 50 miles, the buyer agent must have a way to roughly estimate distances between cities or information about nearby cities. Finally, it should send these seller agents a message to start the negotiation process.

An apparent disadvantage of this approach is that a large amount of data is moved from the middle agent to the buyer agent. When the middle agent includes a large number of tuples related to computer sellers, the size of the message to be delivered is also large. To reduce the 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 an object that inspects tuples in the middle agent and select the best two sellers that satisfy the buyer criteria; the object would also carry information about nearby cities.

Figure 4.1 shows how buyer agent A1 with Q1 sends a message to seller agents through an atSpace.[1] The seller agents with names A2 and A3 are selected by the search algorithm, and the atSpace delivers the `sendComputerBrandName` message to them as a brokering service. Finally, the seller agents send the information about the brand names of their computers to the buyer agent.

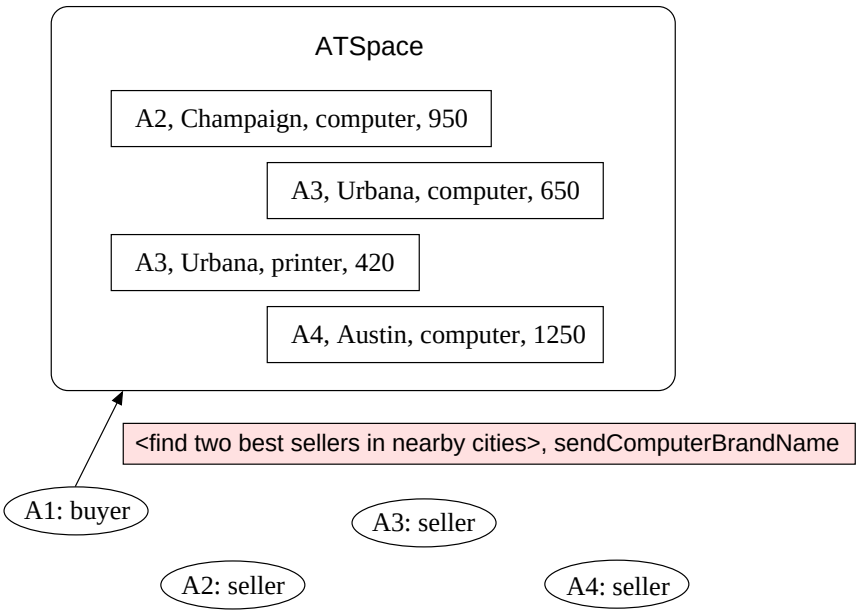


Figure 4.1: An Example of a Brokering Service Using ATSpace

4.2 Application Agent-oriented Middle Agent Services

4.2.1 Architecture of ATSpace

ATSpace consists of three components: a *tuple space*, a *message queue*, and a *tuple space manager* (see Figure 4.2).

Tuple space T in an atSpace is used as a shared pool for *agent tuples*, $t = \langle a, p_1, p_2, ..., p_n \rangle$, each of which consists of a name field, a, and a property part, $P = p_1, p_2, ..., p_n$, where $n \geq 1$; each tuple represents an agent whose name is given by the first field and whose attributes are given by the subsequent fields. ATSpace enforces the rule that no two agent tuples can have the same agent

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

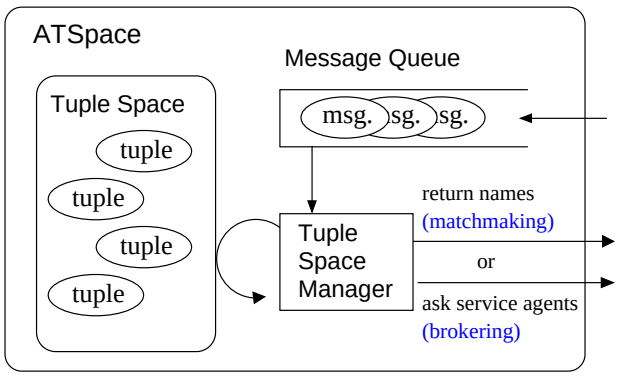


Figure 4.2: Architecture of ATSpace

names with identical property parts. 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):

$$\forall t_i, t_j \in T : i \neq j \rightarrow [\quad (t_i.a = t_j.a) \rightarrow (t_i.P \neq t_j.P)) \quad \wedge$$
$$(t_i.P = t_j.P) \rightarrow (t_i.a \neq t_j.a) \quad]$$

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, alternatively, 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 an object has predefined public methods, such as `find`. The `find` method is called with tuples by the tuple space manager in its atSpace as a parameter, and this method returns the names of agents selected by the search algorithm specified in the mobile object.

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

4.2.2 Operation Primitives

The ATSpace model supports three fundamental primitives of the tuple space model: `write`, `read`, and `take`.[2] The `write` primitive is used to register an agent tuple in an atSpace; the `read` primitive is used to retrieve an agent tuple that matches a given criterion; and the `take` primitive is used to retrieve a matched agent tuple and remove it from the atSpace. When there is more than one agent tuple whose property matches the given criterion, one of them is randomly selected by the agent tuple manager. When there is no matching tuple, these primitives return immediately with an exception. To retrieve all agent tuples that match a given criterion, `readAll` or `takeAll` primitives should be used. The format 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, whereas the others cause service request messages. Note that the `write` primitive does not include an agent tuple but a *tuple*, which contains only the agent's property. This is to avoid a case in which an agent tries to register a property using another agent name in an atSpace. This tuple is then converted to an agent tuple with the name of the sender agent before the agent tuple is inserted into an atSpace.

In some applications, updating agent tuples happens very often. For such applications, *availability* and *integrity* are of great importance. Availability ensures that at least one agent tuple exists at any time, whereas integrity ensures that old and new agent data do not exist simultaneously in an

[2] In the Linda Model [32, 48], the `out`, `rd`, and `in` primitives perform the same operations as the `write`, `read`, and `take` primitives in the ATSpace model. In the JavaSpace model [102], the same names of primitives are used.

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 ensure 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. The primitives for matchmaking services 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 criterion, whereas the `searchAll` primitive is used to retrieve the names of all the service agents that match a given property.

The primitives for a 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 criterion, 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 send mobile objects to support an application agent-oriented search algorithm. We call the matchmaking or brokering services used with mobile objects *active matchmaking* or *brokering services*, because application agent-supported objects are executed in an atSpace. `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(AgentTuple[] ataTuples);
```

Service Specific Request Primitive

One drawback of the previous brokering primitives (i.e., `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 criterion. 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 to selected service agents. However, this approach suffers from the same problems as a template-based middle agent.

To address these shortcomings, we introduce the `exec` primitive. 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; each

agent message includes the name of a receiver agent and a service call message. The atSpace will deliver the service call messages to the selected agents. The formats of the `exec` primitive and the `doAction` method are as follows:

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

4.2.3 Security Issues

By allowing a mobile object to be supplied by an application agent, ATSpace supports active matchmaking and brokering services, which increase the flexibility and dynamism 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 too 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 gain unauthorized access or carry out 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 the 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

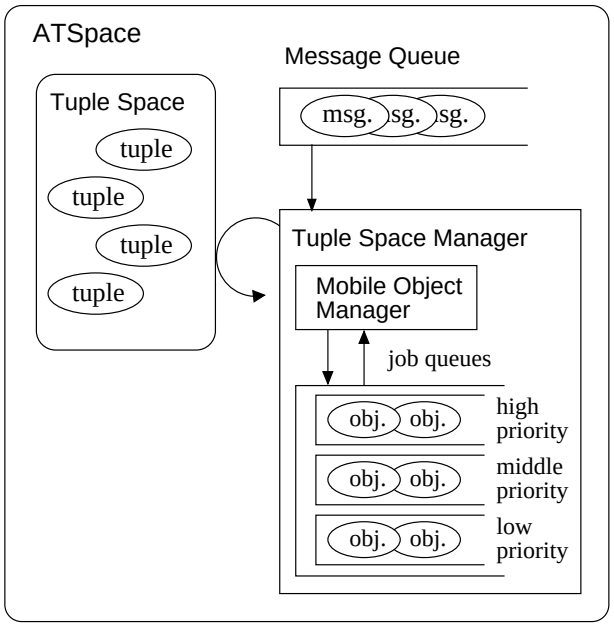


Figure 4.3: Extended Architecture of ATSpace

deep copy, although this will violate the integrity of tuples if an agent tries to delete or change the tuples.

To address the denial of service by consuming all processor cycles, we deployed user-level thread scheduling. Figure 4.3 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 *Mobile Object 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 extended the architecture of ATSpace by implementing job queues.

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 application agents. Also, an atSpace may limit agents to modify only their own tuples.

4.3 Evaluation

The performance benefit of ATSpace can be measured by comparing its active brokering service with a matchmaking service of the template-based middle agent 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 evaluate ATSpace analytically, we use the scenario described in section 4.1.1 in which a service-requesting agent has a complex query that is not supported by the template-based middle agent model.

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

- $\texttt{Service\_Request}_{\texttt{template}}$: a template-based service request message that includes Q2. A service-requesting agent sends this message to a middle agent to bring a superset of its final result.
- $\texttt{Service\_Reply}_{\texttt{template}}$: 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 the 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 its query and sends only a service request message ($\texttt{Service\_Request}_{\texttt{ATSpace}}$) to an atSpace. This message contains a mobile object that represents its criteria along with a service call message to be sent to the agents that satisfy the criteria. The last n messages have the same explanation as in the template-based middle agent model except that the sender is the atSpace instead of the service-requesting agent.

Although 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 middle agent model and the ATSpace model is given by the following equation:

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

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 savings 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 too many tuples from the template-based middle agent.

The amounts of storage space used on the client agent's and middle agent's computers are similar in both cases. In the template-based approach, 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 the creator of an atSpace opts to use a shallow copy of the data, the size of the copy in the atSpace is much smaller than that of the copy in the client agent.

The difference in the 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 minimized. 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, to evaluate Q2, the template-based middle agent needs to inspect all the tuples it has, and second, the service-requesting agent inspects those 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 number of 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 may pass Q2 and be re-evaluated 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 the external communication cost is high, ATSpace will result in a time savings.

4.4 Experimental Results

In addition to this analytical evaluation, we applied the ATSpace model to our micro UAV (Unmanned Aerial Vehicle) simulations described in section 3.4. During the mission, a UAV needs to communicate with other neighboring UAVs within its local communication range (see Figure 4.4). We use the active brokering primitives of ATSpace to accomplish this broadcasting behavior. At every simulation step, every UAV uses the `update` primitive to replace information about its location on an atSpace. When local broadcast communication is needed, the sender UAV (considered a service-requesting agent from the ATSpace perspective) uses the `deliverAll` primitive and supplies as a parameter a mobile object 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 search 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 the 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.

We also used ATSpace to simulate the behavior of all the UAV radar sensors (see Figure 4.5). Each UAV should detect targets within its sensing radar range. The environment agent, which is the simulator component responsible for accomplishing this behavior, sends the `exec` message to perform this task to an atSpace. The mobile object supplied with the `exec` message computes distances between UAVs and targets, and selects neighboring targets for each UAV on the atSpace address space. It then creates messages, each of which consists of the name of its receiver UAV and a service call message to be sent to the receiver UAV agent. Finally, the mobile object returns the set of messages to the tuple space manager, which in turn sends the messages to the respective agents.

Figure 4.6 demonstrates the savings in the runtimes of simulations using ATSpace compared with those using a template-based middle agent that provides matchmaking services with the same semantics. Figure 4.7 shows the runtime ratios of a template-based middle agent to ATSpace. In these experiments, UAVs use either active brokering services or matchmaking services to find their neighboring UAVs. In both cases, the middle agent includes information about the locations

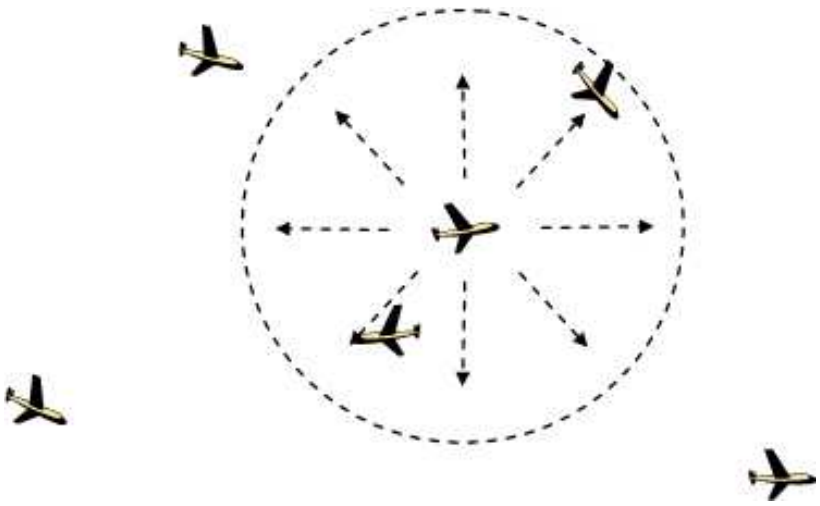


Figure 4.4: Simulation of Local Broadcast Communication

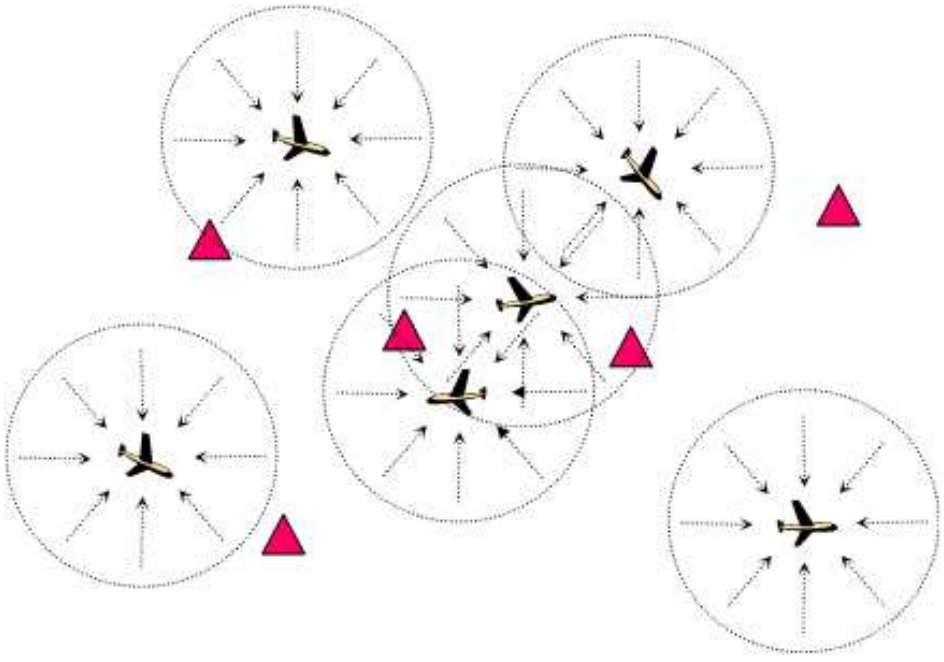


Figure 4.5: Simulation of All the Radar Sensors

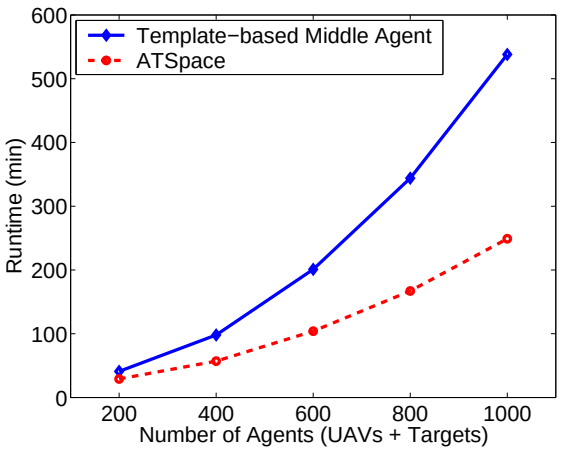


Figure 4.6: Runtimes of Simulations Using a Template-based Middle Agent and ATSpace

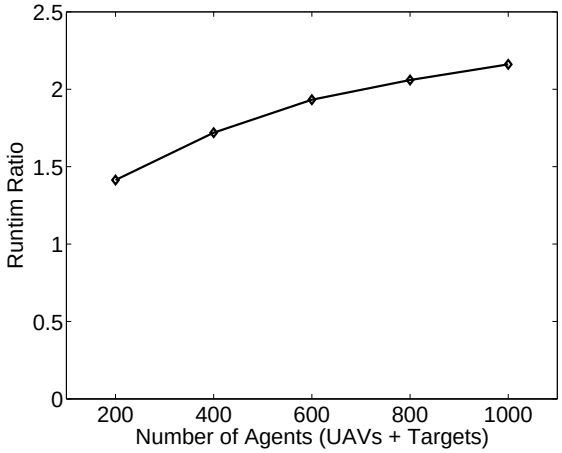


Figure 4.7: Runtime Ratios of a Template-based Middle Agent to ATSpace

of UAVs and targets. In the active brokering service, UAVs send mobile objects to an atSpace, whereas UAVs using the matchmaking service send tuple templates. The simulation time for each run is approximately 35 minutes, and the runtime depends on the number of agents. As the number of agents is increased, the difference becomes large. With 1,000 agents, a UAV simulation using ATSpace is more than two times faster than that using a matchmaking service.

Figure 4.8 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 as the number of agents increases. 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 very small; it was in fact less than 0.01% in our simulations.

Figure 4.9 shows the total message size required in the two approaches. When the search queries are complex, the total message size in the ATSpace approach is much smaller than that in the template-based middle agent approach. In our UAV simulations, the search queries are rather complex and require mathematical calculations, and hence, the ATSpace approach results in a considerable bandwidth savings. It is also interesting to note the relationship between the runtime (as shown in Figure 4.6) and the bandwidth savings (as shown in Figure 4.9). This relationship supports that the savings in the total operation time by ATSpace is largely due to its superiority in utilizing the bandwidth efficiently.

Figure 4.10 shows that the average size of a message in ATSpace increases much more slowly

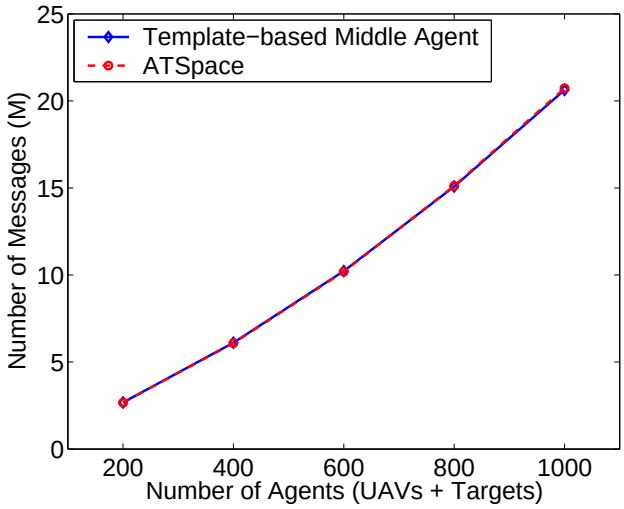


Figure 4.8: The Number of Messages for a Template-based Middle Agent and ATSpace

than that using a template-based middle agent. This result supports the idea that ATSpace is more scalable than a template-based middle agent.

4.5 Related Work

ATSpace can be compared to four related approaches: other tuple space-based middle agent models, the Java Applet model, the mobile agent model, and SQL (Structured Query Language) in the database. ATSpace is also based on the *tuple space* [48], but it provides active matchmaking and brokering services with mobile search objects as well as general matchmaking and brokering services.

4.5.1 ATSpace vs. Other Middle Agent Models

Our work is related to *Linda* [32, 48] and its variants, such as *JavaSpaces* [102] and *TSpaces* [76]. In these models, processes communicate with other processes through a shared common space called a blackboard or a tuple space without considering the references or names of other processes [32, 88]. This approach was used in several agent frameworks, for example, *OAA* [12] and *EMAF* [79]. However, these models support only primitive features for anonymous communication among processes or agents.

From the middle agent perspective, *Directory Facilitator* in the *FIPA* (*Foundation for Intelligent*

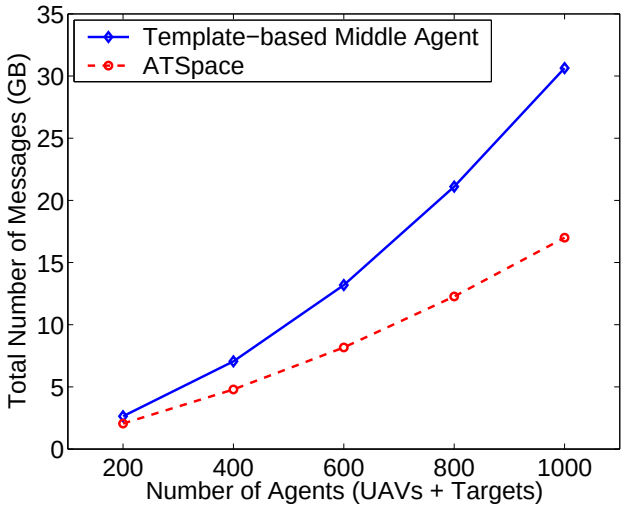


Figure 4.9: The Total Size of Messages for a Template-based Middle Agent and ATSpace

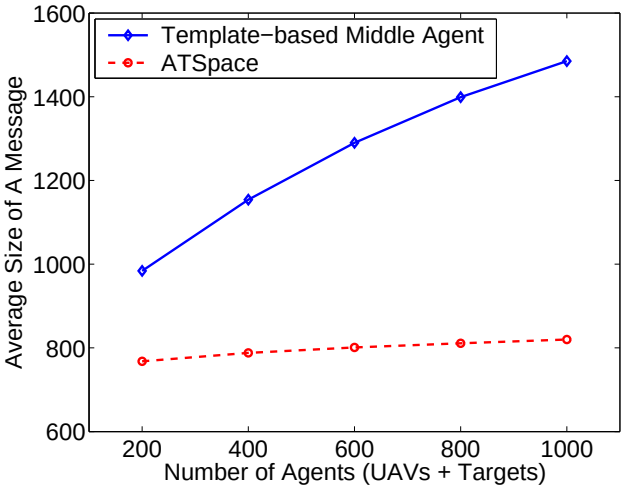


Figure 4.10: The Average Size of a Message for ATSpace and a Template-based Middle Agent

Physical Agents) platform [45] and *Broker Agent* in *InfoSleuth* are related to our research [60]. However, these systems do not support customizable matching algorithms. Although *ActorSpace* [3, 28] allows the use of regular expressions in its matching template, its expressive power is still limited.

Some work has been done to extend the matching capability in the tuple space model. *Berlinda* [108] allows a concrete entry class to extend the matching function, and $\mathcal{TS}$ [61] 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. On the other hand, *OpenSpaces*

[41] does provide a mechanism to change matching policies during execution. OpenSpaces groups entries in its space into classes and allows each class to have its own matching algorithm. The manager for each class of entries can change the matching algorithm during execution. All agents that use entries under a given class are therefore 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* [91] 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 the `rd` and `in` operators. However in ATSpace, the client agent that supplies mobile objects can have a global overview of the tuples stored in the shared space; hence, it can support global search behavior rather than the tuple-based matching behavior supported in Object Space. For example, when using ATSpace a client agent can find the best 10 service agents according to its criteria, whereas this behavior cannot be achieved in Object Space.

TuCSoN [83] and *MARS* [26] 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, ATSpace uses mobile objects to support search algorithms defined by client agents. A reactive tuple handles certain types of tuples and affects various clients, whereas a mobile object handles various types of tuples and affects only its creator agent. Also, TuCSoN and MARS do not provide an execution environment for client agents. Therefore, they may be considered as orthogonal approaches and can be combined with our approach.

Mobile Co-ordination [93] allows agents to move a set of multiple tuple space access primitives to a tuple space for fault tolerance. In *Jada* [34], one primitive may use multiple matching tuple templates. In *ObjectPlaces* [94], dynamic objects are used to change their state whenever corresponding objectplace operations are being called. Although all of these approaches improve the search ability of tuple spaces with a set of search templates or dynamic objects, ATSpace provides

more flexibility to agents with their own search code.

The *Data Distribution Management* (*DDM*) in *HLA* [75] and *Mercury* [15] provides multi-range data transmission. However, multi-range matching is still limited by its expressive power. For example, for a two-dimensional routing space, a region in DDM defines only a rectangular volume, whereas a search object in ATSpace can define any type of volume.

Although in some systems [15, 78] tuples (or shared states) can be moved by the system, ATSpace agents cannot move tuples to each other by themselves. Currently, application agents should decide to which ATSpace agents they will send their tuples. This reduces the location transparency of tuples to agents. If ATSpace can support the dynamic distribution of tuples, application agents can be freed from the ATSpace agent selection problem. For this purpose, ATSpace should be extended to manage the relationship between tuples and application agents. This overhead may be reduced by managing the relationship between tuples and agent platforms. Tuples do not communicate with each other, but they may nevertheless be related to each other. For example, tuples of two closely located UAV agents are more likely to be used together than two tuples of two agents that are located far apart. Therefore, migrating randomly selected tuples may cause other side effects.

4.5.2 ATSpace 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 differs significantly from a Java applet: a mobile object moves from a client computer to a server computer, whereas 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; however, a Java applet receives parameters but no method call from processes on the same computer.

4.5.3 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*, whereas 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: for example, objects in CORBA [112] or DCOM [105] may refer to remote objects.
- The method interface of an object is precisely predefined, and this interface is directly used by a calling object. 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.
- Whereas 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 reference passing in a method call, but an agent uses 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 value passing requires a deep copy of the data.

Besides the features of objects, we imposed 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, it 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. However, a mobile object raises the same security issues on the server side.

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

4.5.4 Mobile Objects vs. SQL

In the database area, SQL is used to communicate a relational database [46, 54]. Using SQL, a user can effectively access related data. Although SQL is based on relational algebra, it is more powerful than relational algebra because it includes aggregate functions, the ordering of tuples, and so on. However, SQL is not as powerful as a general-purpose programming language [92]. There are queries that cannot be expressed in SQL but that can be programmed in general programming languages such as Java. For example, if a client program performs different actions according to the field values of data in a database, SQL cannot support this operation with one statement. Mobile search objects in ATSpace are implemented in Java, and hence, ATSpace has more expressiveness power than SQL. However, SQL servers attempt to overcome the limitation of the power of SQL by using *stored procedures* [90]. To reduce communication overhead between a client program and an SQL server, the stored procedure mechanism can migrate a set of SQL statements and execute them locally on the server side.

4.6 Discussion

This chapter explains the ATSpace model, which works as a common shared space to exchange data among agents, provides a middle agent to support application agent-oriented brokering and

matchmaking services, and supports an execution environment for mobile objects utilizing data on the space. We describe some security threats that arise when using mobile objects for agent coordination, along with some mechanisms to mitigate them. We also compare the ATSpace model with the template-based middle agent model. Our experiments with UAV surveillance simulations show that this middle agent model may be effective in reducing coordination costs in large-scale multi-agent applications.

ATSpace has overheads for sending a message to multiple receiver agents. However, this approach considerably reduces the total agent communication cost when multi-agent applications have the following attributes: first, a middle agent should manage numerous tuple data; second, a condition to find service agents is so complex that this condition cannot be expressed as either a tuple template or a range query; third, a middle agent and client agents are located on different agent platforms. Therefore, the template-based middle agent approach significantly increases the agent communication overhead; third, the agent platform where the middle agent is located is not significantly overloaded.

Chapter 5

Message Passing for Mobile Agents

5.1 Introduction

In multi-agent systems, agents perform their shared tasks by sending messages to each other. A message to be sent should include the address (or name) of its receiver agent. In many multi-agent systems [80, 110], the address of an agent consists of the IP address of its *birth agent platform*,[1] the port number of the agent platform, and the unique identification of the agent. If no agents change their locations (or agent platforms), this addressing scheme is suitable for message passing. However, mobile agents change their locations to find better system resources or to reduce their communication overhead. With mobile agents, this addressing scheme is not suitable to deliver a message to a receiver mobile agent, because the agent may not be located on its birth agent platform. Therefore, mobile agent frameworks provide additional mechanisms to deliver messages to mobile agents. The *forwarding* and *agent locating* mechanisms are often used.

With the forwarding mechanism, whenever a mobile agent migrates from one agent platform to another, the previous agent platform where the agent was located holds information about the next agent platform to which the agent has migrated. When an agent platform receives a message for a mobile agent that was located on it but that left for another agent platform, this agent platform forwards the message to the next agent platform of the receiver agent. This approach has one shortcoming: a message may follow a long link to reach the final receiver agent. This problem can be mitigated by shortening the link; when a receiver agent receives a message through other agent platforms besides the *source agent platform* where the sender agent is located, the *destination agent*

[1] The *birth agent platform* of an agent is the platform where the agent is created.

platform where the receiver agent exists sends a platform-level message to the agent platforms on the path to update information about the current location of the receiver agent on them. However, this optimization requires that a message include information about all intermediate agent platforms and that additional platform-level messages be delivered to intermediate agent platforms.

With the agent locating mechanism, agents use an agent naming server to find the locations of other mobile agents. Whenever an agent migrates from one agent platform to another, the agent should update information about its new location on the shared space. By interacting with this naming server, a sender agent can find the current location of a receiver agent. This approach has one shortcoming: a sender agent platform should communicate with the agent naming server to find the location of the receiver agent of a message before sending it. This problem worsens if the source and destination agent platforms are located close together, and the agent name sever is located very far from the source and destination agent platforms.

Therefore, several optimization mechanisms have been proposed to send a message directly from the source agent platform to the destination agent platform when the sender and receiver agents are not located on the same agent platform. In this chapter, we evaluate some optimization mechanisms and compare them with their base middle agent mechanisms.

5.2 Message Passing for Agents

The address of a static agent may include the IP address of its birth agent platform, the port number of the agent platform, and the unique identification of the agent. We call this the *universal agent address* (UAA):

```
UAA ::= bp_IP_address ":" bp_port_number "/" agent_id
```

When an agent platform receives a message, the agent platform finds the current agent platform of the receiver agent of the message from the `bp_IP_address` part of its address. If the sender and receiver agents of a message are not located on the same agent platform, the *source agent platform* where the sender agent exists sends the message to the *destination agent platform* where the receiver agent exists using `bp_IP_address` and `bp_port_number` in the address of the receiver agent. When

the destination agent platform receives the message, this agent platform finds the receiver agent using `agent_id` and delivers the message to the agent.

If an agent can migrate from its birth agent platform to another, the UAA of a mobile agent cannot tell where the mobile agent is currently located. Therefore, to deliver a message to the current agent platform of the mobile agent, this protocol should be extended. One extension is that each agent platform performs a forwarding service for its *child mobile agents* who were created on it and were migrated to other agent platforms. For this purpose, the birth agent platform of each mobile agent manages information about the current agent platform of the agent. Thus, when an agent platform receives a message for a mobile agent who was created on it and was migrated to another agent platform, this agent platform forwards the message to the current agent platform of the receiver agent. Also, every mobile agent should update its current location on its birth agent platform whenever it migrates. In this extension, each agent platform works as a *forwarding server* for its child mobile agents. However, this extension introduces an additional hop to send a message to its final mobile agent when the receiver agent does not exist on its birth agent platform. To eliminate this additional hop, we still need to optimize message passing for mobile agents.

In this chapter, we do not consider a centralized server for forwarding or agent naming services. The centralized server can be a bottleneck in agent communication, and the server may be located far from the agent platforms of the sender and receiver agents. Therefore, in this chapter we assume that each agent platform provides either a forwarding or an agent naming service for its child agents. The following describe base and optimization mechanisms of message passing for mobile agents.

1. The *forwarding-based message passing* (FMP) mechanism: Every message is delivered to the birth agent platform of the receiver agent of the message, and the birth agent platform forwards the message to the current agent platform of the receiver agent if the receiver agent does not exist on its current agent platform.

2. The *forwarding and location address-based message passing* (FLAMP) mechanism: This mechanism is based on FMP except that it uses the location addresses of agents instead of their UAAs. If the current agent platform of the receiver agent differs from its birth agent platform, the source agent platform sends the message to the agent platform specified in the location address of the receiver agent instead of its birth agent platform.

3. The *forwarding and location cache-based message passing* (FLCMP) mechanism: This mechanism is also based on FMP, but information about the current agent platforms of remote agents is locally cached on each agent platform when the information is obtained.

4. The *agent locating-based message passing* (ALMP) mechanism: When the receiver agent of a message is not located on the same agent platform as the sender agent, the sender agent platform locates the current agent platform of the receiver agent by interacting with the birth agent platform of the receiver agent before sending the message.

5. The *agent locating and location cache-based message passing* (ALLCMP) mechanism: This mechanism is based on ALMP, but information about the current agent platforms of remote agents is locally cached on each agent platform when the information is obtained.

In general, forwarding mechanisms do not require a forwarding service provided by the birth agent platforms of mobile agents, and mobile agents do not need to update their locations on their birth agent platforms. However, we believe that this extension mitigates the original problem of general forwarding mechanisms; as an agent migrates on many agent platforms, the message forwarding link to the agent becomes longer. Therefore, we consider FMP as the base mechanism of FLAMP and FLCMP. In this section, we describe three optimization mechanisms. Detailed explanations of forwarding and agent locating mechanisms are given in [6, 98].

5.2.1 FLAMP: Forwarding and Location Address-based Message Passing

The FLAMP mechanism extends the address format of an agent; the new address format of the agent includes information about the current location (or agent platform) of an agent as well as its UAA. We call this a *location-based agent address* (LAA):

```
LAA ::= cp_IP_address ":" cp_port_number "/"
        bp_IP_address ":" bp_port_number "/" agent_id
```

where `cp_IP_address` and `cp_port_number` represent the current agent platform of an agent, and `bp_IP_address` and `bp_port_number` represent the birth agent platform of the agent.

Figure 5.1 shows the difference between UAA-based message passing and LAA-based message passing. Figure 5.1.a depicts how UAA-based message passing works with two mobile agents. After agents *one* and *two* are created on agent platform A, agent *one* migrates to agent platform C, and agent *two* migrates to agent platform B. Whenever these two agents communicate with each other, messages are delivered through their birth agent platform. However, with LAA-based message passing, interactions between these mobile agents can be optimized (see Figure 5.1.b). When agent *one* with LAA `C/A:15` sends its first message to agent *two*, the message will be delivered through the birth agent platform of agent *two*, because agent *one* does not know the location of agent *two*. This message includes information about the current agent platform of agent *one*. Therefore, when agent *two* receives the message, it knows the location of agent *one*, and it can now send a message directly to agent *one*. Similarly, when agent *one* receives a message from agent *two*, it knows the location of agent *two*. Finally, the two agents can communicate with each other directly without mediation by their birth agent platform A.

The FLAMP mechanism requires the following four modifications:

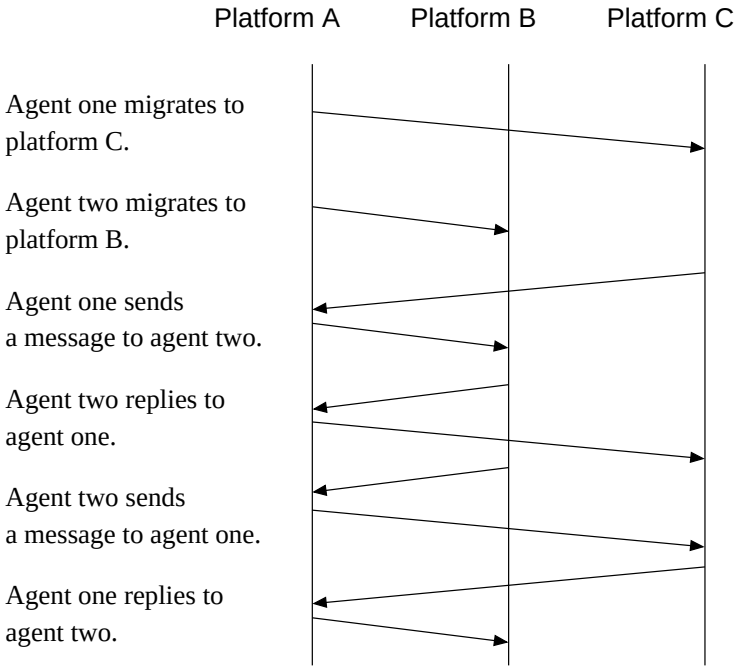
1. The address of an agent should be extended to include information about the current agent platform of the agent.

2. Whenever an agent migrates from one agent platform to another, the new agent platform of the agent should update the address of the agent.

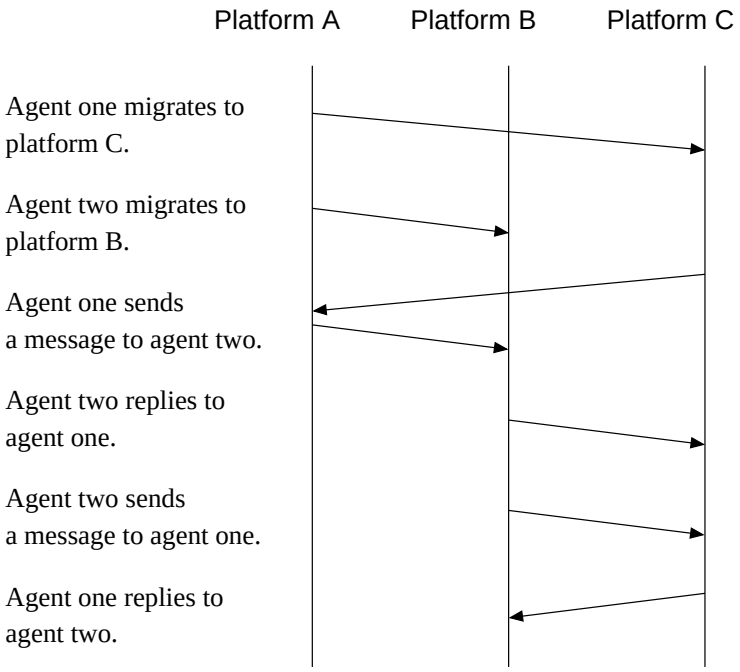
3. When the sender and receiver agents of a message are not located on the same agent platform, and the IP address of the source agent platform differs from `cp_IP_address` in the address of the receiver agent, the message will be delivered to the agent platform specified by `cp_IP_address` instead of `bp_IP_address`.

4. If an agent platform receives a message as the destination agent platform of the message, and the receiver agent has already migrated to another agent platform, the message will be delivered to the birth agent platform of the receiver agent.

This mechanism does not guarantee that every message will be delivered directly from its source agent platform to its destination agent platform. At the beginning of a communication, a message



a. UAA−based Message Passing



b. LAA−based Message Passing

Figure 5.1: Message Passing between Two Mobile Agents

should be delivered to the destination agent platform of the receiver agent through its birth agent platform if the receiver agent is not located on its birth agent platform. Also, if a sender agent can use the UAA or an outdated address of the receiver agent of a message, the message will be delivered either through its birth agent platform or through its previous and birth agent platforms. This causes a longer communication time than that in the base mechanism. However, if two agents send more than one message before one of the two agents starts another migration, the location address-based mechanism will reduce the total number of hops for continuous messages by half.

5.2.2 FLCMP: Forwarding and Location Cache-based Message Passing

The FLCMP mechanism is based on a hash table, called a *location cache*, for information about the current agent platforms of remote agents, which is managed by each agent platform. When an agent platform receives a message from a remote agent, this agent platform registers the UAA of the sender agent of the message and its current agent platform in the location cache. When an agent sends a message, its agent platform uses this location cache to find the current location of the receiver agent with the UAA of the receiver agent of the message. If the cache includes information about the location of the receiver agent, the message is delivered to the agent platform according to information retrieved from the cache. Otherwise, the message will be delivered through the birth agent platform of the receiver agent, as the base mechanism does.

For the FLCMP mechanism, four modifications are required:

1. Each agent platform manages a location cache to store information about the current agent platforms of remote agents.

2. Each agent communication message should include information about the agent platform where the sender agent is located.

3. When an agent platform receives a message from a remote agent, information about the current agent platform of the sender agent is registered in the location cache of the agent platform.

4. When the sender and receiver agents of a message are not located on the same agent platform, the message should be delivered to the agent platform retrieved from the location cache with

the address of the receiver agent. Otherwise, the message will be delivered to the birth agent platform of the receiver agent.

With these modifications, we can expect the same benefits as described in Figure 5.1.b. However, like the FLAMP mechanism, this mechanism has a problem: the receiver agent of a message knows the current agent platform of the sender agent, but the sender agent does not know the current agent platform of the receiver agent until the sender agent receives another message from this receiver agent. This problem can be solved with additional extensions:

5. When the agent platform where the receiver agent is located receives a message from a remote agent who is not located on its birth agent platform, and the message is delivered through the birth agent platform of the receiver agent, the agent platform of the receiver agent sends a platform-level message to the agent platform where the sender agent is located. This platform-level message includes information about the address of the receiver agent and its current agent platform.

6. When an agent platform receives the platform-level message from another agent platform, this agent platform registers the address of an agent and its current agent platform in the location cache.

With this modification, the sender and receiver agents will know the location of the other agent after one agent communication message and one platform-level message are delivered.[2] Figure 5.2 depicts the process flow of the first message sent from agent *one* to agent *two*. After agent *two* receives a message from agent *one*, agent platform B sends a platform-level message to agent platform C. Thus, agent *one* can send a message directly to agent *two*, while agent *two* can send a message directly to agent *one*. However, this mechanism still requires indirect message passing at the beginning of the communication.

5.2.3 ALLCMP: Agent Locating and Location Cache-based Message Passing

The ALLCMP mechanism is similar to the FLCMP mechanism, but this mechanism is based on an agent naming service instead of a message forwarding service. Before an agent platform sends a

[2]This operation may be performed by application agents with the FLAMP mechanism. However, this behavior should be implemented by application programmers. Therefore, it is no longer an automatic operation.

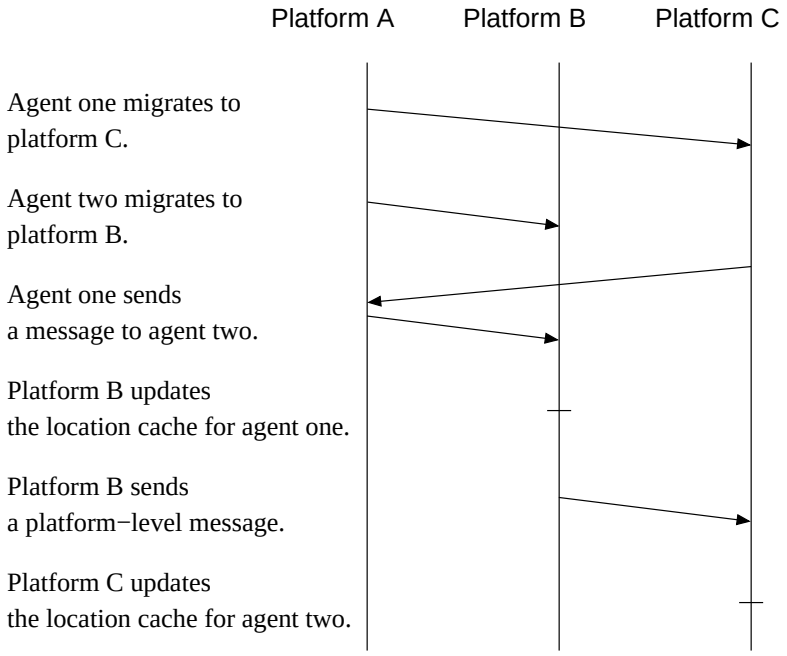


Figure 5.2: Process Flow of the First Message Sent from Agent One to Agent Two in Location Cache-based Message Passing

message to its receiver agent on a different agent platform, the agent platform checks the location of the receiver agent by communicating with the birth agent platform of the receiver agent.

The ALLCMP mechanism requires the following four modifications:

1. Each agent platform manages a location cache to store information about the current agent platforms of remote agents.

2. Each agent communication message should include information about the agent platform where the sender agent is located.

3. When an agent platform receives a message from a remote agent, information about the current agent platform of the sender agent is registered in the location cache of the agent platform.

4. When the sender and receiver agents of a message are not located on the same agent platform, the message should be delivered to the agent platform retrieved from the location cache with the address of the receiver agent. Otherwise, the source agent platform checks the location of the receiver agent by communicating with its birth agent platform before sending a message

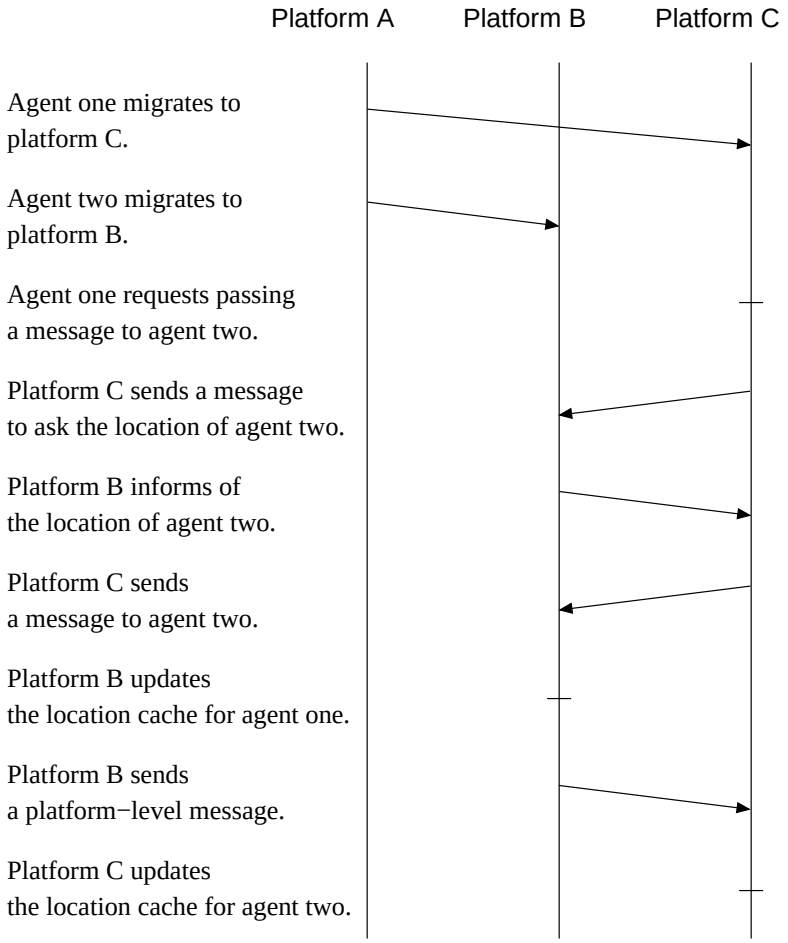


Figure 5.3: Process Flow of the First Message Sent from Agent One to Agent Two in Agent Locating and Location Cache-based Message Passing

to its current agent platform. The retrieved information is stored in the location cache of the source agent platform.

With this modification, the need for the additional extensions used in the FLCMP mechanism is reduced, because the sender agent platform knows the location of the receiver agent before it sends the message. If the sender agent platform recognizes the location of the receiver agent, it will be the latest information. However, if the location cache has outdated information about the location of the receiver agent, the message will be delivered to the destination agent platform through another agent platform, and this information will not be updated until it receives a message from the receiver agent. Therefore, this mechanism still requires the additional extensions. Figure 5.3 depicts the process flow of the first message sent from agent *one* to agent *two*.

5.3 Evaluation

The FLAMP mechanism does not require an agent platform to manage an information table about the new locations of mobile agents, unlike the FLCMP or ALLCMP mechanisms, because this information is managed by the individual agents. Also, the FLAMP mechanism does not require any additional platform-level messages to update the information table, although this mechanism still requires platform-level messages to update information about the location of an agent on its birth agent platform as its base mechanism does. Another advantage of this mechanism is that if an agent migrates to another agent platform, the agent can carry information about the locations of other agents with which this agent has communicated.

One problem of the FLAMP mechanism is that the benefit of this mechanism depends on the efforts of programmers. For example, agent *one* may have multiple LAAs of agent *two* after agent *one* has received multiple messages from agent *two*. If agent *one* uses an outdated LAA of agent *two* instead of its latest LAA, every message from agent *one* to agent *two* will be delivered through a previous agent platform and the birth agent platform of agent *two*. Therefore, agent *one* cannot benefit from location address-based message passing, and the message delivery time in this mechanism from agent *one* to agent *two* is longer than that in the base mechanism.

In the FLCMP mechanism, the sender agent platform knows the location of the receiver agent by a platform-level message, even although this platform does not receive a reply message from the receiver agent. Another benefit of the FLCMP mechanism is that this approach does not modify the address format of an agent. Therefore, agents do not need to distinguish between the UAA and LAA of a receiver agent. Also, because all agents on the same agent platform share the information cache about the locations of remote agents, one update of information about the current agent platform of a remote agent affects all communications between all local agents and the remote agent.

The ALLCMP mechanism has characteristics similar to the FLCMP mechanism. However, when the sender agent platform does not know the location of the receiver agent of a message, this mechanism takes more time to send the message. This is because the sender agent platform should communicate with the birth agent platform of the receiver agent. Even though the receiver agent is located on this birth agent platform, the agent locating operation should be executed. However,

if the size of an agent communication message is much larger than that of a platform-level message, the agent locating phrase may reduce the total communication time of a message.

Unlike the FLAMP mechanism, the FLCMP and ALLCMP mechanisms require the time for an internal operation to search the location of the receiver agent in the location cache. However, this search time is constant and negligible. Also, the FLCMP and ALLCMP mechanisms require a space for the location cache. However, the FLAMP mechanism may require more memory than the FLCMP and ALLCMP mechanisms. This is because if multiple agents on the same agent platform communicate with the same remote agent, information about the current agent platform of the remote agent is duplicated on as many multiple agents as the number of local agents using the FLAMP mechanism.

A shortcoming of the FLCMP and ALLCMP mechanisms is that the size of the information table for the locations of mobile agents increases continuously. To avoid this increase in the size of the table, an aging mechanism can be used; if information about the location of a mobile agent is not used during a certain period of time, this information is eliminated from the location cache.

The following subsections describe two important characteristics of these optimization mechanisms: reliable message passing and message delivery time.

5.3.1 Reliable Message Passing

We assume that communication networks are reliable and that every message is eventually delivered to its receiver agent if the agent is static. However, these assumptions do not guarantee that every message is eventually delivered to its receiver agent if the agent is mobile. In many agent frameworks, messages are delivered by asynchronous message passing. Thus, when an agent platform sends a sequence of messages to another agent platform, the order of the messages sent from the source agent platform may differ from the order of the messages received by the destination agent platform. (Asynchronous message passing can cause a message order shuffling problem.) For example, if an agent was created on agent platform A, and it migrated to agent platform B and then to C, agent platforms B and C will send location update messages to agent platform A. Because of asynchronous message passing, agent platform A may first receive the message sent by agent platform C and then receive the message sent by agent platform B. Thus, agent platform

A may consider agent platform C to be the current location of the mobile agent. After that, if agent platform A receives messages for the migrated agent, agent platform A delivers them to agent platform B, and agent platform B will return the messages to agent platform A. Eventually, the messages will continue to be tossed between agent platforms A and B, and they cannot reach the receiver agent.

To solve this problem, every mobile agent maintains its migration count, and every location update message includes this information. Also, when an agent platform receives a location update message, the platform stores information in the message about the location of an agent only if the information is newer than the previous information stored. This approach allows each agent platform to maintain information about the latest locations of its child mobile agents who were created on it and migrated to other agent platforms.

However, if an agent is continuously moving, information about the location of the agent will easily become outdated. Therefore, a message may not be delivered to the final receiver agent. To solve this problem, an agent can be forced to receive permission for the next migration from its birth agent platform. When its birth agent platform has some messages for this agent, the agent platform will delay sending permission for the next migration until the agent processes the messages. However, this approach may considerably delay the migration of the mobile agent, thus decreasing the performance of operations of the agent. Also, if the birth agent platform cannot reply with permission, the mobile agents created on it can no longer migrate to other agent platforms. Therefore, we do not consider message passing to be reliable for continuously migrating agents.

In this subsection, we demonstrate reliable message delivery for a mobile agent that has stopped agent migration. We also assume that all agent platforms and networks are operating properly, and that they are not crashed.

Theorem 1. With the FLAMP mechanism, every message to a mobile agent that has stopped agent migration is eventually delivered to the agent.

Proof: A message to a mobile agent may be delivered directly to the current agent platform by the LAA of the receiver agent. Otherwise, the message will eventually be delivered to the birth

agent platform of the receiver agent. This is because if the source agent platform or the agent platform that received the message from the source agent platform does not know the current agent platform of the receiver agent, the message will be delivered to the birth agent platform of the agent. If the message is delivered to the birth agent platform of its receiver agent, the message will be forwarded to another agent platform according to information stored about the location of the receiver agent. If the receiver agent is migrating or if a location update message is being delayed, the message may continue to be tossed between the birth agent platform and a previous agent platform where the agent stayed, until the birth agent platform updates information about the new location of the receiver agent. However, because information cannot be updated with an older location on the birth agent platform of the agent, and the location update message from the current agent platform of the receiver agent will eventually be delivered to the birth agent platform, the birth agent platform eventually knows the current agent platform of the receiver agent. Finally, the birth agent platform can send the message to the current agent platform of the receiver agent, and the receiver agent eventually receives the message. □

For the FLCMP and ALLCMP mechanisms, we assume that each agent platform updates its location cache when it receives a message from another agent platform or when it receives a location update message.

Theorem 2. With the FLCMP mechanism, every message to a mobile agent that has stopped agent migration is eventually delivered to the agent.

Proof: If the receiver agent did not send any message to others and migrated through several agent platforms, the FLCMP mechanism would work the same as the FMP mechanism. If the receiver agent communicated with other agents on the agent platforms where the agent was located, the agent platforms would have a shortcut to the current agent platform of the receiver agent. If one of these agent platforms receives a message for the mobile agent, the message will eventually be delivered to the current agent platform of the receiver agent according to their location caches. If an agent platform receives a message for this mobile agent, the message will be delivered to

the birth agent platform of the agent. If the birth agent platform of the receiver agent receives a message for the agent and this platform has information about the current agent platform of the receiver agent, the message will be delivered to the current agent platform. Otherwise, the message will be delivered to one of the agent platforms where the mobile agent was located, and then the message will eventually be delivered to the current agent platform of the receiver agent according to their location caches. □

Theorem 3. With the ALLCMP mechanism, every message to a mobile agent that has stopped agent migration is eventually delivered to the agent.

Proof: A message to a mobile agent may be delivered directly to the current agent platform of the receiver agent according to cache information about the location of the agent on the sender agent platform. Otherwise, the sender agent platform or the agent platform that received the message from the source agent platform checks the location of the receiver agent by communicating with the birth agent platform of the receiver agent. Because the birth agent platform updates information about the location of the receiver agent with a newer location and will eventually receive the location update message from the current agent platform of the receiver agent, the agent platform contacting the birth agent platform will eventually receive the current location of the receiver agent. Therefore, the message will eventually be delivered to the receiver agent. □

5.3.2 Message Delivery Time

When a mobile agent does not exist on its birth agent platform, with the FMP mechanism the time to send a message to this mobile agent is as follows:

$$T_{FMP}(m) = t_{sb}(m) + t_{bd}(m)$$

where $t_{sb}(m)$ means the time to send message m from the source agent platform of the message to the birth agent platform of the receiver agent, and $t_{bd}(m)$ is the time to send message m from the birth agent platform to the destination agent platform where the receiver agent is located.

If a mobile agent does not exist on its birth agent platform and the sender has the correct LAA

of the receiver agent, with the FLAMP mechanism, the time to send a message to a mobile agent is as follows:

$$T_{FLAMP}(m) = t_{sd}(\bar{m})$$

where $t_{sd}(m)$ means the time to send message m from the source agent platform of a message to the destination agent platform of the message. In the FLAMP mechanism, message $\bar{m}$ to be delivered has the same contents as the original message m but has the LAAs of the sender and receiver agents of the message instead of their UAAs. Therefore, message $\bar{m}$ is a little larger than message m.

If the sender agent does have the LAA of the receiver agent, the message passing time of the FLAMP mechanism is the same as that of the FMP mechanism. Also, if the sender agent has an outdated LAA of the receiver agent of a message, the message delivery time becomes longer than that of the FMP mechanism:

$$T'_{FLAMP}(m) = t_{se}(\bar{m}) + t_{eb}(\bar{m}) + t_{bd}(\bar{m})$$

where $t_{se}(m)$ means the time to send message m from the source agent platform of a message to an ex-destination agent platform where the receiver agent was located, and $t_{eb}(m)$ means the time to send message m from the ex-destination agent platform to the birth agent platform of the receiver agent of the message. The second and third terms in the above equation are overheads of this mechanism when the sender agent has an outdated LAA of the receiver agent. Also, until this sender agent receives a message from this receiver agent, this outdated LAA cannot be corrected.

With the FLCMP mechanism, if the location cache has information about the location of a receiver agent, the time to send a message to the mobile agent is as follows:

$$T_{FLCMP}(m) = t_{sd}(m).$$

The message in the FLCMP mechanism is the same as the message in the FMP mechanism. Like the FLAMP mechanism, if the location cache has outdated information about a remote agent, the message delivery time for this message becomes longer than that of the FMP mechanism:

$$T'_{FLCMP}(m) = t_{se}(m) + t_{eb}(m) + t_{bd}(m) + t_{ds}(m')$$

where $t_{ds}(m')$ means the time to send platform-level message m' from the destination agent platform to the source agent platform. Platform-level message m' causes the destination agent platform to update information about the current agent platform of the receiver agent on the location cache.

Therefore, in the FLCMP mechanism, the outdated information on the location cache is easily corrected, compared with the updated LAA of the receiver agent in the FLAMP mechanism. Also, because multiple agents share information about the current location of a remote agent, the outdated information about the remote agent can be more promptly updated.

In the ALMP mechanism, an agent uses an agent naming server to locate a receiver agent. If an agent locating service is used before sending a message to a remote agent, the time to send a message to the receiver agent who is not located on its birth agent platform is as follows:

$$T_{ALMP}(m) = t_{sb}(\dot{m}) + t_{bs}(\ddot{m}) + t_{sd}(m)$$

where $t_{bs}(m)$ means the time to send message m from the birth agent agent platform to the source agent platform, $\dot{m}$ is a request message to receive information about the location of the receiver agent, and $\ddot{m}$ is the replay message of $\dot{m}$.

With the ALLCMP mechanism, if the location cache has information about the location of the receiver agent, the message delivery time is as follows:

$$T_{ALLCMP}(m) = t_{sd}(m).$$

When the location cache in the ALLCMP mechanism has outdated information about a remote agent, the message delivery time becomes longer as follows:

$$T'_{ALLCMP}(m) = t_{se}(m) + t_{eb}(\dot{m}) + t_{be}(\ddot{m}) + t_{ed}(m) + t_{ds}(m')$$

where $t_{be}(m)$ means the time to send message m from the birth agent platform of the receiver agent to ex-destination agent platform e where the receiver agent was located, and $t_{ed}(m)$ is the time to send message m from ex-destination agent platform e to the destination agent platform where the receiver agent is located.

Name	Message Delivery Time
FMP	$T_{FMP}(m) = t_{sb}(m) + t_{bd}(m)$
FLAMP	$T_{FLAMP}(m) = t_{sd}(\bar{m})$ if the sender agent knows the location address of the receiver agent $T'_{FLAMP}(m) = t_{sb}(\bar{m}) + t_{bd}(\bar{m})$ if the sender agent does not know the location address of the receiver agent $T''_{FLAMP}(m) = t_{se}(\bar{m}) + t_{eb}(\bar{m}) + t_{bd}(\bar{m})$ if the sender agent has an outdated location address of the receiver agent
FLCMP	$T_{FLCMP}(m) = t_{sd}(m)$ if the sender agent platform has cache information about the location of the receiver agent $T'_{FLCMP}(m) = t_{sb}(m) + t_{bd}(m)$ if the sender agent platform does not have cache information about the location of the receiver agent $T''_{FLCMP}(m) = t_{se}(m) + t_{eb}(m) + t_{bd}(m) + t_{ds}(m')$ if the sender agent platform has outdated cache information about the location of the receiver agent
ALMP	$T_{ALMP}(m) = t_{sb}(\dot{m}) + t_{bs}(\ddot{m}) + t_{sd}(m)$
ALLCMP	$T_{ALLCMP}(m) = t_{sd}(m)$ if the sender agent platform has cache information about the location of the receiver agent $T'_{ALLCMP}(m) = t_{sb}(\dot{m}) + t_{bs}(\ddot{m}) + t_{sd}(m)$ if the sender agent platform does not have cache information about the location of the receiver agent $T''_{ALLCMP}(m) = t_{se}(m) + t_{eb}(\dot{m}) + t_{be}(\ddot{m}) + t_{ed}(m) + t_{ds}(m')$ if the sender agent platform has outdated cache information about the location of the receiver agent

m: an agent message
$\bar{m}$: an agent message including LAA instead of UAA
m': a platform-level message to update a location cache
$\dot{m}$: a platform-level message to ask the location of an agent
$\ddot{m}$: a platform-level message to tell the location of an agent
$t_{xy}(m)$: the time to send message m from agent platform x to agent platform y (where b = birth agent platform, d = destination agent platform, e = ex-destination agent platform, and s = source agent platform)

Table 5.1: Message Delivery Time for a Mobile Agent That Is Not Located on Its Agent Platform.

ALLCMP requires more message hops than FLAMP and FLCMP. However, when the size of an agent message is relatively large compared with that of a platform message, that is, $m \gg (\dot{m} + \ddot{m})$, ALLCMP may be more efficient.

Table 5.1 summarizes the delivery time of a message for every mechanism. The equations in table 5.1 assume that the mobile agent does not migrate while the message is being delivered. If the receiver agent of a message migrates several times while the message is delivered, each message passing mechanism may take more time than that described in table 5.1.

Table 5.1 can be used to evaluate and compare message passing mechanisms on an agent system with known agent migration and message properties. For example, we assume that all agents are

mobile, that all agents are distributed evenly on n agent platforms, that agents created on an agent platform are distributed on n agent platforms at any time, that the probability of message passing from one agent to another is the same, and that the direct message passing time of a message is the same. With these assumptions, local communication occurs with the probability of $\frac{1}{n}$, messages in our base agent framework are delivered directly from their source agent platforms to their destination agent platforms with the probability of $\frac{n-1}{n} \times \frac{2}{n}$,[3] and with the probability of $\frac{n-1}{n} \times \frac{n-2}{n}$, messages are delivered from source agent platforms to destination agent platforms through the birth agent platforms of receiver agents:

$$MT = \frac{1}{n} \times LC + \frac{(n-1) \times 2}{n^2} \times DC + \frac{(n-1) \times (n-2)}{n^2} \times IC$$

where MT means the message delivery time of a message from source agent platform s to destination agent platform d, DC means direct message delivery time, and IC means message delivery time through the birth agent platform when the receiver agent is located neither on its source agent platform nor on its birth agent platform.

If the source agent platform always knows the location of the receiver agent, the message delivery time of a message becomes

$$MT_{OPTIMAL} = \frac{1}{n} \times LC + \frac{n-1}{n} \times DC.$$

When a forwarding mechanism is used, IC requires $2 \times DC$, and thus, the message delivery time of a message becomes

$$\begin{aligned} MT_{FMP} &= \frac{1}{n} \times LC + \frac{(n-1) \times 2}{n^2} \times DC + \frac{(n-1) \times (n-2)}{n^2} \times IC \\ &= \frac{1}{n} \times LC + \frac{2 \times (n-1)^2}{n^2} \times DC. \end{aligned}$$

When an agent locating mechanism is used, the source agent platform of a message should check the location of the receiver agent, even though the receiver agent is located on its birth agent

[3] When agent platform p receives a message from source agent platform s, $\frac{1}{n}$ of agents on agent platform p have come from agent platform s, and $\frac{1}{n}$ of agents on the agent platform are created on agent platform p. Therefore, the probability that the receiver agent of the message is located on agent platform p is $\frac{2}{n}$.

platform. Therefore, IC requires $3 \times DC$, and the message delivery time of a message becomes

$$\begin{aligned} MT_{ALMP} = & \frac{1}{n} \times LC + \frac{(n-1)}{n^2} \times DC + \frac{(n-1) \times (n-1)}{n^2} \times IC \\ = & \frac{1}{n} \times LC + \frac{(n-1) \times (3n-2)}{n^2} \times DC. \end{aligned}$$

The above equation does not count the time to check the location of an agent that is created on agent platform s but the agent is not located on agent platform s. We ignore this value, because this value is considerably smaller when compared with other values.

When there is no outdated location information, the message delivery time of a message in FLAMP, FLCMP, and ALLCMP can be computed using the above equations.

$$\begin{aligned} MT_{FLAMP} &= \frac{1}{n} \times LC + \frac{(n-1) \times 2}{n^2} \times DC + \frac{(n-1) \times (n-2)}{n^2} \times IC' \\ &= \frac{1}{n} \times LC + \frac{(n-1) \times 2}{n^2} \times DC + \alpha \times \frac{(n-1) \times (n-2)}{n^2} \times DC + \\ &\quad (1-\alpha) \times \frac{(n-1) \times (n-2)}{n^2} \times IC \\ &= \frac{1}{n} \times LC + \frac{(n-1) \times (2n - \alpha n + 2\alpha - 2)}{n^2} \times DC. \\ MT_{FLCMP} &= MT_{FLAMP}. \\ MT_{ALLCMP} &= \frac{1}{n} \times LC + \frac{(n-1)}{n^2} \times DC + \frac{(n-1) \times (n-1)}{n^2} \times IC' \\ &= \frac{1}{n} \times LC + \frac{(n-1)}{n^2} \times DC + \alpha \times \frac{(n-1) \times (n-1)}{n^2} \times DC + \\ &\quad (1-\alpha) \times \frac{(n-1) \times (n-1)}{n^2} \times IC \\ &= \frac{1}{n} \times LC + \frac{(n-1) \times (3n - 2\alpha n + 2\alpha - 2)}{n^2} \times DC. \end{aligned}$$

where IC' means message delivery time through an ex-agent platform and the birth agent platform of the receiver agent, and α means the probability that either the sender agent of a message or the source agent platform has location information about the receiver agent. In the above equation, we do not consider the case that either the sender agent of a message or the source agent platform has outdated location information about the receiver agent.

Table 5.2 shows the coefficient of the internode communication cost (DC) for the delivery of a message according to the given parameters n and α. As the value of n increases, the coefficient values of OPTIMAL, FMP, and ALMP converge as 1, 2, and 3. Also, when the value of α is small, the coefficient values of FLAMP, FLCMP, and ALLCMP are close to the coefficient value of

n	OPTIMAL	FMP	FLAMP	FLCMP	ALMP	ALLCMP
10	0.900	1.620	0.972	0.972	2.520	1.062
10^2	0.990	1.960	1.087	1.087	2.950	1.186
10^3	0.999	1.996	1.099	1.098	2.995	1.199
...	...	...	...	...	...	...
∞	1	2	1.1	1.1	3	1.2

a. Coefficient of the Internode Communication Cost When $\alpha = 0.9$.

n	OPTIMAL	FMP	FLAMP	FLCMP	ALMP	ALLCMP
10	0.900	1.620	1.260	1.260	2.520	1.710
10^2	0.990	1.960	1.475	1.475	2.950	1.970
10^3	0.999	1.996	1.498	1.498	2.995	1.997
...	...	...	...	...	...	...
∞	1	2	1.5	1.5	3	2

b. Coefficient of the Internode Communication Cost When $\alpha = 0.5$

n	OPTIMAL	FMP	FLAMP	FLCMP	ALMP	ALLCMP
10	0.900	1.620	0.548	0.548	2.520	2.358
10^2	0.990	1.960	1.863	1.863	2.950	2.754
10^3	0.999	1.996	1.896	1.896	2.995	2.795
...	...	...	...	...	...	...
∞	1	2	1.9	1.9	3	2.8

c. Coefficient of the Internode Communication Cost When $\alpha = 0.1$

Table 5.2: Coefficient of the Internode Communication Cost (DC) for the Delivery of a Message.

OPTIMAL. However, when the value of α is large, the coefficient values of FLAMP, FLCMP, and ALLCMP are close to those of their base message passing.

5.4 Experimental Results

We evaluated five message passing mechanisms on our actor-based multi-agent framework called the *Adaptive Actor Architecture* (AAA) (see chapter 2 and [64, 66]), an extension of the *Actor Architecture* (AA) [63, 65, 68]. AAA was implemented in Java language and the communication among agent platforms is performed by TCP [87]. For these experiments, we used four computers (3.4 GHz Intel CPU and 2 GB main memory) connected by a Giga-bit switch.

Figure 5.4 depicts the runtime of each mechanism for the same random walk simulation. For these experiments, 100 agents were distributed on four computers. Each agent knew all other agents and sent messages to others that were randomly selected. After an agent sent 1,000 messages, this

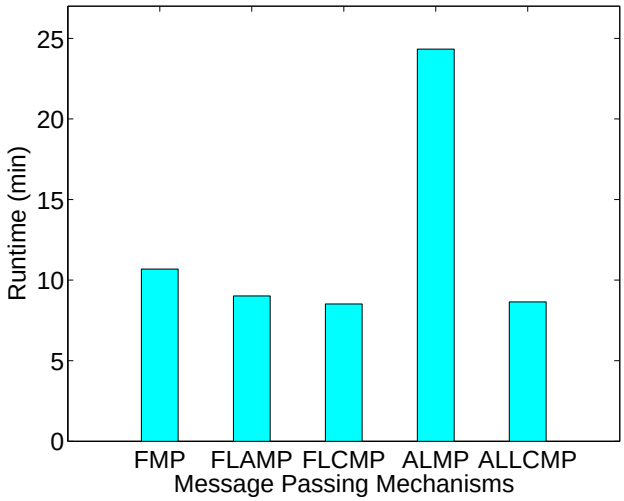


Figure 5.4: Runtimes of Random Walk Simulations According to Message Passing Mechanisms (100 agents, 10,000 messages per agent, and agent migration after 1,000 messages)

agent migrated to another agent platform. After around 1,000,000 messages were delivered to others, the simulation ended. To activate an action to sent a message to another agent, each agent sends a message to itself. Also, an agent migrated as a message, and each simulation had about 1,000 agent migrations. Therefore, the minimum number of messages on the entire system was 2,001,000. To check the runtime of each simulation, the same program was executed five times under the same initial parameters, and the median of their runtime values was selected.

This experiment showed that all the optimization mechanisms were better than their base mechanisms. Specifically, the FLCMP mechanism provided the best performance. This performance ratio was related more to the total size of the inter-node messages than to that of the total messages or the number of messages (see Figures 5.5 and 5.6). Figure 5.7 depicts the normalized ratio of the number and the size of inter-node messages by the runtime of each simulation.

If agents moved more frequently to other agent platforms, the optimization mechanisms performed more poorly than their base mechanisms (see Figure 5.8). Because each agent always carried the agent names of others, the size of an agent to be migrated in the FLAMP mechanism was larger than that in others. Therefore, the runtime of the FLAMP mechanism was considerably longer than that of the FLCMP mechanism.

We also conducted UAV (Unmanned Aerial Vehicle) simulations with three optimization mechanisms (see Figure 5.9). In these simulations, micro UAVs performed a surveillance mission on

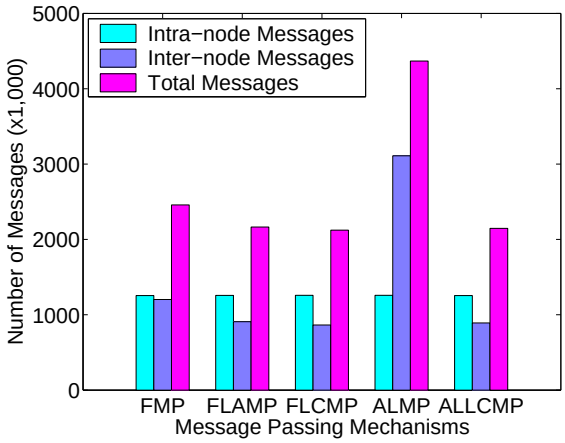


Figure 5.5: The Number of Messages Passed in Random Walk Simulations

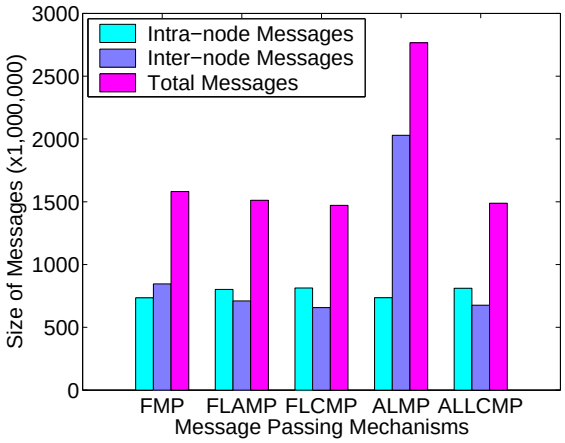


Figure 5.6: The Size of Messages Passed in Random Walk Simulations

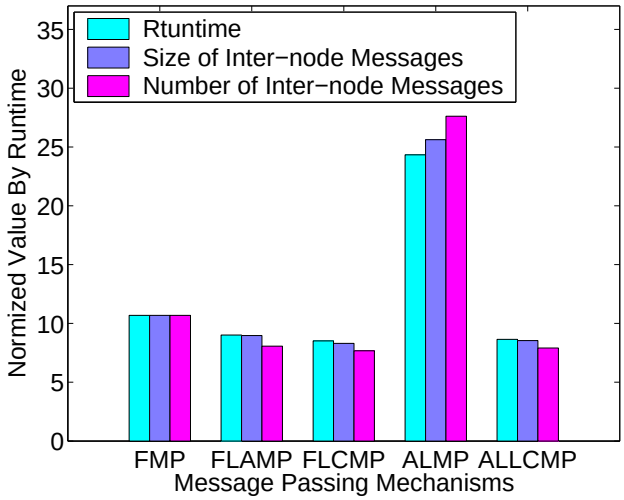


Figure 5.7: The Normalized Values of the Number and the Size of Inter-node Messages by the Runtime of Each Simulation

a predefined mission area by collaborating with each other. When a UAV detected a target, the UAV approached the target to investigate it. Detailed information about our UAV simulations is given in [67, 69]. In each simulation, half of the agents were UAVs and the others were targets. The experimental results showed that the FLCMP and ALLCMP mechanisms provided similar performance but that the FLCMP mechanism was slightly better than the ALLCMP mechanism. Also, both the FLCMP and ALLCMP mechanisms provided considerably better performance than the FLAMP mechanism.

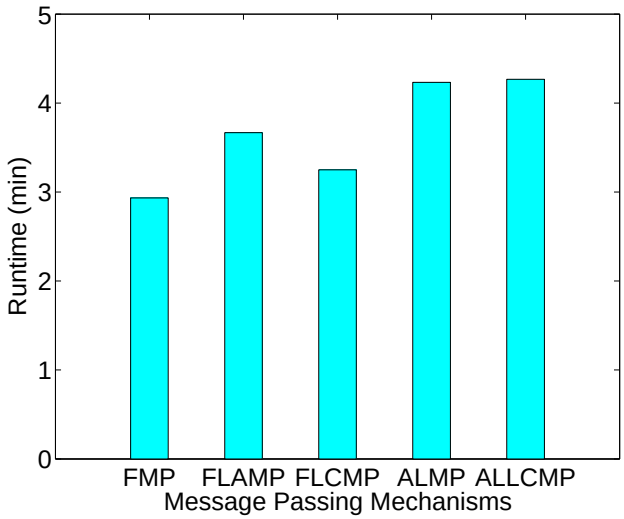


Figure 5.8: Runtimes of Random Walk Simulations According to Agent Addressing Mechanisms (100 agents, 1,000 messages per agent, and agent migration after 10 messages)

5.5 Related Work

In multi-agent frameworks that support mobile agents, the efficiency of message passing for mobile agents greatly affects the performance of agent communication. Message passing mechanisms for mobile agents can be broadly categorized into *forwarding* and *agent locating* mechanisms. In this chapter, we considered these two base mechanisms and three optimization mechanisms: FMP, FLAMP, FLCMP, ALMP, and ALLCMP. Alouf et al. [6] also evaluated two base mechanisms. However, the authors considered a single centralized server for an agent locating service instead of distributed servers. The service mechanism of the centralized server is closed to the ALMP mechanism, but this mechanism is based on synchronous communication.

General forwarding mechanisms are used in *JMAS* [25] and *JavAct* [9]. In this thesis, we describe a modified version of the general forwarding mechanism as the base mechanism of FLAMP and FLCMP to mitigate a shortcoming of the general mechanism. FLAMP is used in AA (Actor Architecture) [65] and early versions of AAA (Adaptive Actor Architecture) [66]. FLCMP is used in the current version of AAA.

ALMP is used in SALSA [111]. In SALSA, *UAL* (*Universal Actor Locator*) represents the location of an actor, and a middle agent called the *Universal Actor Naming Server* manages the UALs of actors and locates the receiver actor. This mechanism requires the receiver agent to

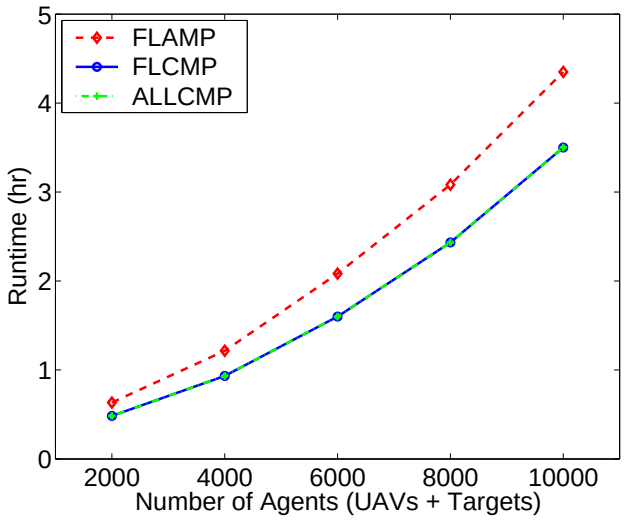


Figure 5.9: Runtime of UAV Simulations

register its location at a certain middle agent, and the middle agent can be a bottleneck in message passing. The message passing mechanism used in FIPA [45] is also ALMP. From the given name of the receiver agent of a message, the AMS (Agent Management System) of the sender agent platform resolves the transport address of the receiver agent at which the receiver agent can be contacted. This name-resolving process requires platform-level communication with AMSs on other agent platforms. Although FIPA does not mention its optimization mechanism, FIPA-compliant frameworks can support an optimization mechanism such as ALLCMP.

The location cache-based message passing mechanism (FLCMP and ALLCMP) is similar to the route optimization mechanism used in *Mobile IP* [86]. Although the domain of Mobile IP differs considerably from that of a mobile agent, the optimization mechanism used for Mobile IP can be applied to mobile agents. This is because message passing in Mobile IP is quite similar to message passing for mobile agents: the birth agent platform of an agent works as a *home agent* in Mobile IP, and the current agent platform works as a *foreign agent* in Mobile IP. In multi-agent systems, whenever an agent changes its location, its new agent platform should report information about its new location to its birth agent platform. In Mobile IP, whenever a mobile node changes its area, its new foreign agent should report information about its new location to its home agent. While a message in multi-agent systems is delivered to the receiver mobile agent through its birth agent platform, a message in Mobile IP is delivered to the mobile node through its home agent. Both

have a *triangle routing* problem [86] and solve this problem using a binding cache.

Stefano and Santoro [98] proposed an agent locating mechanism called the *Search-by-Path-Chase* (SPC) mechanism, and compared the mechanism with *database logging* (or ALMP) and *path proxies* (or FMP). The proposed mechanism is a hybrid style of forwarders and disturbed servers, and uses a location cache to reduce additional hops in message passing for mobile agents.

5.6 Discussion

This chapter describes two base message passing mechanisms and three optimization mechanisms for mobile agents: FMP (forwarding-based message passing), FLAMP (forwarding and location address-based message passing), FLCMP (forwarding and location cache-based message passing), ALMP (agent locating-based message passing), and ALLCMP (agent locating and location cache-based message passing). Optimization mechanisms attempt to send a message directly to the destination agent platform where the receiver agent is located. Also, they are more fault tolerant; even though the birth agent platform of the receiver agent is temporarily disconnected or does not operate, optimization mechanisms work properly.

Our experimental results show that when agents communicate intensively with each other, the FLCMP and ALLCMP mechanisms provide better performance than others. However, the performance of these mechanisms depends on the properties of a given application, especially, the ratio of agent communication to agent migration. To investigate the characteristics of each mechanism, more experiments need to be conducted with various cases. For example, if an aging mechanism is applied to remove the old history of the locations of mobile agents, the performance of the FLCMP mechanism might differ considerably from that of the ALLCMP mechanism.

Chapter 6

Operational Semantics

6.1 Introduction

The proposed services were developed and evaluated on our actor-based multi-agent framework called the *Adaptive Actor Architecture* (*AAA*) (see chapter 2 and [64, 66]). Therefore, agents in our multi-agent framework have the characteristics of actors, and the behaviors of agents and agent platforms can be explained by actor semantics. Agha et al. [5] formulated the operational semantics for actor systems using a transition relation on *configurations*. Here a configuration means a snapshot of an actor system, and it includes actors, messages, the names of internal actors known by external actors (called *receptionists*), and the names of external actors known by internal actors. This operational semantics formally defines the meaning of operators used in the actor model, shows the ability to merge actor systems into a large system, and is used to test equivalence on actor systems.

We modified this operational semantics to describe the behavior of an actor platform and the procedure of message passing between actor platforms more specifically. This modification includes a new definition of the configuration to express the scope of an actor system more clearly and information about the locations (or actor platforms) of actors to describe actor mobility. In this chapter, we also show how our fundamental operational semantics can be extended to describe different message passing schemes and how this semantics can be used to check the problems of a given operational semantics.

6.2 Operational Semantics

6.2.1 Actor Systems

We use a configuration to present a snapshot of an actor system that includes one actor platform and actors on the platform. A configuration also includes the location identification of an actor platform as well as actors and messages on the platform. In addition, a configuration has a list of names of all *local actors* that are executing on the actor platform. Because an actor platform supports mobile actors, a configuration has a map to change the names of *remote actors* that were created on the actor platform and moved to their current locations (or actor platforms).

The name list of local actors is similar to a receptionist in [5]. However, the receptionist does not include the names of all the actors on the actor platform. For example, when an actor is created locally, the name of this actor exists on the list of local actors, but the receptionist does not include this name. The external actor names in [5] are not utilized by actor platforms, because an actor platform delivers a message to another platform according to information about the location of its receiver actor stored in its name. Therefore, a new definition of an actor configuration is specified as follows.

Definition (Actor Configuration) An *actor configuration* with an actor map α, a set of messages μ, its location identification l, a set of names of local actors $\mathcal{N}$, and an actor location map $\mathcal{L}$ is written

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

where α maps a finite set of actor names to their behaviors, μ is a finite set of messages, l is the identification of this actor platform, $\mathcal{N}$ is a finite set of names of local actors on this actor platform, and $\mathcal{L}$ maps a finite set of the names of remote actors to their current locations. (Note that α manages actors that were originally created at actor platform l but have migrated to other actor platforms.)

The state of an actor in a configuration is represented by one of the following forms:

- $(b)_a$: ready to accept a message, where b is the behavior of the actor a.

- $[e]_a$: busy executing e, where e is the current processing state of the actor a.

Actor primitives (i.e., `create`, `become`, and `send`) are described by the transition relation on actor configurations. The single-step transition relation $\mapsto$ on configurations occurs by the following definition. Reduction rules for the single-step transition relation $\overset{\lambda}{\mapsto}_X$ on functional redexes are given in [5], where X is a finite set of actor names. To describe the internal operations of actor platforms, `receive`, `in`, `out`, `ready`, and `die` are added in this transition relation. The variables and functions used in this definition have the following meanings:

Notation:

- e: an internal expression (or operation) of an actor except actor primitives (i.e., `create`, `become`, and `send`).
- R: the reduction context in which the expression currently being evaluated occurs.
- $\mathrm{FV}(\alpha(a))$: free variables of actor a.
- $\texttt{<}a \Leftarrow v\texttt{>}$: a message on an actor platform; its receiver actor is a, and its content is v.
- $\texttt{app}(b, v)$: a method invocation; the method in behavior b related to message v is executed.
- $[l : m]$: a message on the Internet; its destination location (or actor platform) is l, and its content is m.
- $\texttt{loc}(a)$: the location (or identification) of the current platform of actor a.

Definition ($\mapsto$):

`<fun>`

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow \left\langle\!\!\left\langle \alpha, [e]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [e']_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

`<create>`

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{create}(b')]]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [R[a']]_a, (b')_{a'} \mid \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

where a' is fresh and $\mathcal{N}' = \mathcal{N} \cup \{a'\}$

<become>

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{become}(b')]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_{a'}, (b')_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

where a' is fresh

<send>

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{send}(a', v)]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_a \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

where $a' \in \mathrm{FV}(\alpha(a))$ and $m = \texttt{<}a' \Leftarrow v\texttt{>}$

<receive>

$$\left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[app}(b, v)\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $a \in \mathcal{N}$

<out>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; m, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $a \notin \mathcal{N}$

where $m = \texttt{<}a \Leftarrow v\texttt{>}$ and $\texttt{loc}(a) = l'$

<in>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

where $m = \texttt{<}a \Leftarrow v\texttt{>}$

<ready>

$$\left\langle\!\!\left\langle \alpha, \texttt{[nil]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $a \in \mathcal{N}$

<die>

$$\left\langle\!\!\left\langle \alpha, \texttt{[nil]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $a \notin \mathcal{N}$

After the transition of <out> and before the transition of <in>, it is assumed that message m is on the networks. When actor a finishes its current expression (or method), that is, e becomes nil, this actor may die. Therefore, in [5], actors have to reactivate their behaviors with the become or ready primitive to remain active and be ready for another message. These primitives have to carry the new behavior of the actor and may take some state variables. However, in many cases, the new behavior of an agent is the same as its previous behavior. Therefore, in our semantics,

we assume that after an actor finishes its current expression, it becomes the ready state with its current behavior and variables unless the `become` primitive is explicitly called. However, if the actor is a temporary actor, that is, if its name does not exist in $\mathcal{N}$ of its current actor platform, the actor is automatically removed by the actor platform.

In applying these transition rules to given programs, we can understand how actor platforms and actors work together. The following transitions show how message passing from actor platform l to l' is processed.

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{send}(a', v)]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} + \left\langle\!\!\left\langle \alpha', (b')_{a'} \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_a \;\middle|\; \mu, \texttt{<}a' \Leftarrow v\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} + \left\langle\!\!\left\langle \alpha', (b')_{a'} \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} + \left\langle\!\!\left\langle \alpha', (b')_{a'} \;\middle|\; \mu', \texttt{<}a' \Leftarrow v\texttt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} + \left\langle\!\!\left\langle \alpha', \texttt{[app}(b', v)\texttt{]}_{a'} \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

where $A + B$ means the composition of actor platforms A and B acting in parallel. Note that the transition between the sender actor and the receiver actor is asynchronous, but we do not model the networks.

6.2.2 Actor Mobility

In the basic transition rules defined in the previous subsection, $\mathcal{L}$ is not changed because the definition does not explain actor mobility. This variable is used to describe the locations of remote actors that were created on this actor platform and migrated to other actor platforms. To describe actor mobility, in addition to $\mathcal{L}$, the basic transition rules are extended with the following definition. The complete transition rules used to describe actor mobility appear in Appendix B.2.1. The new symbols used in the extended actor configurations have the following meaning:

Notation:

- $[(b)_a]$: a serialized form of actor a with behavior b.
- $\texttt{origin\_loc}(a)$: the location (or identification) of the birth actor platform where actor a was created.

Definition ($\mapsto$):

In the following transition rules, we assume that $l \neq l'$.

<migrate>

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{migrate}(l)]]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [R[\texttt{nil}]]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{migrate}(l')]]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [R[\texttt{nil}]]_{a'} \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}'}$$

if $l = \texttt{origin\_loc}(a)$

where a' is fresh, $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$, $\mathcal{N}' = \mathcal{N} - \{a\}$, and $\mathcal{L}' = \mathcal{L} \cup \{[a, l']\}$

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{migrate}(l')]]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [R[\texttt{nil}]]_{a'} \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

if $l \neq \texttt{origin\_loc}(a)$

where a' is fresh, $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$, and $\mathcal{N}' = \mathcal{N} - \{a\}$

<receive>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_a]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [a,l']} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

if $l = \texttt{origin\_loc}(a)$

where $\mathcal{N}' = \mathcal{N} \cup \{a\}$

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_a]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; m, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

if $l \neq \texttt{origin\_loc}(a)$ and $l' = \texttt{origin\_loc}(a)$

where $\mathcal{N}' = \mathcal{N} \cup \{a\}$ and $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [a, l']\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [a,l'']} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [a,l']}$$

<out>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; m, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$ or $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$

<in>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

where $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$ or $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$

With these transition rules, actor migration from actor platform l to l' is described using the following transitions. The process whereby a temporary actor becomes `nil` is expressed by a multi-step transition, $\xrightarrow{L}$.

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{migrate}(l')]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_{a'} \;\middle|\; \mu, \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\xrightarrow{L} \left\langle\!\!\left\langle \alpha, \texttt{[nil]}_{a'} \;\middle|\; \mu, \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu', \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha', (b)_a \;\middle|\; \mu', \texttt{<}l \Leftarrow [a, l']\texttt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ a \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \texttt{<}l \Leftarrow [a, l']\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha', (b)_a \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}',\ a \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha', (b)_a \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}',\ a \mid \mathcal{L}'}$$

6.2.3 Location Address-based Message Passing

In location address-based message passing, an actor name a consists of the location identification l of its birth actor platform and the local identification i of the actor: $a = [l + i]$. The function $\texttt{loc}(a)$ is defined as follows:

$$\texttt{loc}(a) = \begin{cases} l' & \text{if } [a, l'] \in \mathcal{L} \\ l & \text{otherwise} \end{cases}$$

When location address-based message passing is used, and the receiver actor of a message does not exist on the actor platform of the sender actor, the actor platform must send the message to the birth actor platform of the receiver actor even though the receiver actor does not exist on the platform. This is because only the birth actor platform knows the current actor platform of the receiver actor. The location address-based message passing scheme mitigates this problem.

In location address-based message passing, an actor name a consists of the location identification l' of its current actor platform, the location identification l of its birth actor platform, and the local identification i of the actor: $a = [l' + l + i]$. Even though the actor naming scheme is extended, $\mathcal{N}$ and $\mathcal{L}$ in the actor configurations continuously use universal actor addresses instead of location-based actor addresses. Therefore, the function $\texttt{loc}(a)$ is defined as follows:

$$\texttt{loc}(a) = \begin{cases} l'' & \text{if } [\texttt{uaa}(a), l''] \in \mathcal{L} \quad \text{where } \texttt{uaa}(a) = [l + i] \\ l' & \text{otherwise} \end{cases}$$

In both message passing schemes, the location identification of the birth actor platform of an actor and the local identification of the actor can be achieved with the following functions:

$\texttt{origin\_loc}(a) = l$

$\texttt{id}(a) = i$

When an actor arrives from another actor platform, and the identification of the previous actor platform is retrieved by $\texttt{previous\_loc}(a)$.

$\texttt{previous\_loc}(a) = l' \quad$ where $a = [l' + l + i]$

To support location address-based message passing, some of the previous rules should be replaced. In the following definition, the same transition rules under one label as the rules defined in the previous subsection have been omitted. The complete transition rules for location address-based message passing appear in Appendix B.2.2.

Definition ($\mapsto$):

<receive>

$$\left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \texttt{<}a' \Leftarrow v\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[app}(b, v)\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_{a'}]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [\texttt{uaa}(a), l']} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$
if $\texttt{uaa}(a) \in \mathcal{N}$ and $\texttt{uaa}(a) = \texttt{uaa}(a')$

if $l = \texttt{origin\_loc}(a)$ and $\texttt{uaa}(a) = \texttt{uaa}(a')$

where $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a)\}$

($a = [l + l + i]$ and $a' = [l' + l + i]$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_{a'}]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$
if $l \neq \texttt{origin\_loc}(a)$, $l' = \texttt{origin\_loc}(a)$, and $l' = \texttt{previous\_loc}(a)$

where $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a)\}$, and $\texttt{uaa}(a) = \texttt{uaa}(a')$

($a = [l + l' + i]$, $a' = [l'' + l' + i]$, and $l' = l''$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_{a'}]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; m, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$
if $l \neq \texttt{origin\_loc}(a)$, $l' = \texttt{origin\_loc}(a)$, and $l' \neq \texttt{previous\_loc}(a)$

where $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a)\}$, $\texttt{uaa}(a) = \texttt{uaa}(a')$, $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$,

($a = [l + l' + i]$, $a' = [l'' + l' + i]$, and $l' \neq l''$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [a', l']\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [\texttt{uaa}(a), l'']} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [\texttt{uaa}(a'), l']}$$
if $\texttt{uaa}(a) = \texttt{uaa}(a')$

($a = [l'' + l + i]$ and $a' = [l' + l + i]$)

As an example, the following transitions show actor migration from actor platform l to l'.

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{migrate}(l')]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_{a'} \;\middle|\; \mu, \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} - \{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a), l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha, \texttt{[nil]}_{a'} \;\middle|\; \mu, \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} - \{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a), l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} - \{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a), l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} - \{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a), l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu', \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} - \{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a), l']} + \left\langle\!\!\left\langle \alpha', (b)_a \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ \texttt{uaa}(a) \mid \mathcal{L}'}$$

6.2.4 Delayed Message Passing

With the preceding transitions, we can find a problem related to the message delivery for moving actors. For example, this problem happens when one actor platform sends a message to the actor platform where the receiver actor was created but the receiver actor is still moving. The following example shows the case in which a message is continuously passed between the birth and the previous actor platforms of the receiver actor of the message until the receiver actor finishes its migration.

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{migrate}(l')]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_{a'} \;\middle|\; \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha, \texttt{[nil]}_{a'} \;\middle|\; \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu', \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \texttt{<}a \Leftarrow v\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu', \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu', \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \texttt{<}a \Leftarrow v\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu', \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

...

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \texttt{<}a \Leftarrow v\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu', \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \texttt{<}a \Leftarrow v\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l']} + \left\langle\!\!\left\langle \alpha', (b)_{a'} \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ \texttt{uaa}(a') \mid \mathcal{L}'}$$

where $a = [l + l + i]$, $a' = [l' + l + i]$, and $\texttt{uaa}(a) = \texttt{uaa}(a')$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\mathtt{uaa}(a)\} \mid \mathcal{L},\, [\mathtt{uaa}(a),l']} + \left\langle\!\!\left\langle \alpha', (b)_{a'} \;\middle|\; \mu', \mathtt{<}a \Leftarrow v\mathtt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ \mathtt{uaa}(a') \mid \mathcal{L}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\mathtt{uaa}(a)\} \mid \mathcal{L},\, [\mathtt{uaa}(a),l']} + \left\langle\!\!\left\langle \alpha', \mathtt{[app}(b,v)\mathtt{]}_{a'} \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ \mathtt{uaa}(a') \mid \mathcal{L}'}$$

As can be seen from the above transitions, the message $\mathtt{<}a \Leftarrow v\mathtt{>}$ sent from actor platform l' is continuously passed between two actor platforms, l and l', until mobile actor a finishes its migration. To avoid this situation, a buffer for the names of moving actors and a buffer for the messages for moving actors have been added into an actor platform, and transition rules for the previous message passing mechanism have been extended. The complete transition rules for the delayed message passing appear in Appendix B.2.3.

Definition (Extended Actor Configuration) An *actor configuration* with an actor map α, a set of messages μ, a set of messages for moving actors β, its location identification l, a set of names of local actors $\mathcal{N}$, an actor location map $\mathcal{L}$, and a name buffer $\mathcal{B}$ is written

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

where β is a finite set of messages for moving actors that are leaving this actor platform or that have left but do not finish migration, and $\mathcal{B}$ contains the universal names of the moving actors.

Definition ($\mapsto$):

<migrate>

$$\left\langle\!\!\left\langle \alpha, \mathtt{[}R[\mathtt{migrate}(l')]\mathtt{]}_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, \mathtt{[}R[\mathtt{nil}]\mathtt{]}_{a'} \;\middle|\; \mu, m \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}' \mid \mathcal{B}'}$$

where a' is fresh, $m = \mathtt{<}l' \Leftarrow [(b)_a]\mathtt{>}$,

$\mathcal{N}' = \mathcal{N} - \{\mathtt{uaa}(a)\}$, $\mathcal{L}' = \mathcal{L} \cup \{[\mathtt{uaa}(a), l']\}$, and $\mathcal{B}' = \mathcal{B} \cup \{\mathtt{uaa}(a)\}$

<delay>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \mathtt{<}a \Leftarrow v\mathtt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} + \mathcal{E} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta, \mathtt{<}a \Leftarrow v\mathtt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

if $\mathtt{uaa}(a) \notin \mathcal{N}$ and $\mathtt{uaa}(a) \in \mathcal{B}$

<receive>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \mathtt{<}l \Leftarrow [(b)_{a'}]\mathtt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\, [\mathtt{uaa}(a),l'] \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu, m \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L} \mid \mathcal{B}}$$

if $l = \texttt{origin\_loc}(a)$ and $l'' = \texttt{previous\_loc}(a)$, and $\texttt{uaa}(a) = \texttt{uaa}(a')$

where $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a)\}$, and $m = \texttt{<}l'' \Leftarrow [a, l]\texttt{>}$

($a = [l + l + i]$ and $a' = [l'' + l + i]$)

($l' = l''$ or $l' \neq l''$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_{a'}]\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu, m \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L} \mid \mathcal{B}}$$

if $l \neq \texttt{origin\_loc}(a')$, $l' = \texttt{origin\_loc}(a')$, and $l' = \texttt{previous\_loc}(a)$

where $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a)\}$, $\texttt{uaa}(a) = \texttt{uaa}(a')$, $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$

($a = [l + l' + i]$, $a' = [l'' + l' + i]$, and $l' = l''$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_{a'}]\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu, m_1, m_2 \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L} \mid \mathcal{B}}$$

if $l \neq \texttt{origin\_loc}(a')$, $l' = \texttt{origin\_loc}(a')$, and $l' \neq \texttt{previous\_loc}(a)$

where $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a)\}$, $\texttt{uaa}(a) = \texttt{uaa}(a')$,

$m_1 = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$, $m_2 = \texttt{<}l'' \Leftarrow [a, l]\texttt{>}$,

($a = [l + l' + i]$, $a' = [l'' + l' + i]$, and $l' \neq l''$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [a', l']\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [\texttt{uaa}(a), l''] \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [\texttt{uaa}(a'), l'] \mid \mathcal{B}'}$$

if $l = \texttt{origin\_loc}(a')$, $\texttt{uaa}(a) = \texttt{uaa}(a')$, and $\texttt{uaa}(a') \in \mathcal{B}$

where $\mathcal{B}' = \mathcal{B} - \{\texttt{uaa}(a')\}$

($a = [l'' + l + i]$ and $a' = [l' + l + i]$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [a', l']\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}'}$$

if $l \neq \texttt{origin\_loc}(a')$, $\texttt{uaa}(a) = \texttt{uaa}(a')$, and $\texttt{uaa}(a') \in \mathcal{B}$

where $\mathcal{B}' = \mathcal{B} - \{\texttt{uaa}(a')\}$

<out>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; m, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} + \mathcal{E} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

if $m = \texttt{<}a \Leftarrow v\texttt{>}$, $\texttt{uaa}(a) \notin \mathcal{N}$, and $\texttt{uaa}(a) \notin \mathcal{B}$

where $\texttt{loc}(a) = l'$

<delay-out>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; m, \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} + \mathcal{E} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, m \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

if $m = \texttt{<}a \Leftarrow v\texttt{>}$ and $\texttt{uaa}(a) \notin \mathcal{B}$

With delayed message passing, we can remove unnecessary message passing between the previous actor platform and the birth actor platform of a moving actor. The following sequence of transition rules shows the flow with delayed message passing.

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{migrate}(l')]\texttt{]}_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu' \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}' \mid \mathcal{B}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_{a'} \;\middle|\; \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l'] \mid \mathcal{B},\ \texttt{uaa}(a)}$$

$$+ \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu' \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}' \mid \mathcal{B}'}$$

$$\xrightarrow{L} \left\langle\!\!\left\langle \alpha, \texttt{[nil]}_{a'} \;\middle|\; \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l'] \mid \mathcal{B},\ \texttt{uaa}(a)}$$

$$+ \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu' \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}' \mid \mathcal{B}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l'] \mid \mathcal{B},\ \texttt{uaa}(a)}$$

$$+ \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu' \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}' \mid \mathcal{B}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l'] \mid \mathcal{B},\ \texttt{uaa}(a)}$$

$$+ \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu', \texttt{<}a \Leftarrow v\texttt{>}, \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}' \mid \mathcal{B}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \texttt{<}a \Leftarrow v\texttt{>} \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l'] \mid \mathcal{B},\ \texttt{uaa}(a)}$$

$$+ \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu', \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}' \mid \mathcal{B}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta, \texttt{<}a \Leftarrow v\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l'] \mid \mathcal{B},\ \texttt{uaa}(a)}$$

$$+ \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu', \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}' \mid \mathcal{B}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta, \texttt{<}a \Leftarrow v\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l'] \mid \mathcal{B},\ \texttt{uaa}(a)}$$

$$+ \left\langle\!\!\left\langle \alpha', (b)_{a'} \;\middle|\; \mu', \texttt{<}l \Leftarrow [a,l]\texttt{>} \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ \texttt{uaa}(a') \mid \mathcal{L}' \mid \mathcal{B}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \texttt{<}l \Leftarrow [a,l]\texttt{>} \;\middle|\; \beta, \texttt{<}a \Leftarrow v\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l'] \mid \mathcal{B},\ \texttt{uaa}(a)}$$

$$+ \left\langle\!\!\left\langle \alpha', (b)_{a'} \;\middle|\; \mu' \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ \texttt{uaa}(a') \mid \mathcal{L}' \mid \mathcal{B}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta, \texttt{<}a \Leftarrow v\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l'] \mid B} + \left\langle\!\!\left\langle \alpha', (b)_{a'} \;\middle|\; \mu' \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ \texttt{uaa}(a') \mid \mathcal{L}' \mid \mathcal{B}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \texttt{<}a \Leftarrow v\texttt{>} \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l'] \mid B} + \left\langle\!\!\left\langle \alpha', (b)_{a'} \;\middle|\; \mu' \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ \texttt{uaa}(a') \mid \mathcal{L}' \mid \mathcal{B}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l'] \mid B} + \left\langle\!\!\left\langle \alpha', (b)_{a'} \;\middle|\; \mu', \texttt{<}a \Leftarrow v\texttt{>} \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ \texttt{uaa}(a') \mid \mathcal{L}' \mid \mathcal{B}'}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{\texttt{uaa}(a)\} \mid \mathcal{L},\ [\texttt{uaa}(a),l'] \mid B} + \left\langle\!\!\left\langle \alpha', \texttt{[app}(b,v)\texttt{]}_{a'} \;\middle|\; \mu' \;\middle|\; \beta' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ \texttt{uaa}(a') \mid \mathcal{L}' \mid \mathcal{B}'}$$

6.2.5 Reliable Message Passing

The current transition rules have a serious problem related to message order shuffling on the Internet. Because the actor model is based on asynchronous message passing, a message may not reach the current actor platform of the receiver actor. The following transitions show one example. (To maintain the simplicity of our actor configurations and transition rules, we explain this problem using the configuration and transition rules defined for actor mobility.)

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{migrate}(l')]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v'\texttt{>}, \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \texttt{<}a \Leftarrow v''\texttt{>}, \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_{a'} \;\middle|\; \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v'\texttt{>}, \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \texttt{<}a \Leftarrow v''\texttt{>}, \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\xrightarrow{L} \left\langle\!\!\left\langle \alpha, \texttt{[nil]}_{a'} \;\middle|\; \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v'\texttt{>}, \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \texttt{<}a \Leftarrow v''\texttt{>}, \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \texttt{<}a \Leftarrow v'\texttt{>}, \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \texttt{<}a \Leftarrow v''\texttt{>}, \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu', \texttt{<}a \Leftarrow v'\texttt{>}, \texttt{<}l' \Leftarrow [(b)_a]\texttt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \texttt{<}a \Leftarrow v''\texttt{>}, \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha', (b)_a \;\middle|\; \mu', \texttt{<}a \Leftarrow v'\texttt{>}, \texttt{<}l \Leftarrow [a,l']\texttt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ a \mid \mathcal{L}'}$$

$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \texttt{<}a \Leftarrow v''\texttt{>}, \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha', \texttt{[app}(b, v')\texttt{]}_a \;\middle|\; \mu', \texttt{<}l \Leftarrow [a,l']\texttt{>} \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ a \mid \mathcal{L}'}$$

$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \texttt{<}a \Leftarrow v''\texttt{>}, \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \texttt{<}l \Leftarrow [a,l']\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha', \texttt{[app}(b, v')\texttt{]}_a \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ a \mid \mathcal{L}'}$$

$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \texttt{<}a \Leftarrow v''\texttt{>}, \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

We assume that message v' causes another migration of actor a to actor platform l''.

$$\xrightarrow{L} \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \texttt{<}l \Leftarrow [a,l']\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha', \texttt{[}R[\texttt{migrate}(l'')]\texttt{]}_a \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}',\ a \mid \mathcal{L}'}$$

$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \texttt{<}a \Leftarrow v''\texttt{>}, \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \mathtt{<}l \Leftarrow [a, l']\mathtt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha', \mathtt{[}R[\mathtt{nil}]\mathtt{]}_{a'} \;\middle|\; \mathtt{<}l'' \Leftarrow [(b)_a]\mathtt{>}, \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$
$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \mathtt{<}a \Leftarrow v''\mathtt{>}, \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\xrightarrow{L} \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \mathtt{<}l \Leftarrow [a, l']\mathtt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha', \mathtt{[nil]}_{a'} \;\middle|\; \mathtt{<}l'' \Leftarrow [(b)_a]\mathtt{>}, \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$
$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \mathtt{<}a \Leftarrow v''\mathtt{>}, \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \mathtt{<}l \Leftarrow [a, l']\mathtt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mathtt{<}l'' \Leftarrow [(b)_a]\mathtt{>}, \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$
$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \mathtt{<}a \Leftarrow v''\mathtt{>}, \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \mathtt{<}l \Leftarrow [a, l']\mathtt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$
$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \mathtt{<}a \Leftarrow v''\mathtt{>}, \mu'', \mathtt{<}l'' \Leftarrow [(b)_a]\mathtt{>} \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \mathtt{<}l \Leftarrow [a, l']\mathtt{>}, \mathtt{<}a \Leftarrow v''\mathtt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$
$$+ \left\langle\!\!\left\langle \alpha'' \;\middle|\; \mu'', \mathtt{<}l'' \Leftarrow [(b)_a]\mathtt{>} \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'' \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \mathtt{<}l \Leftarrow [a, l']\mathtt{>}, \mathtt{<}a \Leftarrow v''\mathtt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$
$$+ \left\langle\!\!\left\langle \alpha'', (b)_a \;\middle|\; \mu'', \mathtt{<}l \Leftarrow [a, l'']\mathtt{>} \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'',\ a \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \mathtt{<}l \Leftarrow [a, l']\mathtt{>}, \mathtt{<}a \Leftarrow v''\mathtt{>}, \mathtt{<}l \Leftarrow [a, l'']\mathtt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$
$$+ \left\langle\!\!\left\langle \alpha'', (b)_a \;\middle|\; \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'',\ a \mid \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, \mathtt{<}l \Leftarrow [a, l']\mathtt{>}, \mathtt{<}a \Leftarrow v''\mathtt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\} \mid \mathcal{L},\ [a,l'']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}' \mid \mathcal{L}'}$$

$$+ \left\langle\!\!\left\langle \alpha'', (b)_a \;\middle|\; \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'',\ a\ |\ \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, <a \Leftarrow v''> \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\}\ |\ \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}'\ |\ \mathcal{L}'} + \left\langle\!\!\left\langle \alpha'', (b)_a \;\middle|\; \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'',\ a\ |\ \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\}\ |\ \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu', <a \Leftarrow v''> \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}'\ |\ \mathcal{L}'} + \left\langle\!\!\left\langle \alpha'', (b)_a \;\middle|\; \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'',\ a\ |\ \mathcal{L}''}$$

$$\mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, <a \Leftarrow v''> \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}-\{a\}\ |\ \mathcal{L},\ [a,l']} + \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle^{l'}_{\mathcal{N}'\ |\ \mathcal{L}'} + \left\langle\!\!\left\langle \alpha'', (b)_a \;\middle|\; \mu'' \right\rangle\!\!\right\rangle^{l''}_{\mathcal{N}'',\ a\ |\ \mathcal{L}''}$$

...

The last two transitions for delivering message $<a \Leftarrow v''>$ are infinitely repeated unless actor a migrates again and the $\mathcal{L}$ on actor platform l is updated. This problem can be solved using a migration counter. Every mobile actor maintains its migration count, and every location update message includes this information. Also, when an actor platform receives a location update message, the platform stores information about the location of an actor in the message only if the information is newer than the previous information stored. This approach allows each actor platform to maintain information about the latest locations of its child mobile actors who were created on it and migrated to other actor platforms.

6.3 Discussion

This chapter describes a new definition of an actor configuration and the transition rules for fundamental actor primitives, actor mobility, location address-based message passing, and delayed message passing. Our actor configuration includes the status of an actor platform as well as internal actors and messages. New transition rules show how actor platform-level operations work: name management for local actors, location update for remote actors, and message passing using the location identifications of actor platforms. Therefore, our operational semantics can be used to define specifically how actor platforms support actors when actor primitives are called.

Agha et al. [5] proposed the operational semantics for fundamental actor systems, and this

semantics was extended to mobile actors in [4]. The extended semantics treats actor migration using one transition rule like a synchronous operation. In this chapter, we define actor migration using multiple transition rules, because communication between actor platforms is based on asynchronous message passing. With our operational semantics, we could detect two message looping problems of the base operational semantics caused by message order shuffling and moving actors, and correct them. However, so far this kind of problem checking using the operational semantics is performed manually. It would be very useful to develop tools to perform this task automatically.

Chapter 7

Conclusion

As the scale of a single multi-agent application increases, its runtime becomes a more important concern in the development and execution of multi-agent systems. Reducing the runtime of a large-scale application will improve the quality of the application by increasing the chances to detect problems with a greater number of executions. Also, we can apply more time-consuming but more sophisticated routines and increase the size of the application with the saved time. In large-scale multi-agent systems, the runtimes are strongly related to communication among the agents. Therefore, in this thesis we have proposed and evaluated agent framework-level services to reduce agent communication costs from three points of view: dynamic agent distribution, application agent-oriented middle agent services, and message passing for mobile agents.

These three services dynamically adapt to the characteristics of a given multi-agent application. By monitoring the communication patterns of agents and the workload of agent platforms, each agent platform can assess its current workload and attempt to distribute local agents to other agent platforms. ATSpace focuses on the individual interests of application agents by using search algorithms delivered from them instead of the default search algorithm of ATSpace. Message passing mechanisms for mobile agents adapt to the location changes of mobile agents by using the extended naming scheme of agents or location caches. Thus, agent platforms may deliver messages directly to the current agent platforms of receiver agents without routing messages through their birth agent platforms.

However, we acknowledge that the proposed services introduce additional overhead: monitoring and decision making for agent distribution, object migration for ATSpace services, and location cache management and additional platform-level messages for location-based message passing.

Moreover, these services work effectively with multi-agent applications that have certain attributes. For dynamic agent distribution, the number of agents should be large enough to require multiple computers, and the communication among agents should be one of the most important factors in the runtime of a given application. In addition, the communication patterns of agents should change continuously. In the ATSpace model, agents should have different interests in the same data in a tuple space, and their interests should change continuously. For message passing to mobile agents, the locations of application agents should change continuously. When a multi-agent application has these behaviors, agent frameworks can compensate for the additional overhead with the performance benefit of the proposed services.

In this thesis, we have also presented a modified definition of the operational semantics published in [5]. This operational semantics describes the fundamental behavior of agent platforms as well as that of agents. With this operational semantics, we could analyze the basic services of our agent framework and detect two message looping problems caused by asynchronous message passing and moving actors. This operational semantics will be a base to specify the behavior of the proposed agent framework services and verity it.

The proposed services have been synthesized in an actor framework called *AAA* (*Adaptive Actor Architecture*) [64, 66], an extension of *AA* (*Actor Architecture*) [63, 65]. This framework was developed in Java programming language and uses TCP/IP [99] for agent communication. These characteristics restrict agents on an AAA platform so that they communicate only with agents on other AAA platforms. However, the proposed services need not be limited to our implementation.

In the communication patterns of agents, dynamic agent distribution is related to middle agent services. If agents communicate with others mainly through middle agents, moving tuples among the middle agents may have similar effects as moving agents. Theodoropoulos and Logan [106] explained how interest management could affect load balancing. We would like to apply a similar idea to the ATSpace model. As future research, we plan to extend AAA to support FIPA (Foundation for Intelligent Physical Agents) ACL (Agent Communication Language) [45] and implement AAA platforms in different programming languages, such as C/C++ [74, 100] or Scheme [42] to evaluate the interoperability of AAA platforms with different characteristics and the proposed framework services on them.

Appendix A

Agent-based Simulations

A.1 Introduction

To evaluate the proposed mechanisms in this thesis, we often used UAV (Unmanned Aerial Vehicle) simulations as a target application. This is because agent-based simulations are one of the most popular large-scale multi-agent applications, although online games, avatars in the virtual world, and so on are other applications. This appendix gives a brief introduction to our agent-based simulations.

Figure A.1 depicts the system architecture for our agent-based simulations. This architecture comprises three layers: a distributed agent framework layer, a generic agent simulator layer, and a task specific agent simulator layer. Our *distributed agent framework*, called *AAA* (*Adaptive Actor Architecture*), provides an execution environment for large-scale multi-agent applications. The main services of our agent framework are agent state management, agent communication, agent migration, and middle agent services. Moreover, the agent framework services proposed in chapter 2 have been provided by this layer. A more detailed explanation of our agent framework can be found in [64].

The *Generic agent simulator* provides simulation-specific services. Although our agent framework provides effective and efficient multi-agent framework services for large-scale agent-based simulations, this framework does not support simulation-specific services. For example, although our target simulations are *time-driven discrete-event simulations*, our agent platform does not provide a virtual time synchronization service for task-specific agents. Therefore, two types of simulation-oriented agents are executed in the general agent simulator layer, and they support task-specific

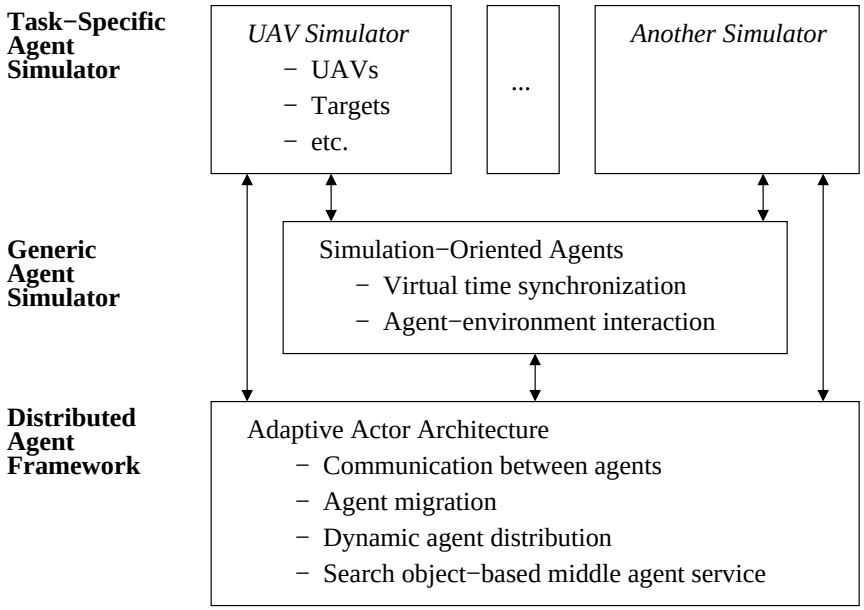


Figure A.1: Three-Layered Architecture for Agent-based Simulations

agents, such as UAV agents and target agents. Simulation-oriented agents are not part of the agent platform, but they can be used with many kinds of simulations.

On top of the distributed agent framework and generic agent simulator, many kinds of task-specific agent simulators can be developed and executed. In this section, we briefly describe our UAV simulation as a task-specific simulation. More detailed information about our UAV simulations can be found in [69].

The Generic agent simulator provides virtual time synchronization and agent–environment interaction services for task-specific agents.

A.1.1 Virtual Time Synchronization

In our simulations, each component manages its virtual time because each agent has its own control thread as an independent entity. However, this causes inconsistency in the virtual times of the agents. To maintain synchronization among these times, our agent-based simulations include a component to synchronize the virtual times of agents. This component is called the Virtual Time Synchronizer (VTS). Since our simulator is a distributed system, this component should be distributed on participating agent platforms. Therefore, each agent platform has a Local Virtual Time Synchronizer (LVTS) that manages the virtual times of local agents and a Global Virtual

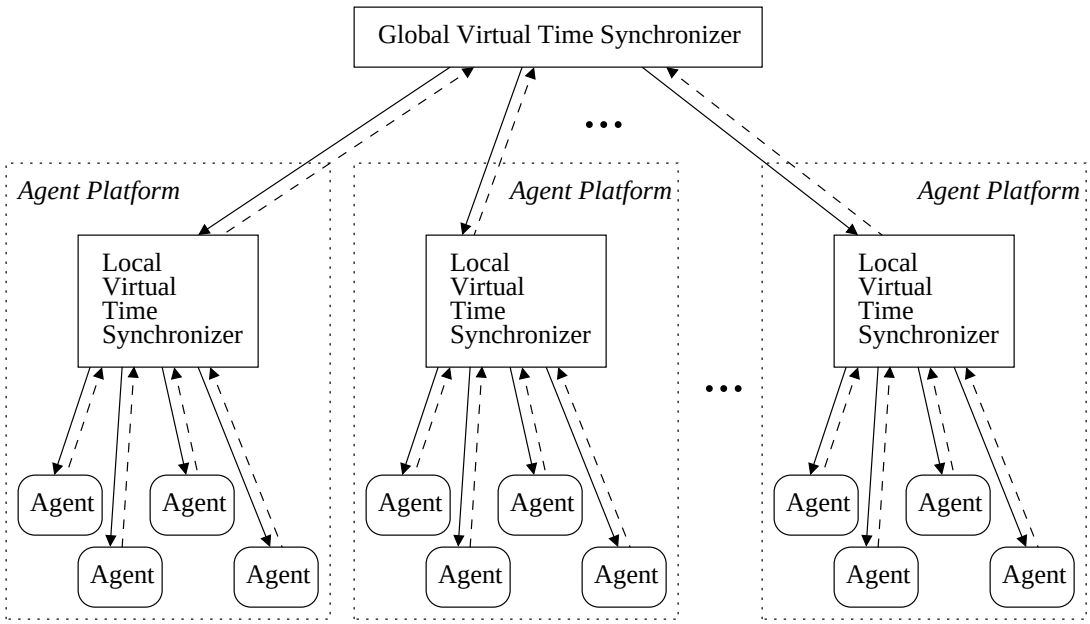


Figure A.2: Control Flow for Virtual Time Synchronization

Time Synchronizer (GVTS) that manages the virtual times of LVTSs (see Figure A.2). The GVTS and LVTSs are implemented as agents.

We assume that, at the beginning of a simulation, the initial value of the local virtual time of each agent is set as 0. After all agents start, the GVTS broadcasts a virtual time clock message to the LVTSs, and each LVTS then rebroadcasts the message to agents on the same agent platform (the solid line in Figure A.2). When an agent receives the message, this agent performs a subtask assigned for a simulation time step. For example, in UAV simulations, a UAV updates its location and direction vector, and also checks whether new objects, such as other UAVs or targets, are detected. If a new neighboring UAV is detected, the UAV might exchange some information with the new neighboring UAV. After an agent finishes this operation, it increases its local time and sends a reply message to its LVTS. When an LVTS receives the reply messages from all other local agents, the LVTS increases its virtual time and then sends a reply message to the GVTS (the dashed line in Figure A.2). When the GVTS receives reply messages from all the LVTSs, the GVTS increases its virtual time and broadcasts another virtual time clock message to all LVTSs. These steps are repeated until the current simulation ends.

This procedure for managing the synchronization among the virtual times of agents requires reliable communication among the agents. If a virtual time clock message is lost, a task-specific

simulator cannot progress the simulation any longer. Therefore, our distributed agent framework supports a reliable transport communication layer.

With this time management model, agents cannot proceed until all other agents finish their operations. Therefore, balancing workloads significantly affects the overall performance of our system. Dynamic agent distribution mechanisms help to balance the overloads of each computer node. However, since each agent platform in our agent framework does not have global workload information, our mechanisms do not completely balance the workloads of all the computer nodes.

Agent–environment interaction

Our task-specific simulations use the *agent–environment interaction* model [114]. Although this simulation-oriented service used for agent–environment interaction does not depend on our current task-specific simulations, in order to give a concrete explanation, here we describe the behavior of this service with our UAV simulations.

In UAV simulations all UAVs and targets are implemented as intelligent agents, whereas the navigation space is implemented as an environment. To avoid creating a centralized component, the environment is implemented as a set of environment agents. Each environment agent takes charge of a certain *navigation area*, and environment agents are assigned to different agent platforms. It is not necessary to let each agent platform have only one environment agent; an agent platform may have more than one environment agent or no environment agent.

Although UAV agents interact directly with each other by message passing, they can also communicate indirectly with neighboring UAVs and targets through the environment. For this purpose, each agent updates its location on the environment agent that manages its subarea. When a UAV or target agent is near a subarea boundary, the agent should report its location to more than one environment agent. During execution, an application agent should decide to which environment agent it should send a service request message.

The behavior of all the radar sensors of the UAVs and the local message broadcasting is implemented as a function of the environment agents. Environment agents are periodically activated to search neighboring UAVs and targets of each UAV and then report the result to the UAV. This message is regarded as a radar-sensing result. Also, when a UAV wants to broadcast to its neigh-

boring UAVs, the UAV sends a search object to an environment agent. This search object includes information about the location of the agent and its local communication range as well as a message to be delivered to other UAVs. With this information, the object computes the distances between itself and others to find neighboring UAVs, and the environment agent delivers the message to them.

For this procedure, environment agents are implemented on top of ATSpace. As an extended agent of the ATSpace agent, an environment agent has all the functions of an ATSpace agent, and it has additional functions to periodically activate operations on the tuple space in the ATSpace agent. More detailed information about ATSpace can be found in [65].

Since UAVs and targets can continuously move from one divided area to another during a simulation time, application agents and environment agents can be placed on different agent platforms. This placement increases the amount of inter-node message passing, but the problem is mitigated by our dynamic agent distribution mechanisms.

A.2 UAV Simulations

As a task-specific simulator, a UAV simulator was developed on top of the generic agent simulator and the agent distributed framework, and this simulator performed a variety of UAV simulations. The purpose of our simulations was to analyze the effects of various UAV coordination strategies under given mission scenarios. Specifically, the UAVs performed a surveillance mission. At the beginning of a simulation, UAVs were launched to a mission area without any prior information about the locations of points of interest. However, each UAV was equipped with some sensors that could detect objects within the sensing range. When a UAV detected an object, the UAV approached the object to survey it. We refer to such objects as *targets*.

A.2.1 UAV Coordination Strategies

When a UAV detected more than one target, the UAV selected the best target according to the distance between the UAV and the target and the value of the target. In our UAV simulations, each target had its own value. This value could be interpreted in several different ways. The value might represent the number of injured people or the importance of a building. Alternatively, the value

might represent the time required to investigate and/or serve a target by a UAV. For simplicity, in our simulations we used a single numeric value for each target, instead of using symbolic information or information about time constraints.

We assumed that a UAV handled one target at a time, but that a target might be served by several UAVs at the same time. When several UAVs served a target together, they consumed the value of the target quickly. However, controlling target assignments might improve the global mission performance of UAVs. In our UAV simulations, we investigated the effects of three UAV strategies: a self-interested UAV strategy, an information sharing-based cooperation strategy, and a team-based coordination strategy. These three strategies were based on distributed decision making; there was no centralized component, and each UAV made its own decision, although it might exchange some information with its neighboring UAVs.

Self-interested UAV strategy

In the self-interested UAV strategy, when a UAV sensed a target, this UAV approached it with the intention of consuming its entire value. When another UAV detected the same target, it also proceeded to consume the value of the target, irrespective of what the other UAVs did. Persistent polling of the target value until such time as it was consumed completely served as a means of interaction among the UAVs. It was not unusual to have more than one UAV concentrated on a target, resulting in quicker consumption of its value but also possibly in the duplication of services.

Information sharing-based cooperation strategy

In this strategy, once a UAV had discovered and located a target, it broadcasted this information so that other UAVs could direct their attention to the remaining targets. The reception of such information would result in the UAVs purging the targets that were advertised. This approach allowed for a larger set of targets to be visited in a given time interval and was thus expected to be faster in accomplishing the mission goal. Exchange of information between UAVs referring to the same target would result in a UAV with a lower identification number to determine the UAV that would be responsible for that target based on parameters such as the distance from the target.

Team-based coordination strategy

In the team-based coordination strategy, a UAV took on the mantle of the leader of its team and dictated a course of action to the other UAVs about the targets they needed to visit. A team was dynamically formed and changed according to the set of targets detected; that is, when a UAV detected more than one target, the UAV tried to handle the targets together with its neighboring UAVs. At that time, the main concern was how to select an optimum UAV and decide on the number of UAVs required to accomplish a task when the number of neighboring UAVs was sufficient. As the basic coordination protocol, we used the Contract Net Protocol [96, 97]. The UAV initiating the group mission worked as the group leader UAV, and the other participant UAVs were called member UAVs. When a member UAV detected another target, the UAV delivered information about the new target to the leader UAV, and the leader UAV would add the target to the set of targets to be handled. The leader UAV considered the distance between a target detected and neighboring UAVs to assign the target. When a member UAV consumed the entire value of a target, the UAV seceded from its group.

A.3 Experimental Results

For our experiments, we used a cluster of four computers (3.4 GHz Intel Pentium IV CPU and 2 GB main memory) connected by a Giga-bit switch. All programs except the System Monitor were developed in Java [53], and system call routines of the System Monitor were implemented in C. The system call routines and the System Monitor component were connected by JNI (Java Native Interface) [52]. In our agent framework, message delivery between different agent platforms was performed by TCP [87].

We conducted UAV simulations to evaluate different UAV coordination strategies. The size of the simulation region was set to 1,000,000 ft. × 800,000 ft. × 8,000 ft. (74,000 square km and an altitude of 2.4 km), the size of the mission region to 400,000 ft. × 600,000 ft. × 8,000 ft. (22,000 square km and an altitude of about 2.4 km), the radius of local broadcast communication of a UAV to 15,000 feet (4.6 km), and the radius of a radar sensor to 7,500 feet (2.3 km). The simulation area was the entire geographical area for UAV navigation, but targets were normally distributed

only in the mission area. Each target had an initial value of 100, and a UAV could consume 5 of this value per second when the UAV was within 1,000 feet of the target.

For simplicity, handling a target was simulated by consuming the value of the target. Also, the value of a target was delivered to every UAV within radar-sensing range. Therefore, when a UAV detected more than one target, the UAV decided on its best target according to its current values and the distance between the UAV and each target.

To investigate how different cooperation strategies influenced the performance of a joint mission, we used the Average Service Cost (ASC), which is defined as follows:

$$ASC = \frac{\sum_i^n (NT_i - MNT)}{n} \tag{A.1}$$

where n is the number of UAVs, NT_i is the navigation time of the i-th UAV, and MNT (Minimum Navigation Time) is the average navigation time of all the UAVs when there were no targets. When there were no targets in the given mission area, the UAVs simply navigated according to their initial routes. Since they did not handle targets, the navigation time of each UAV had to be minimal, regardless of its current coordination strategy. Therefore, to evaluate the additional time for target handling with the chosen coordination strategy, we subtracted this value from the navigation time of each UAV that processed targets. In a real situation, ASC would mean the average of the additional navigation time of UAVs needed to handle given targets, and is related to the average fuel usage of the UAVs.

Figure A.3 shows the ASC for three different cooperation strategies. When the number of UAVs increased, the ASC decreased in every case. However, the information sharing-based cooperation and the team-based coordination strategies were better than the self-interested UAV strategy. From this result, we concluded that collaboration among the UAVs was useful for handling targets, even though the UAVs in the self-interested UAV strategy quickly consumed the value of a target when they handled the target together. Another interesting observation was that although the team-based approach performed more complex coordination protocols than the information sharing-based approach, their runtimes were quite similar. Therefore, we also conclude that a simple coordination algorithm might also be very effective.

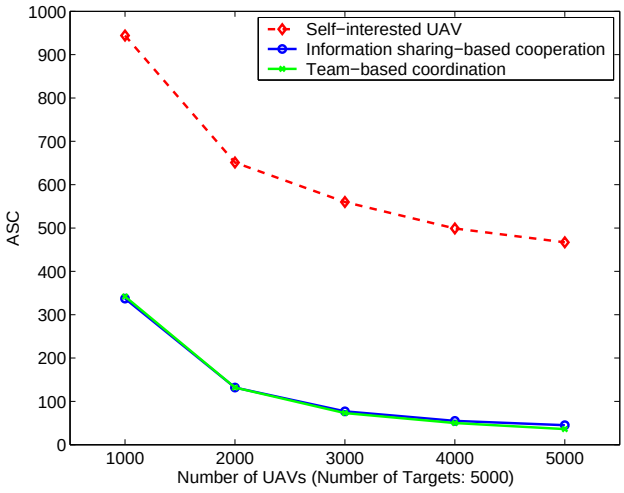


Figure A.3: Average Service Cost (ASC) for Three Different Coordination Strategies

Appendix B

Notation and Transition Rules

B.1 Notation

- e: an internal expression (or operation) of an actor except actor primitives such as `create`, `become`, and `send`.
- R: the reduction context in which the expression currently being evaluated occurs.
- $(b)_a$: actor a with behavior b ready to accept a message.
- $[e]_a$: actor a executing e, where e is the current processing state.
- $\mapsto$: the reduction relation for configurations.
- $\overset{\lambda}{\mapsto}$: the reduction relation for functional redexes.
- $\overset{L}{\longrightarrow}$: a multi-step transition.
- $\texttt{<}a \Leftarrow v\texttt{>}$: a message on an actor platform; its receiver actor is a, and its content is v.
- $[l : m]$: a message on the Internet; its destination location (or actor platform) is l, and its content is m.
- $\mathrm{FV}(\alpha(a))$: free variables of actor a.
- $\texttt{app}(b, v)$: a method invocation; the method in behavior b related to a message v is executed.
- $\texttt{id}(a)$: the local identification of actor a.

- `loc`(a): the location (or identification) of the current platform of actor a.
- `origin_loc`(a): the location (or identification) of the birth actor platform where actor a was created.
- `previous_loc`(a): the identification of the previous actor platform when actor a arrives from another actor platform.

B.2 Transition Rules

B.2.1 Transition Rules for Actor Mobility

The transition rules for UAA (Universal Actor Address)-based message passing are defined as follows. In the following transition rules, we assume that $l \neq l'$.

<fun>

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow \left\langle\!\!\left\langle \alpha, [e]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [e']_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

<create>

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{create}(b')]]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [R[a']]_a, (b')_{a'} \mid \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

where a' is fresh and $\mathcal{N}' = \mathcal{N} \cup \{a'\}$

<become>

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{become}(b')]]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [R[\texttt{nil}]]_{a'}, (b')_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

where a' is fresh

<send>

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{send}(a', v)]]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [R[\texttt{nil}]]_a \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

where $a' \in \mathrm{FV}(\alpha(a))$ and $m = \texttt{<}a' \Leftarrow v\texttt{>}$

<migrate>

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{migrate}(l)]]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [R[\texttt{nil}]]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{migrate}(l')]]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [R[\texttt{nil}]]_{a'} \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}'}$$

if $l = \texttt{origin\_loc}(a)$

where a' is fresh, $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$, $\mathcal{N}' = \mathcal{N} - \{a\}$, and $\mathcal{L}' = \mathcal{L} \cup \{[a, l']\}$

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{migrate}(l')]]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [R[\texttt{nil}]]_{a'} \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

if $l \neq \texttt{origin\_loc}(a)$

where a' is fresh, $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$, and $\mathcal{N}' = \mathcal{N} - \{a\}$

<receive>

$$\left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, [\texttt{app}(b, v)]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $a \in \mathcal{N}$

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_a]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [a,l']} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

if $l = \texttt{origin\_loc}(a)$

where $\mathcal{N}' = \mathcal{N} \cup \{a\}$

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_a]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; m, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

if $l \neq \texttt{origin\_loc}(a)$ and $l' = \texttt{origin\_loc}(a)$

where $\mathcal{N}' = \mathcal{N} \cup \{a\}$ and $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [a, l']\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [a,l'']} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [a,l']}$$

`<out>`

$$\left\langle\!\!\left\langle \alpha \;\middle|\; m, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $m = \texttt{<}a \Leftarrow v\texttt{>}$ and $a \notin \mathcal{N}$

where $\texttt{loc}(a) = l'$

or

if $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$ or $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$

`<in>`

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

where ($m = \texttt{<}a \Leftarrow v\texttt{>}$ and $\texttt{loc}(a) = l$), $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$, or $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$

`<ready>`

$$\left\langle\!\!\left\langle \alpha, [\texttt{nil}]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $a \in \mathcal{N}$

`<die>`

$$\left\langle\!\!\left\langle \alpha, [\texttt{nil}]_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $a \notin \mathcal{N}$

B.2.2 Transition Rules for Location Address-based Message Passing

The transition rules for location address-based message passing are defined as follows:

<fun>

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow \left\langle\!\!\left\langle \alpha, \texttt{[}e\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}e'\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

<create>

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{create}(b')]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[a']\texttt{]}_a, (b')_{a'} \mid \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

where a' is fresh and $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a')\}$

<become>

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{become}(b')]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_{a'}, (b')_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

where a' is fresh

<send>

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{send}(a', v)]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_a \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

where $a' \in \mathrm{FV}(\alpha(a))$ and $m = \texttt{<}a' \Leftarrow v\texttt{>}$

<migrate>

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{migrate}(l)]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{migrate}(l')]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_{a'} \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}'}$$

if $l = \texttt{origin\_loc}(a)$

where a' is fresh, $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$, $\mathcal{N}' = \mathcal{N} - \{\texttt{uaa}(a)\}$, and $\mathcal{L}' = \mathcal{L} \cup \{[\texttt{uaa}(a), l']\}$

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{migrate}(l')]\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_{a'} \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

if $l \neq \texttt{origin\_loc}(a)$

where a' is fresh, $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$, and $\mathcal{N}' = \mathcal{N} - \{\texttt{uaa}(a)\}$

<receive>

$$\left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \texttt{<}a' \Leftarrow v\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[app}(b, v)\texttt{]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $\texttt{uaa}(a) \in \mathcal{N}$ and $\texttt{uaa}(a) = \texttt{uaa}(a')$

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_{a'}]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [\texttt{uaa}(a), l']} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

if $l = \texttt{origin\_loc}(a)$ and $\texttt{uaa}(a) = \texttt{uaa}(a')$

where $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a)\}$

($a = [l + l + i]$ and $a' = [l' + l + i]$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_{a'}]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

if $l \neq \texttt{origin\_loc}(a)$, $l' = \texttt{origin\_loc}(a)$, and $l' = \texttt{previous\_loc}(a)$

where $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a)\}$, and $\texttt{uaa}(a) = \texttt{uaa}(a')$

($a = [l + l' + i]$, $a' = [l'' + l' + i]$, and $l' = l''$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_{a'}]\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; m, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}}$$

if $l \neq \texttt{origin\_loc}(a)$, $l' = \texttt{origin\_loc}(a)$, and $l' \neq \texttt{previous\_loc}(a)$

where $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a)\}$, $\texttt{uaa}(a) = \texttt{uaa}(a')$, $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$,

($a = [l + l' + i]$, $a' = [l'' + l' + i]$, and $l' \neq l''$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [a', l']\texttt{>}, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [\texttt{uaa}(a), l'']} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [\texttt{uaa}(a'), l']}$$

if $\texttt{uaa}(a) = \texttt{uaa}(a')$

($a = [l'' + l + i]$ and $a' = [l' + l + i]$)

<out>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; m, \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $m = \texttt{<}a \Leftarrow v\texttt{>}$ and $\texttt{uaa}(a) \notin \mathcal{N}$

where $\texttt{loc}(a) = l'$

or

if $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$ or $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$

<in>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, m \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

where ($m = \texttt{<}a \Leftarrow v\texttt{>}$ and $\texttt{loc}(a) = l$), $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$, or $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$

<ready>

$$\left\langle\!\!\left\langle \alpha, \texttt{[nil]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $\texttt{uaa}(a) \in \mathcal{N}$

<die>

$$\left\langle\!\!\left\langle \alpha, \texttt{[nil]}_a \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L}}$$

if $\texttt{uaa}(a) \notin \mathcal{N}$

B.2.3 Transition Rules for Delayed Message Passing

The transition rules for delayed message passing are defined as follows:

<fun>

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow \left\langle\!\!\left\langle \alpha, \texttt{[}e\texttt{]}_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}e'\texttt{]}^{b}_{a} \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

<create>

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{create}(b')]\texttt{]}_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

$$\mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[a']\texttt{]}_a, (b')_{a'} \mid \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L} \mid \mathcal{B}}$$

where a' is fresh and $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a')\}$

<become>

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{become}(b')]\texttt{]}_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_{a'}, (b')_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

where a' is fresh

<send>

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{send}(a', v)]\texttt{]}_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_a \;\middle|\; \mu, m \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

where $a' \in \mathrm{FV}(\alpha(a))$ and $m = \texttt{<}a' \Leftarrow v\texttt{>}$

<migrate>

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{migrate}(l)]\texttt{]}_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

$$\left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{migrate}(l')]\texttt{]}_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_{a'} \;\middle|\; \mu, m \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L}' \mid \mathcal{B}'}$$

where a' is fresh, $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$,

$\mathcal{N}' = \mathcal{N} - \{\texttt{uaa}(a)\}$, $\mathcal{L}' = \mathcal{L} \cup \{[\texttt{uaa}(a), l']\}$, and $\mathcal{B}' = \mathcal{B} \cup \{\texttt{uaa}(a)\}$

<delay>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}a \Leftarrow v\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta, \texttt{<}a \Leftarrow v\texttt{>} \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

if $\texttt{uaa}(a) \notin \mathcal{N}$ and $\texttt{uaa}(a) \in \mathcal{B}$

<receive>

$$\left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \texttt{<}a' \Leftarrow v\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, \texttt{[}\texttt{app}(b, v)\texttt{]}^{b}_{a} \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

if $\texttt{uaa}(a) \in \mathcal{N}$ and $\texttt{uaa}(a) = \texttt{uaa}(a')$

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_{a'}]\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [\texttt{uaa}(a),\ l'] \mid B} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu, m \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L} \mid \mathcal{B}}$$

if $l = \texttt{origin\_loc}(a)$ and $l'' = \texttt{previous\_loc}(a)$, and $\texttt{uaa}(a) = \texttt{uaa}(a')$

where $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a)\}$, and $m = \texttt{<}l'' \Leftarrow [a,\ l]\texttt{>}$

($a = [l + l + i]$ and $a' = [l'' + l + i]$)

($l' = l''$ or $l' \neq l''$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_{a'}]\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu, m \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L} \mid \mathcal{B}}$$

if $l \neq \texttt{origin\_loc}(a')$, $l' = \texttt{origin\_loc}(a')$, and $l' = \texttt{previous\_loc}(a)$

where $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a)\}$, $\texttt{uaa}(a) = \texttt{uaa}(a')$, $m = \texttt{<}l' \Leftarrow [a,\ l]\texttt{>}$

($a = [l + l' + i]$, $a' = [l'' + l' + i]$, and $l' = l''$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [(b)_{a'}]\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu, m_1, m_2 \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N}' \mid \mathcal{L} \mid \mathcal{B}}$$

if $l \neq \texttt{origin\_loc}(a')$, $l' = \texttt{origin\_loc}(a')$, and $l' \neq \texttt{previous\_loc}(a)$

where $\mathcal{N}' = \mathcal{N} \cup \{\texttt{uaa}(a)\}$, $\texttt{uaa}(a) = \texttt{uaa}(a')$,

$m_1 = \texttt{<}l' \Leftarrow [a,\ l]\texttt{>}$, $m_2 = \texttt{<}l'' \Leftarrow [a,\ l]\texttt{>}$,

($a = [l + l' + i]$, $a' = [l'' + l' + i]$, and $l' \neq l''$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [a', l']\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [\texttt{uaa}(a), l''] \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L},\ [\texttt{uaa}(a'), l'] \mid \mathcal{B}'}$$

if $l = \texttt{origin\_loc}(a')$, $\texttt{uaa}(a) = \texttt{uaa}(a')$, and $\texttt{uaa}(a') \in \mathcal{B}$

where $\mathcal{B}' = \mathcal{B} - \{\texttt{uaa}(a')\}$

($a = [l'' + l + i]$, $a' = [l' + l + i]$)

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \texttt{<}l \Leftarrow [a', l']\texttt{>}, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}'}$$

if $l \neq \texttt{origin\_loc}(a')$, $\texttt{uaa}(a) = \texttt{uaa}(a')$, and $\texttt{uaa}(a') \in \mathcal{B}$

where $\mathcal{B}' = \mathcal{B} - \{\texttt{uaa}(a')\}$

<out>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; m, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

if $m = \texttt{<}a \Leftarrow v\texttt{>}$, $\texttt{uaa}(a) \notin \mathcal{N}$, and $\texttt{uaa}(a) \notin \mathcal{B}$

where $\texttt{loc}(a) = l'$

or

if $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$ or $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$

<delay-out>

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; m, \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; m, \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

if $m = \texttt{<}a \Leftarrow v\texttt{>}$ and $\texttt{uaa}(a) \notin \mathcal{B}$

`<in>`

$$\left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu, m \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

where ($m = \texttt{<}a \Leftarrow v\texttt{>}$ and $\texttt{loc}(a) = l$), $m = \texttt{<}l' \Leftarrow [(b)_a]\texttt{>}$, or $m = \texttt{<}l' \Leftarrow [a, l]\texttt{>}$

`<ready>`

$$\left\langle\!\!\left\langle \alpha, \texttt{[nil]}_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha, (b)_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

if $\texttt{uaa}(a) \in \mathcal{N}$

`<die>`

$$\left\langle\!\!\left\langle \alpha, \texttt{[nil]}_a \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}} \mapsto \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \;\middle|\; \beta \right\rangle\!\!\right\rangle^{l}_{\mathcal{N} \mid \mathcal{L} \mid \mathcal{B}}$$

if $\texttt{uaa}(a) \notin \mathcal{N}$

References

[1] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: A Self-organizing Structured P2P System. *SIGMOD Record*, 32(3):29–33, 2003.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

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

[4] G. A. Agha and N. Jamali. Concurrent Programming for Distributed Artificial Intelligence. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 505–534. MIT Press, 1999.

[5] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[6] S. Alouf, F. Huet, and P. Nain. Forwarders vs. Centralized Server: An Evaluation of Two Approaches for Locating Mobile Agents. *Performance Evaluation*, 49(1-4):299–319, September 2002.

[7] S. Androutsellis-Theotokis and D. Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004.

[8] J.-P. Arcangeli, L. Bray, A. Marcoux, C. Maurel, and F. Migeon. Reflective Actors for Mobile Agents Programming. In *Proceedings of ECOOP'2000 Workshop on Reflection and Meta-level Architecture*, pages 1–7, Cannes, France, 2000.

[9] J.-P. Arcangeli, C. Maurel, and F. Migeon. An API for High-level Software Engineering of Distributed and Mobile Applications. In *The Eighth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'01)*, pages 155–161, October 2001.

[10] R. K. Arora and S. P. Rana. Heuristic Algorithms for Process Assignment in Distributed Computing Systems. *Information Processing Letters*, 11(4/5):199–203, 1980.

[11] M. Astley. The Actor Foundry–Manual, February 1999. http://osl.cs.uiuc.edu/foundry/.

[12] 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.

[13] C. Banino, O. Beaumont, A. Legrand, and Y. Robert. Scheduling Strategies for Master-Slave Tasking on Heterogenous Proceesor Grids. In *Applied Parallel Computing: Advanced Scientific Computing: 6th International Conference (PARA '02), LNCS 2367*, pages 423–432. Springer-Verlag, June 2002.

[14] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali. A Load Balancing Framework for Adaptive and Asynchronous Applications. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):183–192, February 2004.

[15] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 353–366, 2004.

[16] S. H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, 30(3):550–557, 1981.

[17] S. H. Bokhari. *Assignment Problems in Parallel and Distributed Computing*. Kluwer Academic Publishers, 1987.

[18] A. H. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, San Mateo, California, 1988.

[19] F. Bousquet, I. Bakam, H. Proton, and C. Le Page. Cormas: Common-Pool Resources and Multi-Agent Systems. In *Proceedings of the 11th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE '98)*, pages 826–838. Springer-Verlag, 1998. Lecture Notes in Artificial Intelligence 1416.

[20] M. Bouzid, V. Chevrier, S. Vialle, and F. Charpillet. Parallel Simulation of a Stochastic Agent/Environment Interaction. *Integrated Computer-Aided Engineering*, 8(3):189–203, 2001.

[21] N. S. Bowen, C. N. Nikolaou, and A. Ghafoor. On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems. *IEEE Transactions on Computers*, 41(3):257–273, March 1992.

[22] L. Bray, J.-P. Arcangeli, and P. Sallé. Programming Concurrent Objects Scheduling Strategies using Reflection. In *Proceedings of ACM Workshop on Scheduling Algorithms for Parallel and Distributed Computing–From They to Practice*, pages 7–11, Rhodes, Greece, 1999.

[23] R. K. Brunner and L. V. Kalé. Adaptive to Load on Workstation Clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112, February 1999.

[24] G. Bruns, P. Mossinger, D. Polani, R. Schmitt, R. Spalt, T. Uthmann, and S. Weber. XRaptor: Simulation Environment for Multi-Agent Systems. http://www.informatik.uni-mainz.de/ polani/XRaptor/XRaptor.html.

[25] L. L. Burge III and K. M. George. JMAS: A Java-based Mobile Actor System for Distributed Parallel Computation. In *The 5th USENIX Conference on Object-Oriented Technologies and Systems*, pages 115–130, San Diego, California, May 1999.

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

[27] W. Cai, S. J. Turner, and H. Zhao. A Load Management System for Running HLA-based Distributed Simulations over the Grid. In *Proceedings of the Sixth IEEE International Workshop on Distributed Simulation and Real-Time Applications*, pages 7–14, October 2002.

[28] C. J. Callsen and G. Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel Distributed Computing*, 21(3):289–300, 1994.

[29] N. Camiel, S. London, N. Nisan, and O. Regen. The Popcorn Project: Distributed Computation over the Internet in Java. In *Proceedings of the 6th International World Wide Web Conference, W3*, Santa-Clara, California, April 1997.

[30] J. Cao, D. Spooner, S. A. Jarvis, and G. R. Nudd. Grid Load Balancing Using Intelligent Agents. *Future Generation Comptuer Systems*, 21(1):135–149, 2005.

[31] J. Cao, Y. Sun, X. Wang, and S. Das. Scalable Load Balancing on Distributed Web Servers Using Mobile Agents. *Journal of Parallel and Distributed Computing*, 63(10):996–1005, 2003.

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

[33] K. Chow and Y. Kwok. On Load Balancing for Distributed Multiagent Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(8):787–801, August 2002.

[34] P. Ciancarini and D. Rossi. Coordinating Java Agents over the WWW. *World Wide Web*, 1(2):87–99, 1998.

[35] K. Decker, K. Sycara, and M. Williamson. Middle-Agents for the Internet. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 578–583, Nagoya, Japan, 1997.

[36] 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.

[37] Defense Modeling and Simulation Office. High Level Architecture. https://www.dmso.mil/public/public/transition/hla.

[38] T. Desell, K. El Maghraoui, and C. Varela. Load Balancing of Autonomous Actors over Dynamic Networks. In *Proceedings of the Hawaii International Conference on System Sciences HICSS-37 Software Technology Track*, Hawaii, January 2004.

[39] S. Desic and D. Huljenic. Agents Based Load Balancing with Component Distribution Capability. In *Proceedinngs of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID2002)*, pages 327–331, May 2002.

[40] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan. Design of Dynamic Load-Balancing Tools for Parallel Applications. In *Proceedings of the International Conference on Supercomputing*, pages 110–118, Santa Fe, New Mexicos, 2000.

[41] 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.

[42] R. K. Dybvig. *The Scheme Programming Language*. The MIT Press, 3 edition, 2003.

[43] K. Efe. Heuristic Models for Task Assignment Scheduling in Distributed Systems. *IEEE Computer*, 15:50–56, 1982.

[44] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

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

[46] H. Garcia-Molina, J. D. Ullman, and J. D. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[47] L. Gasser and K. Kakugawa. MACE3J: Fast Flexible Distributed Simulation of Large, Large-Grain Multi-Agent Systems. In *Proceedings of the First International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, pages 745–752, Bologna, Italy, July 2002.

[48] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Language and Systems*, 7(1):80–112, January 1985.

[49] S. Genaud, A. Giersch, and F. Vivien. Load-Balancing Scatter Operations for Grid Computing. *Parallel Computing*, 30(8):923–946, 2004.

[50] R. A. Ghanea-Hercock, J. C. Collis, and D. T. Ndumu. Co-Operating Mobile Agents for Distributed Parallel Processing. In *Proceedings of the Third Annual Conference on Autonomous Agents (AGENT '99)*, pages 398–399, Seattle, Washington, May 1999.

[51] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Dynamic Structured P2P Systems. In *INFOCOMM 2004–Volume 4*, pages 2253– 2262, March 2004.

[52] R. Gordon. *Essential JNI: Java Native Interface.* Prentice-Hall, 1998.

[53] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language^{TM} Specification.* Addison-Wesley Professional, 3rd edition, 2005.

[54] J. R. Groff and P. N. Weinberg. *SQL: The Complete Reference.* McGraw-Hill, 2nd edition, 2002.

[55] W. Grosso. *Java RMI.* O'Reilly & Associates, 2001.

[56] O. Gutknecht and J. Ferber. The MADKIT Agent Platform Architecture. In *Proceedings of International Workshop on Infrastructure for Multi-Agent Systems*, pages 48–55. Springer-Verlag, 2001.

[57] C. Hewitt. *Description and Theoretical Anaysis (Using Schemata) of PLANNER: a Language for Proving Theorems and Manipulating Models in a Robot.* PhD thesis, Massachusetts Institute of Technology, 1971.

[58] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245, 1973.

[59] S. H. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan. Analysis of a Graph Coloring Based Distributed Load Balancing Algorithm. *Journal of Parallel and Distributed Computing*, 10(2):160–166, 1990.

[60] 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.

[61] S. Jagannathan. Customization of First-Class Tuple-Spaces in a Higher-Order Language. In *Proceedings of the Conference on Parallel Architectures and Languages–Volume 2, LNCS 506*, pages 254–276. Springer-Verlag, 1991.

[62] N. Jamali and G. Agha. CyberOrgs: A Model for Decentralized Resource Control in Multi-Agent Systems. In *Proceedings of Workshop on Representations and Approaches for Time-Critical Decentralized Resource/Role/Task Allocation at the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, Melbourne, Australia, July 2003.

[63] M. Jang. The Actor Architecture–Manual, March 2004. http://osl.cs.uiuc.edu/aa/.

[64] M. Jang and G. Agha. Dynamic Agent Allocation for Large-Scale Multi-Agent Applications. In *Proceedings of International Workshop on Massively Multi-Agent Systems*, pages 19–33, Kyoto, Japan, December 2004.

[65] M. Jang and G. Agha. On Efficient Communication and Service Agent Discovery in Multi-agent Systems. In *Proceedings of the Third International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS '04)*, pages 27–33, Edinburgh, Scotland, May 2004.

[66] M. Jang and G. Agha. Adaptive Agent Allocation for Massivley Multi-agent Applications. In T. Ishida, L. Gasser, and H. Nakashinma, editors, *Massively Multi-Agent Systems I, LNCS(LNAI) 3446*, pages 25–39, Berlin, 2005. Springer-Verlag.

[67] M. Jang and G. Agha. Scalable Agent Distribution Mechanisms for Large-Scale UAV Simulations. In *Proceedings of the IEEE International Conference of Integration of Knowledge Intensive Multi-Agent Systems (KIMAS '05)*, pages 85–90, Waltham, Massachusetts, April 2005.

[68] M. Jang, A. Abdel Momen, and G. Agha. ATSpace: A Middle Agent to Support Application-Oriented Matchmaking and Brokering Services. In *Proceedings of IEEE/WIC/ACM IAT (Intelligent Agent Technology)–2004*, pages 393–396, Beijing, China, September 2004.

[69] M. Jang, S. Reddy, P. Tosic, L. Chen, and G. Agha. An Actor-based Simulation for Studying UAV Coordination. In *Proceedings of the 15th European Simulation Symposium (ESS 2003)*, pages 593–601, Delft, The Netherlands, October 2003.

[70] J. Joseph and C. Fellenstein. *Grid Computing*. IBM Press, 2004.

[71] D. R. Karger and M. Ruhl. New Algorithms for Load Balancing in Peer-to-Peer Systems. In *IRIS Student Workshop (ISW)*, Cambridge, Massachusetts, August 2003.

[72] D. R. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *Proceedings ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 36–43, Barcelona, Spain, June 2004.

[73] A. Keren and A. Barak. Adaptive Placement of Parallel Java Agents in a Scalable Computing Cluster. *Concurrency: Practice and Experience*, 10(11-13):971–976, December 1998.

[74] B. W. Kernighan, D. Ritchie, and D.M. Ritchie. *The C Programming Language*. Prentice Hall PTR, 2 edition, 1988.

[75] F. Kuhl, R. Weatherly, and J. Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall PTR, 1999.

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

[77] V. M. Lo. Heuristic Algorithms for Task Assignment in Distributed Systems. *IEEE Transactions on Computers*, 37(11):1384–1397, 1988.

[78] B. Logan and G. Theodoropoulos. The Distributed Simulation of Agent-Based Simulation. *Proceedings of the IEEE*, 89(2):174–185, February 2001.

[79] 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.

[80] D. S. Milojčić, W. LaForge, and D. Chauhan. Mobile Objects and Agents (MOA). In *Proceedings of USENIX COOTS'98*, 1998.

[81] 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.

[82] W. Obelöer, C. Grewe, and H. Pals. Load Management with Mobile Agents. In *Proceedings of the 24th Conference on EUROMICRO–Volume 2*, pages 1005–1012, August 1998.

[83] 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.

[84] J. Pang and J. Weisz. Object Placement in Distributed Multiplayer Games, 2003. http://www-2.cs.cmu.edu/ jweisz/papers/.

[85] R. U. Payli, E. Yilmaz, A. Ecer, H. U. Akay, and S. Chien. DLB–A Dynamic Load Balancing Tool for Grid Computing. In *Proceedings of Parallel CFD'04 Conference*, Canary Island, Spain, May 2004.

[86] C. E. Perkins. Mobile IP. *IEEE Communications Magazine*, 35(5):84–99, May 1997.

[87] L. L. Peterson and B. S. Davie. *Computer Networks: A System Approach*. Morgan Kaufmann, 3rd edition, 2003.

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

[89] M. Philippsen and M. Zenger. JavaParty–Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[90] R. Plew and R. Stephens. *Sams Teach Yourself SQL in 21 Days*. Sams, 4th edition, October 2002.

[91] 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.

[92] R. Ramakrishnan, J. Gehrke, R. Ramakrishnan, and J. Gehrke. *Database Management Systems*. McGraw Hill, 3rd edition, 2002.

[93] A. Rowstron. Mobile Co-ordination: Providing Fault Tolerance in Tuple Space Based Coordination Languages. In *Proceedings of the Third International Conference on Coordination Languages and Models*, pages 196–210, 1999.

[94] 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.

[95] P. K. Sinha. Chapter 7. Resource Management. In *Distributed Operating Systems*, pages 347–380. IEEE Press, 1997.

[96] R. G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, 29(12):1104–1113, 1980.

[97] R. G. Smith and R. Davis. Frameworks for Cooperation in Distributed Problem Solving. *IEEE Transactions on Systems, Man and Cybernetics*, 11(1):61–70, 1980.

[98] A. D. Stefano and C. Santoro. Locating Mobile Agents in a Wide Distributed Environment. *IEEE Transactions on Parallel and Distributed Systems*, 13:844–864, August 2002.

[99] W. R. Stevens. *UNIX Network Programming–Networking APIs: Sockets and XTI–Volume 1*. Prentice Hall, 2 edition, 1998.

[100] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 3 edition, 2000.

[101] D. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

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

[103] G. Tan, A. Persson, and R. Ayani. HLA Federate Migration. In *Proceedings of 38th Annual Simulation Symposium*, pages 243–250, 2005.

[104] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of 'Legacy' Java Software. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 236–255, Budapest, June 2001.

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

[106] G. Theodoropoulos and B. Logan. An Approach for Interest Management and Dynamic Load Balancing in Distributed Simulations. In *2001 European Simulation Interoperability Workshop (2001 Euro-SIW)*, pages 25–27, United Kingdom, June 2001.

[107] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, Malaga, June 2002. http://j-orchestra.org/.

[108] 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. Springer-Verlag, 1997.

[109] A. Tveit, Ø. Rein, J. V. Iversen, and M. Matskin. Scalable Agent-Based Simulation of Players in Massively Multiplayer Online Games. In *Proceedings of the 8th Scandinavian Conference on Artificial Intelligence*, Bergen, Norway, November 2003.

[110] C. A. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, University of Illinois at Urbana-Champaign, April 2001. http://www-osl.cs.uiuc.edu/docs/phd-varela01/varela-phd.pdf.

[111] 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.

[112] S. Vinoski. CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. *IEEE Communications*, 14(2):46–55, February 1997.

[113] C. Walshaw, M. Cross, and M. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *Journal of Parallel and Distributed Computing*, 47:102–108, 1997.

[114] D. Weyns, H. Parunak, F. Michel, T. Holvoet, and J. Ferber. Environments for Multiagent Systems, State-of-the-Art and Research Challenges. In *Proceedings of the First International Workshop on Environments for Multiagent Systems*, New York, 2004.

[115] B. Wims and C.-Z. Xu. Traveler: A Mobile Agent Infrastructure for Wide Area Parallel Computing. In *Proceedings of the IEEE Joint Sumposium ASA/MA'99: First International Symposium on Agent Systems and Applications (ASA'99) and Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, California, October 1999.

[116] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2002.

[117] C. Xu and F. C. M. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, 1997.

Author's Biography

Myeong-Wuk Jang was born in Seoul, Korea, on March 28, 1967. He received his BS degree in Computer Science from Korea University in 1990 and his MS degree in Computer Science from the Korea Advanced Institute of Science and Technology (KAIST) in 1992. He then worked on a research staff of the Electronics and Telecommunications Research Institute (ETRI) until 1998. During this time, he researched and developed multi-agent frameworks and systems, and he conducted interactional projects with SRI International. He also worked as a team manager at WookSung Electronics in 1998 and developed a graphical user interface for video phones. In 1999, he entered the University of Illinois at Urbana Champaign (UIUC) for graduate studies in Computer Science under the supervision of Professor Gul Agha. During his studies, he worked as a research assistant at the National Center for Supercomputing Applications (NCSA) from February 2000 to May 2001 and at the Open Systems Laboratory (OSL) from August 2001 to December 2005. Myeong-Wuk Jang's primary research interest is in the actor model, multi-agent framework services, and distributed systems.