Reactive Tuple Space for a Mobile Agent Platform

Yu Li

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science
Concordia University
Montreal, Quebec, Canada

July 2004

Your file Votre référence
ISBN: 0-612-94747-5
Our file Notre référence
ISBN: 0-612-94747-5

Abstract

Reactive Tuple Space for a Mobile Agent Platform

Yu Li

An agent programming model can simply specify agent behaviors as proactive behaviors and reactive behaviors. Typically a multi-agents system is based on message passing, which usually is space and time coupled. This thesis aims at developing a multiple reactive tuple space coordination media that provides space and time decoupling for agent dynamic interaction. It provides three categories of primitives: (i) synchronous primitives, (ii) asynchronous primitives, and (iii) reactive primitives. The synchronous and asynchronous primitives are intended to support agent proactive behaviors. The reactive primitives are for supporting agent reactive behaviors. The formal specification of the reactive tuple space is also described based on the state machine model. In order to enhance system efficiency, a replication protocol is designed to accommodate both locality and concurrency of tuple accesses. The reactive tuple space has been fully implemented and integrated with the JADE mobile agent platform. Comparing with ACL message passing in JADE, experiments are conducted to show that the reactive tuple space can provide competitive performance in terms of interaction latency and bandwidth for tuple spaces of reasonable size.

Acknowledgments

I would like to express my gratitude and respect to my supervisor Dr. Hon F. Li for his invaluable guidance, encouragement and support during the whole period of my research work. Dr. Li not only teaches me knowledge at class, during meetings and through emails, he also tries to enlighten me on how to do research, how to analyze and solve a problem.

I would also like to thank my colleagues, especially Yu Zhang, in Distributed Systems research team for their suggestions, discussions and help on my thesis.

Finally, I wish to thank my parents, my wife and my lovely nine months daughter.

Table of Contents

Lists of Figures

List of Tables

Chapter 1 Introduction

1.1 Motivation

Software agent has become an interesting model for distributed system design. Agent, as an autonomous entity, can interact with each other to fulfill global goals [Shoham93, Wood00]. A common approach to build a distributed multi-agent system can be abstracted using a coordination model [Ciancarini98]. Coordination can be defined as "the joint efforts of independent communicating actors towards mutually defined goals"[Arbab98, Ciancarini96, Malone94]. For example, a group of basketball players corporate to win a competition, and a flock of birds coordinate their flight in order to stay in formation. A coordination model provides a framework in which the interaction of software agents can be expressed, or in other words, these agents can be glued together. In general, a coordination model deals with the creation and destruction of agents, their communication activities, their distribution and mobility in space. More precisely, a coordination model for multi-agent systems can be refined into three entities: agent programming model, agent management model, and coordination media.

An agent programming model captures the significant properties of agents and provides the highest level abstraction to construct agents as the most important component from the point of view of an application. In order to meet their application requirements, programmers extend an agent programming model to customize the framework. In [Kendall98] they define the agent programming model in seven layers: sensory, beliefs, reasoning, action, collaboration, translation, and mobility. The layered pattern mimics the layered network model in which higher-level behaviors depend on lower-level capabilities. However, we picture the fundamental characteristics of an agent in a simple way as follows: reactivity, and pro-activeness [Zhang04]. Reactivity means that an agent can sense and react to stimulus coming from its environment or other agents. Pro-

activeness means that an agent can choose and control its behavior and cooperate with other agents. Interactions among agents are role-based [Kendall99, Riehle98]. Therefore, agent functionalities can be modeled by two types of behaviors: reactive behavior for reactivity and proactive behavior for pro-activeness. For example in e-commerce application, a seller agent can sell some products through negotiation, and sell other products through public sales. Thus a seller agent has to respond to asynchronous requests, and this is best modeled as a reactive behavior. On the other hand, if a buyer agent wants to purchase a product through negotiation, we can model the negotiation strategy of the buyer agent by defining a proactive behavior to negotiate with the seller agent to get a reasonable price.

An agent management model is used to manage the lifecycle of an agent. An agent often has a real-world counterpart relevant to the application. An agent plays certain roles, has certain goals to achieve, and goes through a lifecycle. Thus, an agent can be in one of following states: initiated, active, waiting, and dead. Initiation of an agent effectively creates and launches it. An agent is active when it is taking actions that are necessary to fulfill its goals. An agent is waiting when its progress is pending on the progress of its environment. An agent ceases to exist when it is aborted, for example, when it has completed its goals. For example in e-commerce application, a buyer agent abstracts a real customer, who is interested in buying some products. A buyer agent can be initialized by a customer with an item to be purchased and some constraints such as the price range. Then the buyer agent can actively search for the item from several markets and selects the best available. During the search process, the buyer agent may need to wait for the reply from a market. As the item is bought, its goal is achieved and the buyer agent can cease to exist. Different agent management model may define different agent lifecycle. For example in order to support agent mobility, a transit state may be added to the lifecycle described above to mark that an agent is in the process of migration. By

adopting an agent lifecycle, it provides a convenient way to manage agents. For instance if an agent is in the transit state, it will not be scheduled to execute. By being mobile, agents can operate in a host locally instead of remotely through the network. Hence it can save network bandwidth and processing latency, and massive communication can be avoided. Agent programming is thus simplified by relying on the services provided by an agent management model. Agent management model can also define other services according to system requirements, for example resource service and security service.

Coordination media is used to support agent interactions. Examples of such media include channels and tuple spaces. A channel supports message passing between two agents. A tuple space is a shared 'bag' that holds data and provides a set of primitives for agent interactions. Agents use the primitives to store and extract data. We call the set of primitives and their semantics a coordination language. In the application layer, in order to interact with each other through a coordination media such as a channel or a tuple space, agents have a communication language to understand each other. A communication language defines the syntax and semantics used in agent communication. For example, ACL (Agent Communication Language) [FIPA] can be used to design high level application protocols to support different types of agent coordination.

Agent management model and coordination media are basic infrastructures to provide services for agents. In other words, agent management model and coordination media are the basis in designing an agent platform.

There are various kinds of multi-agent platforms that support agent coordination via message passing. These include JADE (Java Agent Development Framework) by Telecom Italia Lab [JADE], FIPA-OS by Nortel Networks Harlow Laboratories in the UK [FIPA-OS], Grasshopper by IKV^{++} Technologies AG in Germany [Grasshopper], Aglets by IBM Japan Lab [Aglets], and ZEUS by British Telecommunications Laboratory [ZEUS].

Message passing is space and time coupled. An application designer must decide when an agent needs to communicate, with whom to communicate, and what data to send or receive. Usually, message passing is the natural and efficient solution for agent interactions. However, it is not always the most appropriate to use message passing. For example, in some dynamic sharing scenarios such as public sales in e-commerce, using message communication alone would require the incorporation of an agent to serve as a coordinator among seller agents and buyer agents. Message passing is also weak in its support of dynamic and asynchronous interaction among application agents.

Linda [Gelernter85, Gelernter92] is a form of distributed shared memory [Carter91, Kranz94, Li89] that is well recognized in its usefulness for coordination as it provides space and time decoupling. This means that one agent can access data without knowing whether it has been produced and who has produced it. However, the dynamic coupling supported by a tuple space comes with additional costs. The latency of interaction may be higher than that of a direct message exchange between two agents. It is because that implementation of Linda in a distributed system is still based on lower level message passing, and its consistency requires synchronization among different hosts.

A tuple space can be visualized as a shared associative memory that stores a set of tuples. A tuple is an ordered collection of fields, with each field having a type and a value associated with it. A template is a special tuple, for that a field of a template may only have a type without a value. A field without a value is called a formal field. A field with a value is called an actual field. Linda defines a simple set of primitives as the coordination language. These primitives are described in Section 1.2.1. An agent can read or retrieve (destructively read) a tuple by associative pattern matching between a tuple and a template. A tuple associatively matches a template if the following conditions hold. (i) The number of fields in the template is equal to the number of fields in the tuple. (ii) Each template field type is the same as the corresponding tuple field type. (iii) If a

template field has a value, then the value is equal to the value of the corresponding tuple field. Therefore, Linda allows an agent to access memory by associative matching instead of by address. For example, a mailbox can be modeled as a tuple space. A mail tuple may contain fields such as the receiver, the sender, mail-creation-time, mail-receipt-time and mail-content. Each of these fields has an actual value. One can access a mail tuple by providing a mail template in which only the receiver field and the mail-received-time field have actual values.

In our agent programming model, different agents play different roles at different times. A group of agents collaborates to fulfill some common goal. Hence it is natural to extend from the original Linda model, which defines a single flat tuple space, to multiple tuple spaces, so that different tuple spaces can be associated with different groups of agents. What is more, in order to improve system performance, it is necessary to distribute different tuple spaces or partition a single tuple space into subsets and store them across the network in order to take advantage of concurrency and locality. Therefore in order to design an efficient agent application, the multiple tuple spaces require a programmer to consider clustering related agents into a group through a tuple space, and scattering different tuple spaces and agent groups onto different nodes. This logical partitioning into disjoint tuple spaces provides concurrency while retaining the consistency semantics of a single tuple space formed by the union of these disjoint parts.

The above distributed multiple tuple spaces model is data-driven, which means that the involved agents will access a large body of shared data, examine them and act accordingly. The tuple space establishes how tuples can be stored and extracted from it. At this point, agents are treated as active entities, while the tuple space is a passive repository. As described before, data-driven mechanism is more suitable for applications such as e-commerce systems that have dynamic sharing requirements. On the other hand, in order to support agent reactive behaviors, we would like the tuple space to have event-

driven mechanism. Event-driven means the involved agents tend to center around processing or flow of control, and can observe state changes and react to occurrences of events. The tuple space has to establish how events and state changes can occur and how they propagate to the agents, and provide a means of registering and deregistering events for agents. Thus a tuple space is also an active coordinator to schedule and control agent activities by treating agents as passive workers and by treating certain tuples as events.

We would like to explore the features of such distributed multiple reactive tuple spaces to support our agent programming model. Finally, we want to integrate an efficient implementation of the multiple reactive tuple space with the JADE platform seamlessly, and to map our agent programming model to the agent model in JADE. The goal is not only to show that tuple space can meet both the dynamic sharing and the reactivity requirements, but also to demonstrate that tuple space can give us reasonable performance when compared with peer-to-peer message passing counterparts. Therefore, a multiple agent system can have both tuple space and message passing interactions support on one platform.

1.2 Related Work

1.2.1 LINDA

Proposed by David Gelernter and co-workers, Linda is the first tuple based communication and coordination model. Its principal use is to design a coordination language for parallel programming. Communication among different processes involved in a concurrent computation can only occur through a tuple space, which can be thought of as a bag into which tuples are inserted and retrieved by the processes. Linda has been used in a wide variety of applications, such as parallel string comparison, matrix multiplication and expert systems. For example in matrix multiplication, the master-slave

paradigm can be used. A master process first puts partitions of two matrices into the tuple space. The slave processes retrieves parts of the matrices. After computation, they put the partial results back into the tuple space. The master process is responsible for collecting all the partial results and combining them to get the final result. A more detailed description can be found in [Carriero90]. Since then, many extensions to Linda have been proposed and investigated. As we review these extensions, what we are concerned with most is the support for agent interactions.

Linda provides a coordination language, so called primitives, as follows. In order to clearly illustrate these primitives, we will highlight each primitive with figures by showing the changes to the tuple space caused by each primitive.

The primitive *out (tuple)* writes the given tuple into the tuple space. Figure 1-1 shows the state of the tuple space before and after a process has invoked *out (("a",69))*. Atomicity of the tuple space must be guaranteed. In other words, the results of access operations to the tuple space must be as if they have been performed in some serial order atomically. However, in actual implementation, different processes may be allowed to access different parts of the tuple space concurrently, as long as atomicity is not violated. The left sub-diagram shows the state before executing *out*. The right sub-diagram shows the state after executing *out*. The difference between the two states is the new tuple ("a", 69) that has been inserted. A pointer is used to show the control flow in executing process P. However, since it is non-blocking operation, the tuple may not appear in tuple space immediately as the invocation returns.

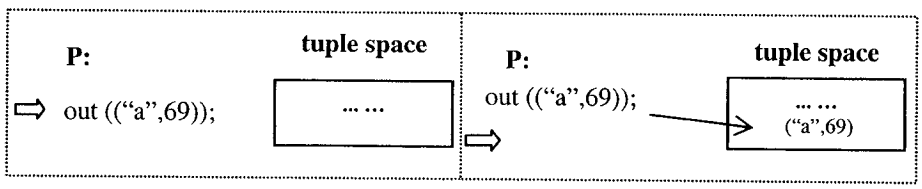


Figure 1-1 Tuple Space States as the Execution of *out*

The primitive *in (template)* attempts to match the tuple template with a tuple in the tuple

space and retrieve it. If a matching tuple does not exist, the operation blocks until a suitable tuple becomes available. If successful, the matching tuple is removed from the tuple space. Figure 1-2a shows the tuple space before and after *in(("a",?int))* is performed when a matching tuple is present. Figure 1-2b shows the case when a matching tuple is not found. Then the process has to be blocked until the required tuple appears in the tuple space. The middle sub-diagram shows the transient states that contain the matching tuple. During the time when the process is blocked, the tuple space may be accessed by other processes and change its state.

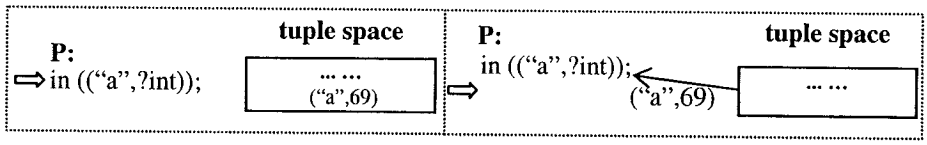


Figure 1-2a Case 1, Tuple Space States as the Execution of *in*

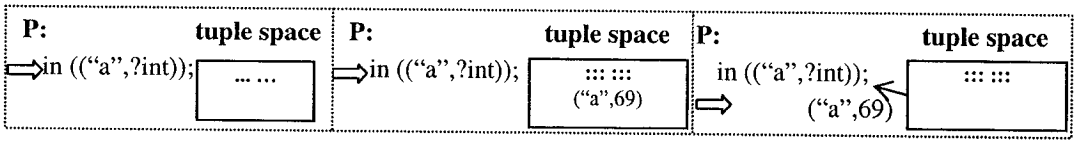


Figure 1-2b Case 2, Tuple Space States as the Execution of *in*

The primitive *inp (template)* functions in a similar way as *in* except that the process will not be blocked when a matching tuple is not found. Figure 1-3a shows the case when a matching tuple is in tuple space. Figure 1-3b shows the case when a matching tuple is absent and the process gets an empty (null value) tuple as response.

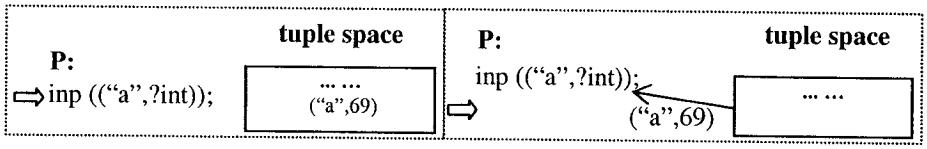


Figure 1-3a Case 1, Tuple Space States as the Execution of *inp*

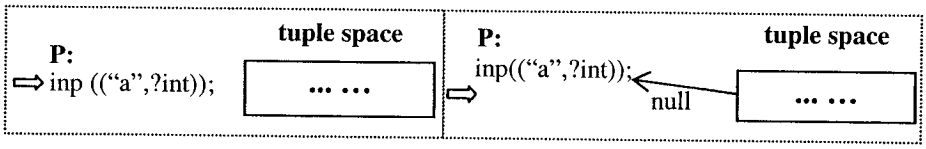


Figure 1-3b Case 2, Tuple Space States as the Execution of *inp*

The primitive *rd (template)* is similar to *in* except the matching tuple is not removed from the tuple space. Figure 1-4a shows the case where a matching tuple is found when a process invokes *rd(("a",?int))*. Figure 1-4b shows the case that there is no matching tuple.

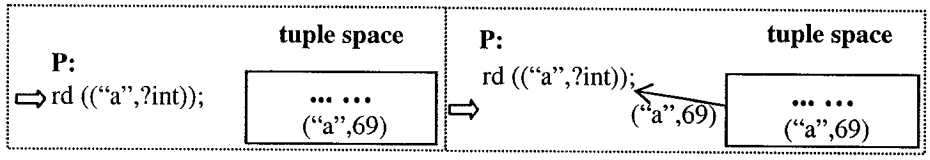


Figure 1-4a Case 1, Tuple Space States as the Execution of *rd*

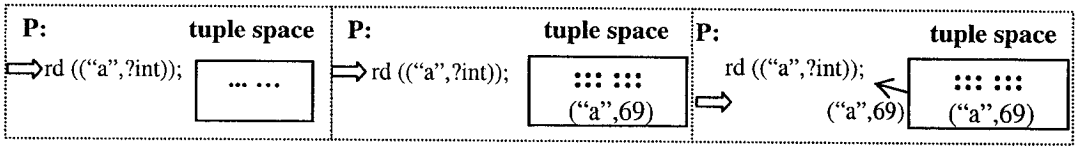


Figure 1-4b Case 2, Tuple Space States as the Execution of *rd*

The primitive *rdp (template)* resembles *rd* except that the operation does not block if a matching tuple is not found. Figure 1-5a shows the case when a matching tuple exists as a process invokes a *rdp(("a",?int))*. Figure 1-5b shows the case when a matching tuple cannot be found.

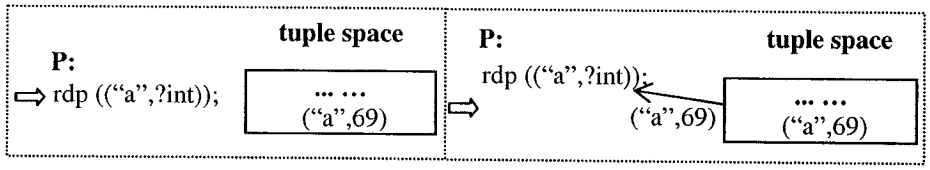


Figure 1-5a Case 1, Tuple Space States as the Execution of *rdp*

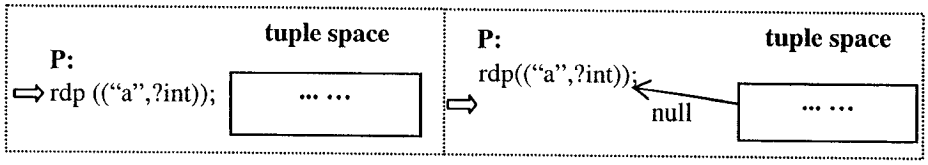


Figure 1-5b Case 2, Tuple Space States as the Execution of *rdp*

The primitive *eval (tuple)* creates a so-called *active* tuple. Each element of the tuple is evaluated concurrently. When completed, the resulting tuple is placed in the tuple space. It is a mechanism for spawning new processes. Figure 1-6 shows the tuple space before

and after a process has invoked *eval((10,?f(1)))*. In this example, Linda will create a process to execute the function *f(1)*. Here it is assumed that *f(1)* evaluates to become 18.

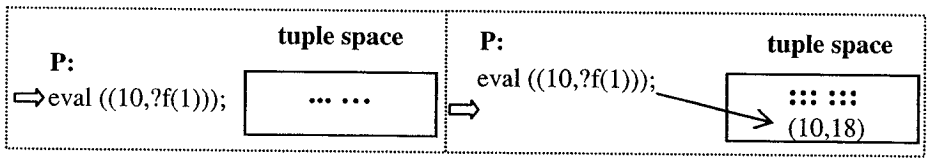


Figure 1-6 Tuple Space States as the Execution of *eval*

1.2.2 LIME

A wireless communication network is composed of physical mobile hosts and logical mobile units. Logical mobile units refer to program fragments such as software agents that can migrate from one host to another while preserving their codes and states. In LIME (Linda in a mobile environment) [Fok03, Picco99, Picco03], a logical mobile unit corresponds to a mobile agent. Physical mobile hosts refer to the portable computing devices such as laptops or PDAs that can host agents. They may roam across the network and provide physical connectivity among application agents and other relevant data. Connectivity is supported by wired and wireless links among hosts and can be altered by mobility or by explicit connections and disconnections. The characteristic of such wireless network favors a decoupled and opportunistic style of computation. Computation is decoupled in that it is expected to proceed even in the presence of disconnection. Computation is opportunistic in that it exploits connectivity whenever it becomes available. In order to assist the rapid development of mobile application, LIME provides a transiently shared tuple space to coordinate both physical and logical mobile units.

Transiently Shared Tuple Spaces As connections among hosts come and go over wireless network, it is natural to break a global tuple space into many individual tuple spaces and to introduce transient sharing among them. LIME provides three different levels of abstractions of tuple spaces. In LIME, each agent is permanently associated with

at least one *interface tuple space* (ITS). Each ITS contains information that the mobile agent is willing to share with others. An ITS actually can be interpreted as the local tuple space of an agent. Agents co-located on a same physical host creates a *host-level tuple space*, which can be regarded as the ITS of a mobile host. Hosts that are connected merge their host-level tuple spaces into a *federated tuple space*. The host-level tuple space and the federated tuple space are transient to application agents. They provide different levels of sharing among agents. Their contents are configured dynamically based on the connectivity. Figure 1-7 depicts the LIME model involving two hosts. At any time, an agent can access the transiently shared tuple space through its local associated ITS. Agents may have multiple ITS's, and the sharing rule among ITS's relies on the name. Only identically-named tuple spaces are transiently shared among the agents over the network. Thus there may be more than one host-level tuple space on a host and more than one federated tuple space over the network. For instance, suppose that there are three agents in the wireless network. An agent ***a*** owns a single ITS named **X** on a host named ***host1***. An agent ***b*** owns two ITSs named **X** and **Y** on the same host ***host1***, and an agent ***c*** owns a single ITS named **X** on another host named ***host2***. The ***host1*** and ***host2*** are disconnected at the beginning. Since the agent ***a*** and the agent ***b*** are on the same host, **X** becomes shared between the two agents. At this point, the shared **X** is a host-level tuple space. As the two hosts ***host1*** and ***host2*** become connected, LIME merges the three ITS's **X** from these three agents into a federated tuple space. At this point, **X** becomes a federated tuple space and shared among the three agents over the network. When the agent ***a*** now accesses **X**, it actually accesses the federated tuple space. However, since the agent ***a*** and the agent ***c*** do not have ITS **Y**, **Y** remains accessible only to the agent ***b***.

From the point of view of coordination, LIME offers Linda-like blocking primitives (e.g., *in, rd*), and probing primitives (e.g., *inp, rdp*) that apply to the entire transiently shared tuple space. In addition, LIME provides *out, in, inp, rd, rdp* primitives annotated with

locations. These will be described next. LIME also provides bulk primitives (e.g., *ing*, *rdg*). A bulk primitive is used to access more than one tuple at a time. For example, *ing* retrieves all tuples that match with the given template.

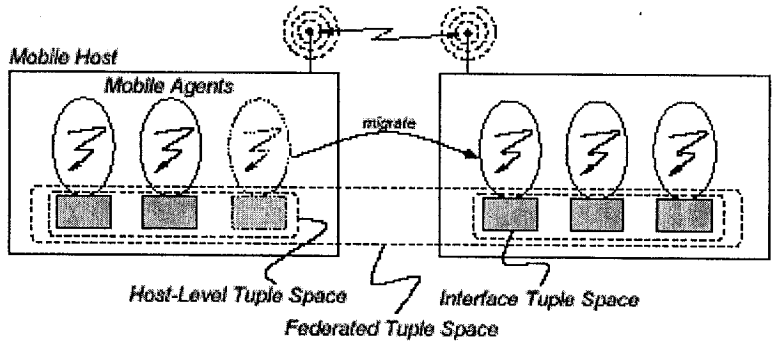


Figure 1-7 LIME Transiently Shared Tuple Space

Location-Aware Computing Transiently shared tuple spaces provide a global context shared by mobile agents. This could potentially simplify application designs. However, performance and efficiency considerations may require fine-grained control over a portion of the transiently shared space. For instance, an agent may be only interested in the information from a specific host. Thus by restricting its access to the specified host-level tuple space an agent can avoid the expensive query on the whole federated tuple space. Therefore, LIME extends Linda operations with tuple location parameters to allow an agent to confine its accesses to a portion of the transiently shared tuple space. A location parameter is defined with an agent identity or a host identity. The agent identity, defined by its IP address, port, and serial number, is used to identify the agent tuple space. The host identity, defined by IP address and port, is used to identify the host-level tuple space. Thus both identifiers must be globally unique.

In LIME, the *out* primitive is extended with a destination location parameter defined by an agent identity whose ITS is supposed to store the written tuple. A host identity value cannot be used as the location parameter in the *out* primitive. If the destination location is not specified, the tuple is stored into the writer's own ITS. For instance suppose that the

agent ***a*** identity is id_a, and the agent ***b*** performs *out(id_a, ("lime",10))*. If agent ***a*** and ***b*** are co-located, the tuple ("lime",10) will be written into the ITS of the agent ***a***. If they are not co-located, the tuple will become a misplaced tuple and will be stored in the ITS of ***b*** temporally. A misplaced tuple is a tuple that is attempted to be in some other agent's ITS but can not because of network disconnection. A misplaced tuple will be transferred to the destination once the needed network connection is properly established. Similarly, the *in, inp, rd, rdp* primitives are extended with both current location and destination location parameters. The current location is used to specify the portion of a transiently shared tuple space. It restricts the accessing scope from the entire federated tuple space to the tuple space associated with the current location. The destination location is used to identify misplaced tuples. In other words, it identifies these tuples that are attempted to be in the portion specified by the destination location, but currently are in the portion defined by the current location. The current location must be defined, and the destination location is optional. Consider *rd* as an example. If the destination location value is specified, it means that the agent wants to read a misplaced tuple that matches the template from the portion specified by the current location. Otherwise if the destination location is not defined, it means that the agent wants to read any matching tuple.

However, in order to restrict access to a portion of the transiently shared tuple space, an agent has to define the location parameter. Thus it may lose the significant property of spatial decoupling. For example if the location parameter is specified by an agent identity, communication between agents will become point-to-point, or equivalently message passing. Furthermore, when an agent restricts its access through the location parameter, it has to maintain connectivity in order to ensure correct communication.

Reacting to Changes In the rapidly changing environment, it is important for an agent to respond to an event as soon as possible. Events can be, for example, the availability of data carried by other agents. LIME introduces the notion of reaction $R(s, p)$,

which means that a code fragment s that implements a reaction will be executed by the tuple space manager when a tuple matching the pattern p is found in the tuple space. There are two kinds of reactions: strong reaction and weak reaction. A strong reaction is used to support reactivity over a host-level tuple space or the ITS of an agent. The semantics of strong reaction requires that the detection of a matching tuple and the corresponding execution of the reaction code to take place on the same host atomically. Thus the strong reaction reflects location-aware computing. This means that an agent is only interested in the events (tuples) from some location, for instance a host. On the other hand, in order to support reactivity spanning the federated tuple space efficiently, LIME provides a notion of weak reaction in which the execution of s does not happen atomically and immediately with the detection of a tuple matching p. However, it is guaranteed to take place eventually only if the connectivity among hosts is preserved and the execution of s takes place on the host on which the agent that registers the weak reaction resides. This means that even if the host in which an agent resides becomes disconnected from the network after the agent registered a weak reaction, it is guaranteed that a reactive event will trigger the weak reaction when a disconnected host re-connects with the network later. Moreover, there are two execution models to be selected for both strong reaction and weak reaction when registering them. They can be executed only once and then deregistered automatically in the same atomic step, or remain registered after the execution.

LIME demonstrates that it is useful in wireless games. Consider a jigsaw assembly game as an example. First, a player can still assemble pieces that she has picked up from the tuple space while disconnected. Second, players who are currently connected should be able to see the pieces assembled by others as soon as possible. The details of the game are described below.

Initially, a federated tuple space is formed using the tuple spaces of all the players in the same game. Two kinds of tuples, *image* and *assembly,* are used in the game. An image tuple represents a puzzle piece and contains two fields: the identifier and the bitmap of the piece. An assembly tuple contains a single field representing a list of the identifiers of the pieces that have already been assembled. As the game starts, an initial agent inserts one image tuple and one assembly tuple into the tuple space for each puzzle piece.

In order to continue assembling while disconnected, the pieces selected by a player shall be located in the player's *local interface tuple space.* That can be accomplished by the selection operation. It means that when a player joins a game, the player will first select the pieces that she wants to assemble. In detail, first both the image and assembly tuples are retrieved by invoking *inp* operations. Then it is followed by *out* operations (without specifying the location parameters) that reinsert the tuples into the player's local interface tuple space.

In order to see subsequent assemblies performed by other players, a player should register for a weak reaction for updating his screen. When puzzle pieces are assembled, the tuples representing the assembled pieces are removed from the player's tuple space using the *inp* primitive. Then a new assembly tuple is generated and inserted into the player's tuple space using the *out* primitive. The reaction will be triggered when the new assembly tuple is written into tuple space, and the screen will be updated.

It is easy to imagine that an implementation of such a game using ACL on a conventional mobile agent platform such as JADE will be rather complicated. First, in order to maintain the overall states of the game, an organizer agent has to interact with all the players. In LIME, the tuple spaces play this organizer role implicitly. Second, in order to update the screen in time, JADE agents will have to sample state to the puzzle assembly frequently and periodically. This consumes system resources unnecessarily.

From the above example, we can see that tuple space is an effective medium that allows mobile agents to couple asynchronously and promptly as permitted by the connectivity that exists. This effectiveness promotes programmability and resulting performance.

However, a programmer on LIME has to pay attention to the following.

(1) LIME supports two classes of agents: stationary agent and mobile agent. Stationary agent cannot migrate from one host to another. Strong and weak reactions associated with stationary agents are easy to use and implement. However, strong reactions associated with a mobile agent are a source of runtime error. Since the semantics of strong reaction require that both a triggering tuple and the reaction reside on a same host, this property may be violated when a mobile agent migrates from a host to another. For instance, suppose an agent registers a strong reaction at a host. Afterwards, the agent migrates to another host. The registered strong reaction cannot be triggered even when the needed tuple appears in the registration host.

(2) Since a federated tuple space spans across the network, the invocation of a primitive (e.g., *rd, in, register a weak reaction*) on the federated tuple space may span multiple hosts. If the number of spanned hosts increases, the system performance will slow down dramatically.

1.2.3 JavaSpace

JavaSpace is a JINI service [JavaSpaces]. JINI extends the java application environment from a single virtual machine to a network environment. It provides the infrastructure that allows clients and services to dynamically interact in a continually changing environment. One of the important services that JINI provides is the *lookup service*, which functions as a name server. Other JINI services require registration before a client can request these services. For example in order to initialize a JavaSpace service, it requires locating the lookup service using a discovery protocol first. Then the service is registered at the

lookup service. Similarly, a client can request this service from the *lookup service* and get the service proxy. Afterwards the client will be able to access the service through the proxy object.

Based on Java RMI and JINI, JavaSpace uses a unified mechanism. It includes five Linda-like primitives and one notification primitive for dynamic communication, coordination, and sharing of resources among processes over a wide heterogeneous network such as the Internet. Processes can be agents or any other components. The five basic primitives are *write, read, take, readIfExists,* and *takeIfExists*. Here they correspond to Linda *out, rd, in, rdp* and *inp* primitives respectively.

In addition, JavaSpace defines two additional new futures for transaction and leasing.

(1) Transaction can group multiple primitive invocations that access multiple JavaSpaces into a bundle that acts as single atomic operation. However, if an agent chooses to use a primitive with transaction support, its semantics will be different from the one without such support. For example, *write* primitive under a transaction means that the tuple that is written is not visible outside its transaction until the transaction successfully commits. If the tuple is taken within the transaction, the tuple will never be visible outside the transaction and will not be added to the space when the transaction commits. As a side-effect, the tuple will not generate notifications to listeners that are not registered under the transaction. If a transaction aborts, then the tuples that are written under this transaction will be discarded.

(2) Leasing means that when a process writes a tuple into the tuple space, it also declares a lifetime for the tuple. When its lifetime expires, a tuple is removed from the space. For instance a public sale scenario in e-commerce, a seller agent needs to promote some products for one week. The seller agent can put these product tuples with one-week leasing time into the market tuple space. Thus it ensures that when the week is over, these products in promotion will be removed.

JavaSpace also introduces event notification feature in the form of the *notify* primitive. This registers a remote event listener to be called when a tuple matching the template becomes available. However, *notify* does not return a tuple to the agent. Instead, it only returns a message indicating that a matching tuple has been inserted into the tuple space. Subsequently, the reaction part of the agent will have to retrieve the matching tuple from the tuple space. However, this retrieval may fail if the tuple has already been retrieved by another agent. Hence notification is not atomic, and critical race could result in the subsequent behavior of the notified agent. In addition, the event notification does not support agent location transparency. This means that if an agent registers an event listener and then migrates to another host, the agent will not be notified even if a matching tuple is inserted into the tuple space.

Many applications that require coordination among distributed components, for example a workflow application, can be designed using JavaSpace. The first application that JavaSpace publicly demonstrated is the animation of the movie Toy Story, in which JavaSpace is used to distribute the processes among an array of hosts running in parallel. The basic idea involves the cooperation of a farmer process, several worker processes and a reaper process. A farmer process places distinct units of data to be processed into JavaSpace. Several worker processes pick up data to be processed from JavaSpace and then put the results back into JavaSpace. The reaper process collects the processed data and transforms them into the required output.

It may not be surprising that there are application scenarios that JavaSpace does not support efficiently. For example in e-commerce application, suppose several customers want to query about TV's from a market place implemented by tuples and there are at least two relevant TV tuples. Such a scenario is described as the *multiple read problem* in [Rowstron96]. One way to realize this scenario in JavaSpace is to serialize the query by using a lock. A customer first acquires the lock, and then retrieves all the matching TV

tuples one by one by using *take* primitive. Afterwards, the customer places all the matching TV tuples back to the market tuple space, and then releases the lock. However, this scenario can be easily implemented by using bulk read primitives, such as the *rdg* primitive in LIME.

Moreover, JavaSpace is a JINI service whose reference implementation comes with Sun's JINI Technology Startup Kit. It is a little complicated to deploy JavaSpace because that the underlying infrastructures have to start first. In order to have a JavaSpace service in a system, the following services have to be started up in order: HTTP server, RMI activation service, lookup service, transaction service, and finally JavaSpace service. Therefore, it incurs relatively expensive overhead for simple agent missions. Further, since a JavaSpace service is centralized, multiple JavaSpace services have to be registered in order to have multiple tuple spaces.

1.2.4 MARS

MARS (Mobile Agent Reactive Spaces) [Cabri00] is designed as reactive tuple spaces to support mobile agents over the Internet. The core idea is to provide programmable tuple spaces through the notion of reaction. Reaction is defined as a programmable unit associated with a triggering event. An event is composed of a tuple template, an operation type and an agent identity. The operation type points out which primitive is invoked and the agent identity specifies the ID of an agent. A null value in the latter means an arbitrary agent. The event can be reading or taking a tuple from a tuple space, or writing a tuple into a tuple spaces by the identified agent. A reaction is a stateful object with a single interface. The single interface is called *reaction()*, whose input is the triggering event and output is a tuple. Examples of how to use the reactions will be shown later. In MARS, there is a meta-level tuple space that is used to store and manage these programmed reactions. Every reaction exists as a specific tuple in the metal-level tuple

space. An agent can manipulate these reactions through the specific primitives such as *deleteReaction*.

Based on JavaSpace service, MARS reactive tuple space model adapts JavaSpace specification, and provides blocking primitives (e.g. *read, take*) and bulk primitives (e.g. *readAll, takeAll*). MARS also provides reaction primitives (e.g. *createReaction* to register a reaction, which constructs a reaction tuple in a metal level tuple space. *deleteReaction* to deregister a reaction, which removes the corresponding reaction tuple). Usually, reactions are registered and deregistered through a system tool in MARS.

In MARS runtime, a reaction is triggered by an event, and executed on the tuple space side by the tuple space manager. The most important feature of a reaction is that it can access a tuple space, change its content, and influence the semantics of the primitives invoked by agents. For example when an agent uses *read* primitive to read a tuple from tuple space, before returning the matching tuple, if a reaction has been registered with such event, the reaction will be executed. It may modify the matching tuple and then return the modified one to the agent. As another example, when an agent uses *write* primitive to place a tuple into tuple space, a reaction is triggered. The reaction may override the write operation and insert another tuple instead of the original one into tuple space. Thus, reactions can be thought of an adaptation layer over tuple space to react to agent accesses in a customized way.

MARS can be easily programmed to incorporate security into its tuple space. For example, we may allow any agent to read but only authorized agents to take tuples from tuple space. In order to meet this requirement, the administrator can program a reaction to monitor invocations of the *take* primitive. If an agent attempts to take a tuple, the reaction will check whether the agent is authorized. If not, the reaction may take a log and return a null value to the agent.

However, the reactive behavior of our agent programming model requires that a tuple

space have the ability to monitor some interested tuples and take actions whenever they are inserted into the tuple space. According to the definition, MARS reactions are not capable of providing such support for agent reactivity. In Chapter 2, we will give more illustration when introducing the reactive primitives. The semantic of MARS reaction is also different from that of LIME, and JavaSpace.

Presently, MARS only provides mobile agents the reference of local tuple space. Thus mobile agent cannot access remote tuple spaces, and agents coordinate with each other only through the tuple space of local host.

1.2.5 WCL

WCL [Rowstron98] is a coordination language designed for distributed agents over the Internet. WCL has synchronous access primitives (e.g. *out_sync, in_sync, rd_sync*). Synchronous means that the operation blocks the agent until the operation is finished. Emphasizing on efficiency, WCL also proposes asynchronous access (e.g. *out_async, in_async, rd_async*), bulk access (e.g. *bulk_in_async, bulk_rd_async, move_async*) and event monitor (e.g. *monitor*) primitives. The detail description of all these primitives can be found in [Rowstron98]. An asynchronous primitive is handled by a system thread which does not block the agent from progressing. The agent will get a handle afterwards and can check the handle later at any time for the result. Asynchronous tuple access allows an agent to tolerate long access latency for tuples that may or may not have been created in time to be used.

It has been demonstrated that bulk access primitive *bulk_rd_async* is helpful in multiple read scenarios described in Section 1.2.3. In WCL, the unique feature of the bulk primitives is that WCL runtime system maintains control of the matching tuples and treats them as a stream. An agent gets the matching tuple from the stream, one at a time, until the end of the stream. WCL runtime can use a lazy protocol to transfer matching

tuples to an agent from a remote server in order to save system bandwidth. However, this may slow down the application as the number of remote transfer increases. Moreover, it is difficult for an agent to construct a local data structure to hold these matching tuples for later use, since the agent does not know the number of the matching tuples before reaching the end of the stream.

The *monitor* primitive specifies a template. All tuples within the tuple space that match with the template are streamed back to the agent. Subsequently, any tuples inserted are also streamed to the agent. With the *monitor* primitive, WCL has a simple event mechanism to ensure a condition can be specified once and used forever. For example, suppose that e-mails are being stored in a tuple space. An e-mail agent can watch for email arriving simply by using the *monitor* primitive, and 'be informed' whenever a piece of email arrives. However, since WCL does not define a general agent framework, it lacks the ability to trigger the agent behavior automatically. Instead, an agent is responsible for checking proactively for the arrival of tuples and act on them accordingly. Therefore, what the 'be informed' means is that the mail agent itself has to check whether there are emails arriving, and then executes its reaction. Usually, an agent may get nothing and waste machine cycles.

There are some scenarios that WCL cannot support very well. For example, in an e-commerce application, a buyer agent wants to check whether there is a specific type of TV in a market tuple space. It is not suitable to solve the problem using synchronous primitive *rd_sync* since it blocks the buyer agent if there is no such specific TV. Although the asynchronous primitive *rd_async* does not block the buyer agent, where to put the code fragment of checking becomes an issue. Moreover, it still cannot be sure that there is no such specific TV in the market even if the checked result is false, as an agent has no way to know whether the request has been processed or not by the tuple space server. The best way to solve such scenario is to use probing primitive *rdp*. However,

WCL does not support probing primitives.

Attempting to achieve language transparency, WCL and its runtime system are designed to be independent of any host language. Currently, WCL has been embedded into C++ and Java. Agents written in C++ can communicate with agents written in Java. However, in order to support this, there must be some sort of mapping between types in the two different languages. In the current implementation, WCL supports only simple built-in type variables. Without user-defined types supporting, it is difficult to place an actual domain object into a tuple. In order to do so, a programmer has to declare several built-in type variables instead of one well-defined object, and pack these variables as different fields into a tuple. Therefore, it is in contradiction to the principle of information encapsulation and such systems are difficult to maintain.

1.2.6 Logic Operator Linda

In previous multiple tuple space environments such as WCL, the defined primitives only provide an agent the ability to access one tuple space at a time. This means that an agent that requires access to several tuple spaces has to do so by serializing the access. In [Snyder02], they define logical operator primitives to enables an agent to coordinate its tuple accesses across multiple clusters of collaboration simultaneously. Logical operator primitives extend traditional Linda primitives. They specify the combination of multiple tuple spaces with logical operators as the input parameter. These logical operators include AND, OR, and NOT. For simplicity reason, multiple tuple spaces can only be combined with just one type of logical operator in current version.

As an example, the *rd* with the AND operator is performed by searching a specified list of tuple spaces for the tuple. If none of these tuple spaces contains a matching tuple, the agent blocks. If the AND operators specifies n tuple spaces, the agent will finally get a list containing n tuples, one from each tuple space. The semantic of *rd* with the OR

operator is to search for at least one tuple matching the given template from a list of tuple spaces. At the same time, not more than one tuple from each tuple space will be returned to the agent.

In e-commerce, suppose, a buyer agent wants to search for a specific type of TV from two market tuple space. Without logical operator primitives, the agent will probe the two markets sequentially to search for the TV. Bulk access primitives enable this to be performed in a single operation. This simplifies programming and enhances performance through the improved concurrency.

However, current logical operator primitives only apply logical operators to tuple spaces, but not to templates. Thus in such system, one cannot express the following requirement: retrieve a tuple which is stored in either tuple space *ts1* or tuple space *ts2*, and matches either template *A* or template *B*. Such a feature will be part of the reactive tuple space that will be implemented in this thesis.

1.3 Contributions and Outline

As we see from the related work, a lot of efforts have been spent on the extensions of tuple space. JavaSpace is primarily a JINI service. WCL supports agent coordination efficiently with language transparency. MARS intends to influence the semantics of primitives through a programmable reactive tuple space. LIME targets a wireless environment. Logic operator Linda introduces multiple tuple space access as a single operation. These extensions involve different primitives. For instance, JavaSpace does not provide bulk primitives and WCL does not have probing primitives. In the previous section, we have already seen that it may be harder to implement some application scenarios when certain primitives are absent.

Agents can have both proactive behaviors and reactive behaviors. With proactive

behaviors, agents can interact with each other synchronously and asynchronously. With reactive behaviors, agents can quickly react to stimulus from the environments. Thus we propose a distributed tuple space to support agent interactions.

In order to support synchronous interaction, we provide traditional Linda like blocking and probing primitives, probing bulk primitives. In order to support agent interaction asynchronously, we have Linda like primitives in asynchronous form, asynchronous bulk primitives and asynchronous logical template primitives. The idea of bulk primitives is from WCL. WCL only have asynchronous bulk primitives. However, WCL treats the matched tuples as a stream. An agent has no idea of the number of the matched tuples before reaching the end of stream, while we give bulk primitives with no such limitation. Logical operator primitives enable an agent to access multiple tuple spaces simultaneously. We extend the use of logic operators to both tuple spaces and templates and we call these logical template primitives.

In order to support agent reactive behaviors, we define reactive primitives through which an agent can register a reaction with location transparency. JavaSpace event notification does not support location transparency. WCL can only stream back events to agents but lacks a framework to notify and triggered agent behaviors. Although LIME has the weak reaction with location transparency, it is for wireless environments and its implementation is based on the three levels of abstraction. Our multiple tuple spaces do not have such structural relationship. Furthermore, with reactive primitives, an agent can possess fine-grain control of an event: a complex condition can be defined and such a condition is used to check the event first on the server side before triggering a reaction remotely.

The distributed multiple reactive tuple spaces have been fully implemented and integrated with the JADE platform. Therefore, the availability of these primitives enables flexible and efficient agent collaboration.

In order to have an efficient implementation, we design a replication protocol to facilitate the transfer of tuples to potential agents in advance. The replication protocol makes decision based on the application runtime information collected by the kernel. The correctness of the replication protocol has also been verified.

Finally, we compare the performance of agent coupling realized using tuple spaces and that realized using ACL message passing on the JADE platform. We conclude that the performance of single or multiple tuple space is compatible with that using ACL message passing.

The rest of the thesis is organized as follows. Chapter 2 gives a detailed description of the access primitives of the reactive tuple space. Chapter 3 presents the replication protocol and its correctness proof. Chapter 4 presents the tuple space architecture and detail design. Chapter 5 contains a performance test of agent coupling via tuple spaces and ACL messaging. Chapter 6 concludes the thesis with some projection of future work.

Chapter 2 Agent Interaction and Coordination Primitives

2.1 Agent Interaction Problem

In a multi-agent system, the agent interaction can be synchronous coupling or asynchronous coupling. Synchronous coupling means that the interaction between two agents is interleaved, and one agent needs to get necessary information from the other agent before deciding the next action. For instance in the auction protocol in e-commerce application, a bidder agent has to wait for the current price announced by the auctioneer agent in order to propose a price. The auctioneer agent, in turn, receives the proposed prices and then makes a new announcement. In general, most agent interaction protocols are synchronous. On the other hand, asynchronous coupling means that an agent does not block for some information from the other agent. However, when the information shows up, the agent will proceed to perform the corresponding action/reaction.

Agent interaction is used to describe the communication relationship between two agents. On the other hand, agent coordination is used to describe that a group of agents needs to interact with each other in order to achieve a common goal. It includes multiple agent interaction scenarios from the group of agents. Thus, these agents that involve the coordination scenario share a consistent view about the application states and form a dependency relationship. One coordination example is that flying birds try to stay in formation as described in Chapter 1. In this case, every bird may see the similar route and speed, and birds that are close to others have to adjust themselves through some interaction in order to keep a proper distance from each other.

In order to support agent interaction, an agent is abstracted to have proactive behaviors and reactive behaviors in an agent programming model. A proactive behavior typically involves synchronous coupling and a reactive behavior involves asynchronous coupling.

Both proactive behaviors and reactive behaviors can be implemented through either message passing or tuple space. The advantage of using tuple space is that it not only supports dynamic sharing efficiently, but also provides a consistent view for the group of agents. Thus, we like to investigate the use of tupe space as coordination media to facilitate agent interaction.

Agent proactive behavior can be supported by synchronous primitives and asynchronous primitives defined by tuple space. From a programmer's point of view, agent interaction can be abstracted by high level interaction protocols. Most interaction protocols are designed to be synchronous. Naturally, it requires the tuple space to provide synchronous primitives. On the other hand, asynchronous primitives provided by tuple space also support synchronous coupling. The advantage of asynchronous primitives is that it can improve agent performance since an agent can continue its computation concurrently before the result is ready. Especially in case that the agent model is logically multiple threaded but the agent is physically mapped to a single thread, as in the JADE agent programming model. When such an agent plays multiple roles, any blocking in a role will block all roles. Therefore, for such cases the asynchronous primitives seem to become the first choice. It is not only for performance reason, but also for having correct application behaviors. For every asynchronous primitive invocation, a thread has to be generated to perform the task. Overuse of the asynchronous primitives may slow down application performance, since it takes too much time to schedule threads.

An agent has to get the needed information in order to proactively interact with others. Some information is from inherent knowledge of the agent, and others may be from its outside environment such as other agents. For a tuple space, the necessary information is contained in the tuples. On one hand, an agent may be interested to retrieve any one tuple that matches the specified requirement defined by a template. On the other hand, an agent may want to retrieve all the tuples that match the specified template from the single tuple

space, which is what the bulk primitives do. Further, in some scenarios, giving an agent the ability to retrieve tuples from multiple tuple spaces jointly and simultaneously is helpful for agent performance.

The reactive behavior is another important aspect of our agent programming model. The main logic of an agent is described through its interaction protocol and captured by its proactive behaviors. However, there are some scenarios that are more suitably supported with reactive behaviors. For instance, in order to facilitate system administration, some agents may listen to the instructions from system administrator through reactive behaviors. Another example has been introduced in Chapter 1. The common characteristic among these scenarios is that it is impossible to predict when an event that interests an agent will happen, and actually it may never happen. It is necessary for a tuple space to provide a sub-framework to help an agent to define and register reactions for some specific events. When an event happens, the corresponding reaction will be notified. An event is expressed by a tuple in the tuple space. Therefore, we have a set of reactive primitives to support the agent reactive behavior.

2.2 Interaction Primitives

As described in previous section, the set of primitives supported through tuple space can be classified into three categories: synchronous, asynchronous and reactive (notification) primitives. In addition, in order to facilitate programming as well as to improve the resulting system performance, we extend each synchronous or asynchronous primitive into three sub-categories: single access, bulk access and logic template based access. In designing the primitives, we do not require that it have both the synchronous and asynchronous primitives. For instance, we do not have a synchronous logic template based primitive which can be easily simulated with the corresponding asynchronous ones. Figure 2-1 outlines the set of primitives that we have implemented in Java. In the

following sections, we will informally describe the various tuple space primitives before formally specifying them using the state machine model [Weihl93]. The use of these primitives has been demonstrated through an e-commerce application in [Zhang04].

```
interface TupleSpaceService
{
 public TupleSpaceID TSCreate(String name);
 public TupleSpaceID TSFind(String name);

 //synchronous single tuple access
 public Tuple   in(TupleSpaceID tsId, Tuple atemplate);
 public Tuple   in(TupleSpaceID tsId, Tuple atemplate, long timeout);
 public Tuple   read(TupleSpaceID tsId, Tuple atemplate);
 public Tuple   read(TupleSpaceID tsId, Tuple atemplate, long timeout);

 //asynchronous single tuple access
 public void    asynOut(TupleSpaceID tsId, Tuple atuple);
 public Future  asynIn(TupleSpaceID tsId, Tuple atemplate);
 public Future  asynRead(TupleSpaceID tsId, Tuple atemplate);

 // synchronous bulk primitives, which manipulate more than one tuple at a time
 public int   move(TupleSpaceID ts_source, TupleSpaceID ts_dest, Tuple atemplate );
 public int   copy(TupleSpaceID ts_source, TupleSpaceID ts_dest, Tuple atemplate );
 public TupleSet bulkInWithoutWait(TupleSpaceID tsId, Tuple atemplate);
 public TupleSet bulkReadWithoutWait(TupleSpaceID tsId, Tuple atemplate);

 // asynchronous bulk primitives, which manipulate more than one tuple at a time
 public Future asynMove(TupleSpaceID ts_source, TupleSpaceID ts_dest, Tuple atemplate );
 public Future asynCopy(TupleSpaceID ts_source, TupleSpaceID ts_dest, Tuple atemplate );
 public Future bulkAsynIn(TupleSpaceID tsId, Tuple atemplate);
 public Future bulkAsynRead(TupleSpaceID tsId, Tuple atemplate);

 // logic-template-based asynchronous access
 public Future  asynIn(LogicTemplate alogic_template);
 public Future  asynRead(LogicTemplate alogic_template);

 // reactive primitives
 public AgentRegisterID  register(TupleSpaceID tsId, Tuple atemplate, IReactive ref);
 public AgentRegisterID  register(TupleSpaceID tsId, EventChecker checker, IReactive ref);
 public void deregister(AgentRegisterID  register_id);

//for replication purpose
 public void subscribeForReplication(TupleSpaceID tsId, Tuple atemplate);
}
```

Figure 2-1 Interaction Primitives

2.2.1 Tuple and Template Matching Rules

As introduced in Chapter 1, a tuple and a template are matched only if they have the same number of fields and every corresponding field is matched. However, the matching rules of two fields are slightly different from that introduced in Chapter 1. It is because that these primitives are implemented in Java, and we model the tuple and field as containers that may contain any object that can be serialized. In detail, suppose f1 is a tuple field and f2 is a template field, both are matched by the following rules.

(1) If f2 is a formal field, they are matched only if the types are matched. It means that both fields have same types or the type that f2 is the subtype of that of f1. For instance, there are two classes class *A* and class *B*. The class *B* inherits from class *A*. A field *field1* contains an instance of class A, and a field *field2* is a formal field which only has the type of class B without instance. Thus *field1* and *field2* are matched.

(2) On the other hand, if f2 is an actual field, they are matched only if both fields have the same types and with the same value.

2.2.2 Tuple Space Creation and Find

(1) *TSCreate(String name)*: It creates a tuple space with the given name. The primitive returns a TupleSpaceID *tsId*. The location of tuple space created is transparent to agents. This primitive enables an agent initially create a tuple space and shares with its group member. The group member will access the tuple space by name.

(2) *TSFind(String name)*: It finds an existing tuple space by the given name. The primitive returns a TupleSpaceID *tsId*. This primitive enables an agent to join a group of agents if it is aware of the name of the tuple space.

2.2.3 Synchronous Single Tuple Access

(1) *in(TupleSpaceID tsId, Tuple atemplate)*: It has the same semantics as Linda-like *in* primitive described in Chapter 1. It retrieves (destructive read-out) a tuple that matches with the template *atemplate* from tuple space *tsId*. Since the primitive is synchronous, the agent will block until a matching tuple is available.

(2) *in(TupleSpaceID tsId, Tuple atemplate, long timeout)*: It retrieves (destructive read-out) a tuple that matches with the template *atemplate* from tuple space *tsId*. Since the primitive is synchronous, the agent will block until a matching tuple is available or timeout is triggered. If timeout is triggered, an empty tuple (null value) is returned. This primitive can simulate the probing Linda primitive *inp*. Figure 2-2 shows the tuple space *ts1* before and after states when there is no matching tuple as *in(ts1,("a",?int),100)* is invoked. If a matching tuple such as ("a",69) is inserted into the tuple space before the timeout is triggered, the process will retrieve the matching tuple. Otherwise the process gets an empty tuple (null value).

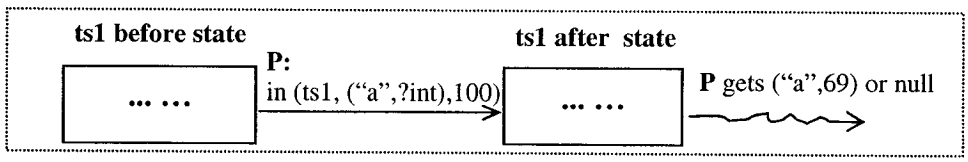


Figure 2-2 Tuple Space States as the Execution of *in*

(3) *read(TupleSpaceID tsId, Tuple atemplate)*: It has the same semantic as *in(tsId, atemplate)* primitive except that the matching tuple is not removed.

(4) *read(TupleSpaceID tsId, Tuple atemplate, long timeout)*: It has the same semantics as *in(tsId, atemplate, timeout)* primitive except that the matching tuple is not removed.

2.2.4 Asynchronous Single Tuple Access

(1) *asynOut(TupleSpaceID tsId, Tuple atuple)*: It insert a tuple *atuple* into tuple space *tsId*. When the primitive terminates, the tuple *atuple* is not guaranteed to be present in the tuple space *tsId*. However, it will appear in the tuple space as soon as possible. It has the

same semantic as Linda-like *out* primitive. Since an agent does not need to receive a response when writing a tuple into the tuple space, we do not provide *out* primitive in synchronous form.

(2) *asynIn(TupleSpaceID tsId, Tuple atemplate)*: It asynchronously retrieves a tuple that matches with the template *atemplate* from tuple space *tsId*. The primitive is asynchronous and the agent will not be blocked. It returns a tuple-holder object (called *future*). The agent can check the future object to see if the matching tuple has become available, and fetch it from the future object locally. This enables an agent to asynchronously interact with another agent without stalling when the latter is not ready. Figure 2-3 shows the tuple space *ts1* before and after states as a process invokes a *asynIn(ts1,("a",?int))* primitive. Usually the tuple space state will not be changed immediately as the *asynIn* invocation terminates. However, it will be changed when the matching tuple is removed.

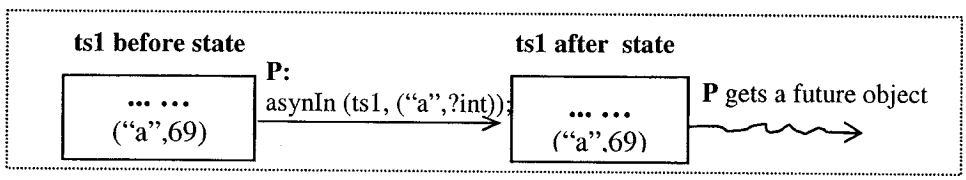


Figure 2-3 Tuple Space States as the Execution of *asynIn*

(3) *asynRead(TupleSpaceID tsId, Tuple atemplate)*: It has the same semantics as *asynIn(tsId, atemplate)* primitive except that the matching tuple is not removed.

2.2.5 Synchronous Bulk Access

(1) *move(TupleSpaceID ts_source, TupleSpaceID ts_dest, Tuple atemplate)*: It moves all the tuples that match template *atemplate* from tuple space *ts_source* to tuple space *ts_dest*. In fact, it consists of two operations: one is to retrieve tuples from source tuple space, and the other is to write these tuples into the destination tuple space. It returns an integer as the number of tuples that have been moved. If there is no matching tuple in the source tuple space, it returns zero to the caller. This primitive enables an agent to manage the

shared tuples according to its roles. Sometimes it is beneficial for an agent to move some chosen tuples from one space to another as part of its responsibility in coupling with other agents through multiple roles. Figure 2-4 shows that the state change as *move(ts1,ts2, ,("a",?int))* is invoked. As the primitive terminates, the matching tuple has been moved from ts1 to ts2 and corresponding states changed.

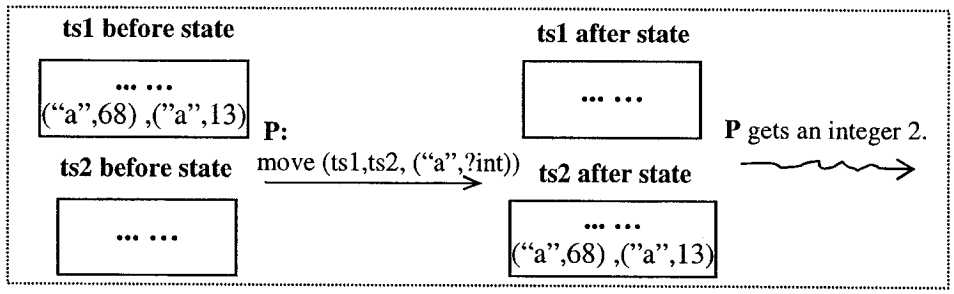


Figure 2-4 Tuple Space States as the Execution of *move*

(2) *copy(TupleSpaceID ts_source, TupleSpaceID ts_dest, Tuple atemplate)*: It has the same semantic as *move* primitive except that the matching tuples will not be removed from the source tuple space. It is like duplicating the tuples.

(3) *bulkInWithoutWait(TupleSpaceID tsId, Tuple atemplate)*: It retrieves all the tuples that match with template *atemplate* from tuple space *tsId*. Figure 2-5 shows the state change as a process invokes *bulkInWithoutWait(ts1, ("a",?int))*. The process gets a *TupleSet* object that contains all the matching tuples. In this example, there are two matching tuples. If there is not even a matching tuple, an empty tuple set is returned.

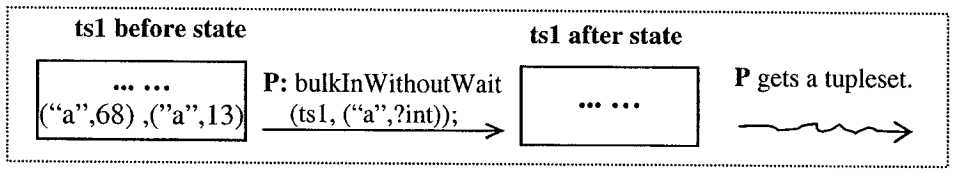


Figure 2-5 Tuple Space States as the Execution of *bulkInWithoutWait*

(4) *bulkReadWithoutWait(TupleSpaceID tsId, Tuple template)*: It has the same semantic as the *bulkInWithoutWait* primitive except that the matching tuples will not be removed.

2.2.6 Asynchronous Bulk Access

(1) *bulkAsynRead(TupleSpaceID tsId, Tuple atemplate)*: It reads all tuples that match the template *atemplate* from tuple space *tsId*. It returns a tuple-holder object (called *future*). The agent can check the *future* object for the matching tuples locally. This primitive enables an agent to read more than one tuple at a time and overcomes the problem of multiple read. When a process invokes a *bulkAsynRead* primitive, the tuple space state will not be changed because it only read the matching tuples.

(2) *bulkAsynIn(TupleSpaceID tsId, Tuple atemplate)*: It has the same semantic as the *bulkAsynRead* primitive except the matching tuples will be removed from tuple space *tsId*. When a process invokes a *bulkAsynIn* primitive, the tuple space state will not be changed immediately as the invocation terminates. However, the state will be changed as soon as the matching tuples are actually retrieved.

(3) *asynMove(TupleSpaceID ts_source, TupleSpaceID ts_dest, Tuple atemplate)*: It moves all the tuples that match template *atemplate* from tuple space *ts_source* to tuple space *ts_dest*. It returns a tuples-holder object (called *future*). The agent can check the *future* object for the number of tuples that have been moved. When a process invokes a *asynMove* primitive, both tuple spaces' states will not be changed immediately as the invocation terminates. However, the states will be changed as the matching tuples are actually moved to the destination tuple space from the source tuple space.

(4) *asynCopy(TupleSpaceID ts_source, TupleSpaceID ts_dest, Tuple atemplate)*: It has the same semantics as the *asynMove* primitive except that the matching tuples are not removed from tuple space *ts_source*. When a process invokes a *asynCopy* primitive, the source tuple space state will never be changed because it does not remove the matching tuples, and the destination tuple space state will not changed immediately as the invocation terminates. However, its state will be change as the matching tuples are copied into it.

2.2.7 Logic Template Based Asynchronous Access

(1) *asynRead(LogicTemplate alogic_template)*: This primitive reads those tuples that match with the logic template a*logic_template* asynchronously. It returns a holder of tuples (called *future*). The agent can then fetch the matching tuples from future object locally when they become available. This primitive enables an agent to read tuples across multiple clusters of collaboration simultaneously.

To clarify logic-template, its syntax in *BNF* notation is given below. Suppose that *EXP* represents the logic- template:

<EXP> ::= *<OP>(<ARGS>)*
<ARGS> ::= *<ARG>* | *<ARG>,<ARGS>*
<ARG> ::= *(tsId, template)*| *<EXP>*
<OP> ::= *and* | *or*

The semantic of *and* operator is that it supports retrieval of tuples from the tuple spaces identified by its both left and right statements (similarly as if both statements evaluate to be true). The semantic of *or* operator is that it supports retrieval of tuples from the tuple spaces identified by either left or right statements (similarly as if at least one evaluates to be true). When a process invokes a logic template *asynRead* primitive, the states of tuple spaces will not be changed because it only read the matching tuples from tuple spaces.

(2) *asynIn(LogicTemplate alogic_template)*: It has the same semantic as the logic template *asynRead* primitive except that the matching tuples are removed from their tuple spaces. When a process invokes a logic template *asynIn* primitive, usually the states of tuple spaces will not be changed immediately as the invocation terminates. However, it will be changed as the matching tuples are removed.

2.2.8 Reactive Support

Reaction is introduced to model agent reactive behaviors and support agent asynchronous coupling. A reaction is defined through the reactive interface called *Ireactive*, which has

only one method *reactTo(AgentReactionEvent ev)*. It is modeled as a serialized local object and collocated with the agent. The parameter *AgentReactionEvent* provides the event and its source agent. An event is a tuple. For a programmer, defining a reaction is to implement the *Ireactive* interface, and the slice of code is just the logic of how the agent reacts to the specified event. At runtime, the kernel keeps track of agent location and knows where to notify to the agent if it has migrated to another host. What is more, the execution of reaction happens asynchronously with the detection of an event. In detail, when an event detector finds a matching event, the detector only sends a notification message (which is a *AgentReactionEvent* object) to the reaction executor that resides on the same host as the targeted agent. Then the reaction executor executes the corresponding reaction code according to the notification message. A reaction can only be triggered once by a same event.

The reason that we choose the reaction execution model this way is about the performance consideration. From the point view of where to execute a reaction, it can be designed to execute at the tuple space server side or at the client side. For instance MARS, as we see in Chapter 1, its intention is to influence the semantics of the primitives, and naturally the reactions are stored and executed at server side. However, MARS is not suitable to support the reactive behavior in our agent programming model. It is because that usually the functionality of an agent is realized together by the reactive part and the proactive part. In MARS, the reactive part is separated from the proactive part on different nodes. This reduces programmability as these behaviors do not share variables. In addition, it also will take too much resource (for instance threads and CPU cycles) for executing the reactions on server side, and slow down the tuple space server. In JavaSpace, a reaction is executed on the client side through a remote invocation issued from the server side. However, such a solution does not support location transparency. In addition, it also may slow down the tuple space server as the number of remote

invocation increases. Considering our choice, we intend to minimize the workload of event notifications and improve the capability of supporting agent proactive behaviors on the server side. It is because that usually most logic of an agent is realized through the proactive behaviors. Another advantage is that it is also easy to handle the issue of location transparency.

(1) *register(TupleSpaceID tsId, Tuple atemplate, IReactive ref)*: It registers a reaction to tuple space *tsId*. The primitive returns a register identifier *register_id*. At runtime, if a tuple that matches the template *atemplate* becomes available in tuple space *tsId*, the reaction that implements *IReactive* interface is triggered. If more than one agent registers the same template, all registered reactions will be triggered. This primitive enables the reactive parts of an agent to sense stimulus from other agents passively. Meanwhile, the tuple space manager takes over the responsibility of detecting the stimulus.

(2) *register(TupleSpaceID tsId, EventChecker checker, IReactive ref)*: It has the same semantic as the first register primitive, except that the EventChecker *checker* is used to check whether some user-specified event has occurred in the tuple space. The *checker* contains a template and a function implemented by the agent programmer. The template is used to match the event (tuple). The function can be understood as a filter. At runtime, the kernel uses this function to check the matching tuple (event). If this function returns true, it indicates that the event is qualified to trigger the reaction. This primitive enables an agent to program and define a complex trigger condition as a stimulus. The checking process happens at the server site, thus it is helpful to improve the agent performance.

(3) *deregister(AgentRegisterID register_id)*: It deregisters a reaction specified by *register_id*. When an agent deregisters a reaction, the corresponding stimulus will not notify it any more.

2.3 Formal Specification

This section presents a specification of the reactive tuple space based on the state machine model, following the notation used in [Weihl93]. For simplicity we will present the specification of the tuple space formed of the collection of distributed tuple spaces. This means that one can interpret the tuple space id as an extension field (for instance an identity field) of a tuple in the tuple space specified.

The specification involves a set of external actions and a set of internal actions. External actions correspond to tuple space operators, i.e., interface stimulus. Internal actions correspond to kernel (tuple space) actions that are triggered because of kernel state alone without an interface stimulus.

The details of the specification are given in Figure 2.6.

VAR

T: Set[Tuple] := Φ	*set of current tuple stored inside TS*
F: Indexed set [Op] := null	*indexed set of future operations each initialized null*
R: Indexed set [Tuples] := Φ	*indexed set of future responses each initialized to Φ*
E: Set[(Event, Reaction] := Φ	*current set of (event, reaction) in TS*
i : integer :=0	*indexing distinct members of F and R*

External Actions

in(x: Tuple) = $\exists$ y: (y $\in$ T $\wedge$ y=x) |➔ response :={y}; T=T-{y}

read(x: Tuple) = $\exists$ y: (y $\in$ T $\wedge$ y=x) |➔ response :={y};

bulkInWithoutWait(x: Tuple) = response := m(T,x); T := T – m(T,x)
where m(T,x) = T' $\subseteq$ T such that ($\forall$y$\in$T': y = x) $\wedge$ ~ ($\exists$ T'$\subseteq$T*$\subseteq$T $\forall$z$\in$T*:z = x)

bulkReadWithoutWait(x: Tuple) = response := m(T,x);
where m(T,x) = T' $\subseteq$ T such that ($\forall$y$\in$T': y = x) $\wedge$ ~ ($\exists$ T'$\subseteq$T*$\subseteq$T $\forall$z$\in$T*:z = x)

asynIn(x: Tuple) = F(i) := (x,'in'); response:= R(i); i := i +1;

asynRead(x: Tuple) = F(i) := (x,'read'); response := R(i); i := i +1;

bulkAsynIn(x: Tuple) = F(i) := (x,'bulkin'); response:= R(i); i := i +1;

bulkAsynRead(x: Tuple) = F(i) := (x,'bulkread'); response := R(i); i := i +1;

asynIn(lt:LogicTemplate)= F(i) := (lt,'logic_in'); response := R(i); i := i +1;

asynRead(lt:LogicTemplate) = F(i) := (lt,'logic_read'); response := R(i); i := i +1;

out(x:Tuple) = T := T ∪ {x}; response:= {r'| (y,r') ∈ E ^y = x};

register(x: tuple, r:Reaction) = E:= E∪{(x, r)}; *//for simplicity, assume r is distinct //and consider case of event = tuple*

deregister(x: tuple, r: Reaction) = E:= E-{(x,r)};

Internal Actions

asynIn_int = ∃F(i) = (x,'in') ∧ ∃ y∈T ∧ y=x) |➔
R(i) := {y}; T :=T-{y}; F(i) := null ;

asynRead_int = ∃F(i) = (x,'read') ∧ ∃ y∈T ∧ y=x) |➔
R(i):= {y}; F(i) = null ;

bulkAsynIn_int = ∃F(i) = (x, 'bulkin') ∧ ∃y∈T : y=x |➔
R(i) := m(T,x) ; T := T – m(T,x) ; F(i) := null ;
where m(T,x) = T' ⊆ T such that
(∀y∈T': y = x) ∧ ~ (∃ T'⊆T*⊆T ∀z∈T*:z = x)

bulkAsynread_int = ∃F(i) = (x, 'bulkin') ∧ ∃y∈T : y=x |➔
R(i) := m(T,x); F(i) := null ;
where m(T,x) = T' ⊆ T such that
(∀y∈T': y = x) ∧ ~ (∃ T'⊆T*⊆T ∀z∈T*:z = x)

logicIn_int = ∃F(i) = (lt, 'logic_in") ∧ ∃T'⊆ T : lt(T') = true |➔
R(i) := T' ; F(i) := null ; T := T – T' ;

logicRead_int = ∃F(i) = (lt, 'logic_in") ∧ ∃T'⊆ T : lt(T') = true |➔
R(i) := T' ; F(i) := null ;

Figure 2-6 Tuple Space Specification

Referring to Figure 2-6, we will first explain the variables used in the specification. T is the set of tuples currently in the tuple space. F is the set of future (asynchronous) operations yet to be performed. We use an indexed set so that members can be easily referred to distinctly. Op is a data type corresponding to the various asynchronous tuple operators. R is the set of future responses to asynchronous operations. A response consists of a set of tuples. So a tuple operation returns to the accessor a (possibly empty) set of tuples. E is the current set of reactions maintained by the tuple space. Registering a reaction means inserting the (event, reaction) tuple into E. Id is the tuple space identity of a distributed tuple space.

The specification consists of external actions and internal actions. External actions correspond to invocations of access operators by users of the tuple space. Internal actions correspond to internal actions triggered asynchronously in the tuple space due to changes of its state. More details can be explained as follows.

Consider the *in(x)* operation. The specification says that the response will be some tuple y that matches with x. Afterwards, y is deleted from T. *BulkInWithoutWait(x)* returns a maximal set of tuples (m(T,x)) contained in T each of which matches with x. Afterwards, m(T,x) is deleted from T. *AsynIn(x)* assigns the asynchronous operator (x,'in') to the next unassigned element in F, i.e., F(i), and returns with an empty set that can be updated any time (immediately or much later) when *asynIn_int* is performed. When the kernel selects this to be performed successfully then R(i) will be updated with the actual tuple y that matches with x. *AsynRead*, *bulkAsynIn* and *bulkAsynRead* can be understood similarly.

Accesses using logic templates such as *asynIn(lt)* are essentially similar to asynIn except the access condition is specified by a logical template lt, in which case lt(T') is evaluated to true whenever the logic template lt is satisfied by T'. For example, suppose the logic template is ($a \wedge b \vee c$ and there exist tuples x and y such that a = x and b = y then lt({x,y}) = true. A reaction is registered with *register(x, r)* that inserts (x,r) into the set E. Afterwards, whenever *out(x)* occurs, besides x is inserted in T, it also triggers responses corresponding to each reaction r' for which $(y,r') \in E$ and x = y.

The specification given in this section serves to clarify the informal description given in earlier sections, with an aim that the abstraction is not biased with solution details. Hence sets and infinite sets have been used in the specifications. It also takes care of the

atomicity semantics. No distinct of users (accessors) is made. Furthermore, as stated in the beginning, the collection of distributed tuple spaces formed by different groups is treated as a single tuple space to define its consistency semantics represented by the given specification.

Chapter 3 Tuple Space Replication Protocol

3.1 Locality Problem

The tuple space model can be used to support multiple tuple spaces. This allows a programmer to cluster related agents into a group through a tuple space, and scatter different tuple spaces and agent groups onto different nodes. Following such a strategy, concurrency and locality of data access may promote the efficiency of distributed multi-agent systems. However, practically, this may not be easily achieved. For instance agents among different groups cannot be isolated. There is a need for an agent in a group to act as a mediator to communicate with other group members. If different agent groups are located on different nodes, the mediator agent has to talk to other group members through remote tuple spaces. Consequently, the locality of data access has been lost.

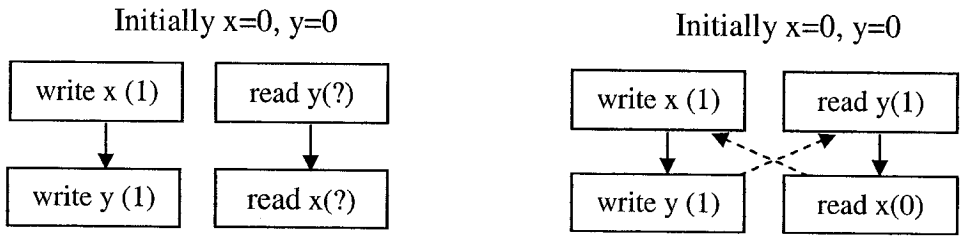
Data replication is a good way to try to keep data local to agents [Shrivastava99], although replication helps primarily in solving availability problem. Two common data replication strategies have existed [Gray96, Kemme00], namely, eager and lazy update. Eager replication is the strategy where an update is propagated (pushed) to all nodes eagerly and usually synchronously. Synchrony incurs extra delay to the producer, which will be blocked until all copies are updated. This approach provides coherence trivially. But it is expensive in both space and time. In contrast, lazy replication only updates the local data, and avoids the synchronization overhead. Instead, updates are provided only when required, which is usually accomplished through a pull operation. However, it still can lead to poor performance as the lazy update slows down the access that triggers the pull operation. Under the lazy update strategy, copy incoherence is obviously possible. Hence the correctness of the protocol implemented lazy replication must be carefully verified before being adopted. When we design a replicated tuple space, it is important to

verify the correctness of the protocol, and analyze its performance to ensure that the protocol is indeed correct and may potentially improve system performance rather than incurring excessive overhead in messaging.

Updates can be performed primary-copy-first or update-everywhere [Wiesmann00]. The primary copy approach requires that all update requests to be performed first at one copy, which is called the primary copy, and then at the other copies. Primary-copy-first update is more likely to cause congestion when the primary copy is the residing at a same server. In contrast, update everywhere strategy does not force an ordering of update at any node. This asynchrony is useful when updates can be broadcasted from a node to all nodes without centralizing it at the home node. It may speed up data access but make coordination among data servers more complex, as the other servers might have made conflicting updates at the same time. The copies on the different servers might not only be stale but inconsistent. Thus a reconciliation protocol has to be designed in order to ensure correctness. However, the process of reconciliation can potentially incur runtime cost. Balancing between the simplicity of primary copy and the complexity of update everywhere, it may be more advantageous to use primary copy strategy for tuple space replication.

Tuple space is a special form of distributed shared memory. Thus the consistency model of distributed shared memory is still suitable for tuple space. There are a number of consistency models for distributed shared memory, for instance release consistency [Gharachorloo90, Keleher92], and causal consistency [Ahamad95] models. In designing the tuple space replication protocol, we require that replication protocol guarantee sequential consistency. Sequential consistency was first formulated by Lamport [Lamport79], which says that the result of any execution is the same as if the memory operations by all processes on the data store were executed in some sequential order and the operations of each individual process appears in this sequence in the order specified

by its program. In other words, the memory operations issued by all processes that access shared memory are interleaved, and the order of the operations is as if the program orders are not violated. For example, Figure 3-1a shows two processes running in a distributed shared memory. If the distributed shared memory is sequential consistent, the execution result (x,y) = (0,1) can never occur. Otherwise such a result violates sequential consistency, because it cannot be obtained by any interleaved execution that does not contradict the program order. Indeed, the result requires the read x(0) to happen before write x(1), and similarly the write y(1) to happen before the read y(1). Thus it leads to a "cycle" representing the contradiction established among the conflict resolution (between read and write of a variable) and program order. It is shown in Figure 3-1b.



a. two processes on DSM

b. execution violation of sequential consistency

Figure 3-1 An Example on Distributed Shared Memory

3.2 Related Work

This section reviews two tuple space implementations. It shows that a replication protocol that works well under a particular system structure such as the S/Net or a tree structure may not be a good solution under a LAN environment. In addition, the process of invalidation can affect system performance significantly.

3.2.1 The S/Net's Linda Kernel

The S/Net [Carriero85] is a multi-computer, which is a collection of not more than sixty-four memory-disjoint computer nodes communicating over a fast, word-parallel

broadcast bus. Hence broadcasting can be easily and efficiently performed via the bus to all nodes. The broadcast latency is independent of the number of nodes, except when too many broadcasts are conflicting on the bus at the same time.

A Linda tuple space was implemented on the S/Net. The tuple space involves replication of tuples. The coherence protocol aims to improve access latency at the expense of communication bandwidth and local storage space. In detail, when a process invokes *out*(*t*) to insert a tuple *t* into tuple space, tuple *t* will be broadcasted to every node in the network. So this is an update-everywhere protocol. When a process invokes a *in*(*s*) to retrieve a tuple that matches template *s*, if there is a matching tuple *t* on its local node, the local node attempts to delete *t* using a deletion protocol. If the attempt succeeds, *t* is returned to the process. If the attempt fails, it means that another process on another node have retrieved *t* successfully. Hence if there is no matching tuple or a deletion attempt fails, all arriving tuples will be checked until a match occurs. The *read* primitive works in the same way as *in* primitive, except that no tuple deletion will be done.

The deletion protocol describes that the node which wants to destructively read a tuple *t* first broadcasts a 'delete a tuple *t*' message to all nodes. Then this node waits for a message from the special node from which *t* is generated. The message will inform it either that "*t* has been assigned to you: proceed" or "*t* has not been assigned to you: wait". In this protocol, the node which generates the tuple is responsible for allowing one process, and only one, to delete it.

The above solution has expensive storage cost. Another protocol that requires less space is proposed. This protocol is based on data partition and migration. An *out* primitive requires only a local install. An *in(s)* causes template *s* to be broadcasted to all nodes if there is no matching tuple locally. Whenever a node receives a template *s*, it checks *s* against all of its locally stored tuples. If there is a match, it sends the matched tuple to the template's node. Otherwise it stores the template for a certain time, and then throws it out.

If the template's original node has not received a matching tuple after the special delay, it rebroadcasts the template. More than one node may respond with a matching tuple to a template-broadcast. When a template-broadcaster receives more than one tuple, it simply installs the extras with its locally generated tuples and sends them onward when they are needed.

The tuple space kernel using the above replication strategy or partition strategy works well under the architecture of S/Net. However, over a LAN environment, operations such as broadcasting to all nodes and waiting for an acknowledgement will take longer time.

3.2.2 A scalable Tuple Space for Structured Parallel Programming

This paper [Corradi95] proposes a tree-structured tuple space implementation. Because of the use of a tree, it is conceivable that the solution is more scaleable. Here the leaf nodes are execution nodes that host the client application. The non-leaf nodes are memory nodes that store the tuple space. Each execution node is connected to a memory node at the next (higher) level. A k-array tree can be used so that each memory node has k other nodes at the level below it.

When a tuple is generated from a leaf that invokes *out(t)*, the tuple will be replicated along the path up to the root. In order to maintain consistency, the runtime system defines an eager up-down protocol to guarantee that any tuple can be retrieved by only one *in* primitive invocation. In detail, when an *in* primitive is invoked at a leaf, it propagates upward towards the tree to find a matching tuple. If there is a matching tuple is found at a node, the protocol will re-direct the propagation down the tree towards the leaf from which the tuple was created. The matching tuple is extracted from every node in this path. If the tuple extraction process successfully proceeds all the way down to the originating leaf, the tuple will be returned to caller process. On the other hand, if the extraction fails in some node, this indicates that it has already been extracted by another process. In that

case, a NOT-OK message is returned to the node where the first match has occurred, and the search will continue until it reach the root and block there.

The tree structure reduces synchronization latency and bandwidth requirements. Indeed a logarithmic complexity can be easily envisaged for the access latency for *in(t)* operations. This is better than a linear complexity if a point-to-point message protocol is used. On the other hand, it avoids using broadcasts. However, the tree-structure may not exactly be matching a LAN environment.

An improvement of the access time can be obtained by restricting that tuples can only be replicated in a given sub-tree. In other words, a tuple have a limited visibility scope. However, it leads to other problems. Under such restriction, a parallel program has to be structured with a predefined access scope. The components of a parallel program have to be designed so that their interactions are restricted in the sub-tree that contains them. Such a restriction makes the program design complex and error-prone. Evidently if the interactions fail to satisfy the sub-tree scope rule, then some program components located at some leaf nodes may not see some tuples that they should see.

3.3 Replication Assumptions

An important assumption in protocol design is the system configuration in which the protocol works. Here we assume a LAN environment. The network environment affects the protocol in the following ways. It is not desirable to replicate a whole tuple space from one node to another node in a LAN environment. However, it maybe reasonable for doing so in a WAN environment since it may take longer time for a remote access.

On the other hand, the software configuration is mainly concerned with the way in which the tuple space server works. Different design of the server may require different replication protocol. Currently, we design the tuple space server with the following assumptions. (i) There is only one tuple space manager on each physical node. (ii) Every

tuple space manager can manage several (logical) tuple spaces. In other words, there can be more than one tuple space on a physical node. A tuple space manager can be viewed as a physical node, and a tuple space can be viewed as a logical node. (iii) A tuple space is not partitioned, and a physical node is the home of its assigned tuple spaces.

3.4 The Tuple Properties

We classify tuples into three distinct categories. The categorization forms the basis for our design in deciding whether a tuple should or should not have replicas in the network. The first category consists of asynchronous tuples which can only be read but not removed by any agent other than the producer agent that has created it. The second category consists of synchronous tuples which can be removed but not read by any agent. The third category consists of hybrid tuples that can be both read or removed by any agent. Obviously hybrid type is the default type, if agent access restrictions to tuples are not known at all.

3.5 Replication Protocol

In our design, we choose not to replicate synchronous tuples. The reason is that if the tuple can only be accessed by a delete (for instance *in*) primitive, then its replica in the system requires a reconciliation protocol that is expensive. But at the same time, without nondestructive read, the justification of replication becomes weaker.

An application designer has more knowledge about the use of tuples in the application. Hence it is possible to leave the responsibility of declaring if some tuples are synchronous to the application designer. In other words, which tuples are only accessed via the *in* primitives is explicitly declared in the code. Alternately, programmer-inserted replication control primitive *subscribeForReplication(tsId, template)* can also be used to

inform the kernel to replicate tuples from tuple space *tsId* at a node where the agent is located. These provide two different mechanisms for programmer-directed tuple replication support in the implementation.

On the other hand, an application designer may choose not to be responsible for tuple replication. In such situation, there is no static knowledge about tuple replications. The replication manager will treat all the tuples as hybrid and use the knowledge about actual agent couplings collected at runtime to decide whether and where tuples should be replicated.

There are various patterns in parallel computing that can be used to model agent coordination. Pipeline pattern means that agents form a pipeline to finish a global task. The animation of the movie Toy Story described in Chapter 1 is an example. Master-slaver pattern contains a master agent that produces data to be processed by slave agents. Each slave processes the data and returns the results to the master. The matrix multiplication described in Chapter 1 is such an example. Cluster pattern involves a set of agents that form a group and interact with others to solve a problem together. The subtasks performed by these agents need coordination. Typically, there is an agent in each group that acts as the coordinator that will collaborate with coordinators of other groups. No matter which pattern application takes, there is always a coordination protocol that governs the progress of these agents. Thus, the coordination protocol becomes the useful information for the replication manager. In this chapter, the presentation assumes that the replication manager already has the runtime knowledge about agent couplings. Chapter 4 will describe how such knowledge can be collected and used.

Usually, a hybrid tuple can be replicated to more than one client site only if agents located on different sites have subscribed for the tuple statically through the *subscribeForReplication*. However, if only based on the runtime knowledge, a hybrid tuple will be chosen to replicate at only one client site. This is for the consideration of

invalidation cost. Certainly, an asynchronous tuple can be replicated at any client site if necessary.

In the following sections, we will describe the client site and server site algorithms. The main functionality of the client site is to maintain the states of the local cache. The main functionalities of the server site are to make decision on replication, and to be responsible for invalidating the replicas. We choose that a replica will be piggybacked with the response to the client site. Compared with the option of pushing the replica directly to a client site, it can save system resource since it does not need to build any extra connection between two sites. In order to make correctness proof clear in next section, we label some statements in the protocol. State machine is also used to illustrate the protocol.

3.5.1 Client Site Algorithm

From the primitive (operation) point of view, the client site algorithm will mainly focus on the *single read* access. The destructively read invocations such as *in* will be directly transferred to the corresponding tuple space manager regardless of the local cache state. The bulk primitives are also not served locally because they refer to the state of the whole tuple space rather than that of the local cache. On the other hand, a logic template primitive actually corresponds to several single tuple primitives and can be handled as the union of the latter.

Notations:

tp : a tuple that the agent wants to insert into tuple space.

tplt : a template that the agent uses to match a tuple.

tsId : ID of the tuple space that will be operated on.

$LocalC_{tsid}$: local cache, which is a set to hold replicated tuples from tuple space tsId.

Rt: a reply returned from home tuple space. It contains two elements. *repli*, which is a set of tuples that needed to be replicated. *o*, which is a generic object. It represents the result

that an agent wants. For instance, if an agent invokes the bulk *read* primitive, o will be a set of tuples.

updateCache(Rt.*repli*): update the states of local cache. It will insert the tuples from *Rt.repli* into the local cache. In kernel level, every tuple has a unique key, and it is easy to distinguish whether one tuple is a duplicate of the other, rather than another tuple with the same attributes.

(1) An agent invokes *out* primitive at client site. An *out* primitive involves sending the tuple to the home tuple space specified by the tsId. The agent is not blocked.

(2) An agent invokes single *in* primitive at client site. An *in* primitive will not access the local cache. Instead, it will access the home tuple space directly. Directly accessing the home tuple space can avoid the expensive reconciliation among different cached clients.

The main routine:

send *in* request to the home tuple space tsId;
receive the matching tuple tp from the home tuple space tsId;
return tp to the agent;

(3) An agent invokes single *read* primitive at client site. It will first look through the local cache, and then the home tuple space. In addition, the response from the home tuple space may contain the piggybacked replicas. If so, it will update the local cache.

The main routine:

//check whether there is a copy in local cache
if ($\exists$ $\theta \in LC_{tsid}$: θ = tplt)
read_commit1: return θ to the agent;
//else, send request to the home tuple spaces
send *read* request to the home tuple space tsId;
read_commit2: receive the response Rt from the home tuple space tsId;
if (tsId is on a remote node)
updateCache (Rt.repli); *// update local cache*
return (Tuple)Rt.o to the agent;

(4) The invalidation process at client site is simple. As it receives an invalidation request from the home tuple space, it will invalidate the replica and reply an acknowledgement.

(5) The local cache can locally decide whether a replica needs to be uncached. In order to save resource, the maximal size of the local cache has to be set. Therefore, the replicas that have not been accessed for a certain time can be uncached. The local cache will send an uncached message to the home site if a replica is hybrid typed.

3.5.2 Server Site Algorithm

Notations:

T_{tsid} : a set to contain tuples referenced by tsId.

N : number of the nodes that the tuple space manager has communicated.

$NeedR_{tsid}[i]$: a set to hold the tuples from tsId that need to be replicated to node *i*.

$HasR_{tsid}[i]$: a set to hold the tuples from tsId that have been replicated onto node *i*.

needRepli(tsId, tp): a function to determine whether the *tp* need to be replicated to some nodes.

(1) A *out* primitive is invoked at server site. In this case, as a tuple is inserted into the tuple space, the home will make a decision on whether the tuple needs to be replicated to some client nodes. If so, the tuple will be recorded, and piggybacked with a response to that client node later on.

The main routine as receive a out request:

```
out_commit:  Ttsid = Ttsid ∪{ tp };
             makeDecisionOnReplication(tp);
```

The makeDecisionOnReplication routine:

```
// make decision on replication, this can be done by the replication manager asynchronously.
if (tp is asynchronous tuple or hybrid tuple)
{       nodes=needRepli (tsId, tp); // which nodes will accept the replica
        for (int i=0, i< nodes.Length; i++)
        {
                // record the tuple, and it will be piggybacked with a response later on
                NeedRtsid[nodes[i]] = NeedRtsid [[nodes[i]] ∪ { tp };
        }
}
```

(2) A *read* primitive is invoked from node *j* at server site. In this case, before a matching tuple is returned to the agent, if there are some tuples that needed to be replicated to the client node where the agent is located, these tuples will be piggybacked with the response to that client node.

The main routine as receive a read request:

```
//search for a matching tuple from tsId, it may block until the matching tuple is found.
θ = Ttsid.read(tplt);
//get the tuples that need to be replicated to node j, and piggyback with the response.
Rt.repli=getPiggybackedReplicas(j);
Rt.o = θ ;
reply Rt to the client;
```

The getPiggbackedReplicas routine:

```
//piggyback the tuples that need to be replicated to node j
tempt[] = NeedRtsid[j];
HasRtsid[j] = HasRtsid[j] ∪ NeedRtsid [j];
NeedRtsid[j] = { }; // set to empty
return tempt;
```

(3) A *in* primitive is invoked from node *j* at server site. In this case, the home site will first send invalidation requests to cached client sites and wait for the invalidation acknowledgements if the matching tuple has been replicated.

The main routine as receive a in request:

```
        θ = Ttsid.in(tplt); //retrieve a matching tuple from tsId. It may block.
        //do the tuple invalidation synchronously.
in_commit: invalidation( θ );
        reply θ to the client;
```

The invalidation routine:

```
if ( θ is hybrid tuple)
{       for (int i=0, i< N; i++)
        {       // there is possibility that the tuple has not been actually replicated to some node.
                if ( θ ∈NeedRtsid[i])
                        NeedRtsid[i]= NeedRtsid[i] - { θ };
                else
```

```
                    if (( θ ∈ HasRtsid[i] )∧ (i!=j))
                    {       // if the tuple has been actually replicated, send a invalidation
                            //message and wait for acknowledgement.
                            send a invalidation message to the node i;
                            wait for acknowledgement;
                    }
            }
    }
```

(5) Other primitives such as bulk *in* or *read* primitives are invoked at server site. For bulk *in*, if a matching tuple has been replicated, it will trigger the invalidation process before returning the matching tuples to the agent.

3.5.3 Illustrate the Replication Protocol with State Machine

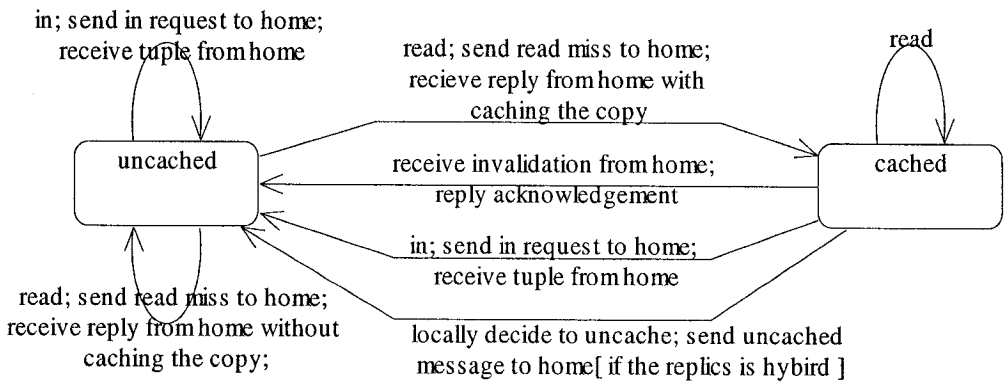


Figure 3-2 State Transition of a Tuple at the Client Site

Figure 3-2 shows the state of a tuple at client side. There are two states for a tuple: cached, and uncached. The cached state means that the tuple is replicated at this client site. The uncached state means that there is no the replica at this site. How to handle read miss is a little complicated. For a read request, if the state is uncached, a read miss message will be sent to the home tuple space. As it receives reply (the response Rt as shown in the protocol) from the home, if the reply indicates the matching tuple can be replicated, the state will transit from uncached into cached. Otherwise, the state is still uncached. This is done by the method *updateCache* in the protocol. The reason that the

copy can be not automatically cached at client site is for the consideration of invalidation cost. For a *in* request, it will be directly sent to the home tuple space no matter the state is cached or uncached. The transition from cached state to uncached state can be caused by an invalidation request from the home or a decision to uncache the replica made by the local cache.

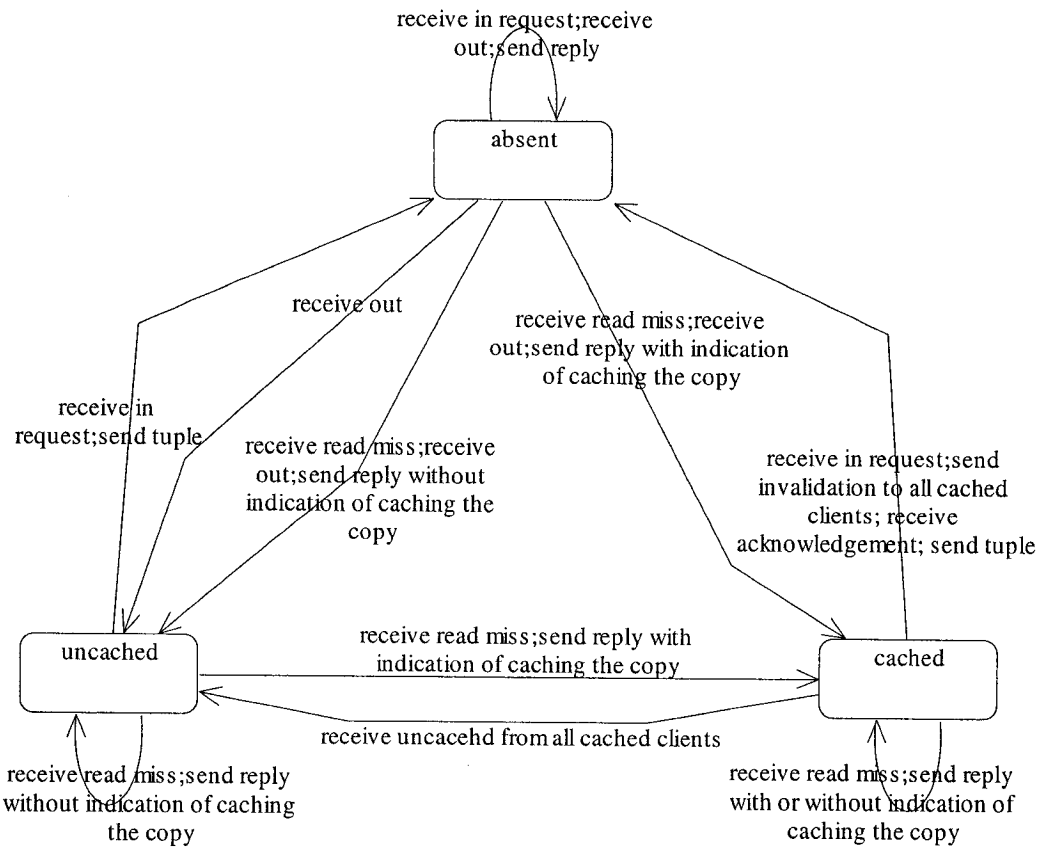


Figure 3-3 State Transition of a Tuple at the Home Site

Figure 3-3 shows the state of a tuple at home site. There are three states: absent, cached, and uncached. The absent state means that there is no the tuple at the home site. The cached state means that the tuple has been replicated to some other client sites. The uncached state means that the tuple has no been replicated. In any one state, the home can receive a *in* request or a read miss message. For a *in* request, especially as the state is cached, the home tuple space will send invalidation requests to all cached clients and wait

for the acknowledgements. The state will transit from the cached to the absent. The number of the cached clients will be only one if the replication decision is based on the runtime knowledge. For a read miss message, it is a little complicated. At any state, as the reply is sent back to the client, the reply will contain the tuples that needs to be cached at the client site (through piggybacking). Therefore, for example, it can transit to uncached or cached state from the absent state by handing the read miss message. Moreover, as the state is cached, a copy can still be replicated to some other site if some other agents have subscribed for the tuple statically.

3.6 Replication Protocol Correctness

In this section, we try to prove that the above replication protocol is correct using the view model [Girard99, Li03]. The view model describes the execution of a system based on distributed shared memory originally. First we borrow the idea to illustrate the execution of a multi-agent system that is based on the reactive tuple space. Then the correctness of the protocol is shown.

3.6.1 The View Model for Tuple Space

The significant characteristic of a tuple is that it cannot be updated. It can only be read or retrieved. In this section, we first introduce some tuple space access notations, then the definition of the global view for tuple space.

An execution of a multi-agent system results in a set of linear traces, one per agent. Each trace contains the sequence of program-ordered tuple space accesses. $out_i(x,v)$ represents the writing of value v into a tuple x by agent i. $read_i(x,v)$ represents the reading of value v from a tuple x by agent i. $in_i(x,v)$ represents the retrieving of value v from a tuple x by agent i. $read_i(X,V)$ represents the bulk *read*, which reads a set of values V from a set of tuple X by agent i. For instance, $read_i(\{x,y\},\{a,b\})$ means that the agent i reads tuples x

and y with corresponding values a and b. $in_i(X,V)$ represents the bulk in, which retrieves a set of value V from a set of tuple X by agent i. For simplicity, we assume that each (x,v) is distinct among the multiple tuple spaces, and omit the parameter tuple space ID. We may also omit the agent label i when the context is clear.

The global view a global view for tuple space is formed of (i) program-ordered tuple accesses, which represents that tuple accesses are invoked in a sequential process. (ii) coupling orders required by the semantic of tuple space. It means that for all *read(x,v)/in(x,v)*, there must have *out(x,v)* → *read(x,v)/in(x,v)*, which we call the directly coupling. In addition, for all *read(x,v)* and *in(x,v)* that occur, there must have been *read(x,v)* →*in(x,v)*, which we call the indirectly coupling. The "→"is the order relation used in the view model. Figure 3-4a shows an execution example, and Figure 3-4b shows its global view.

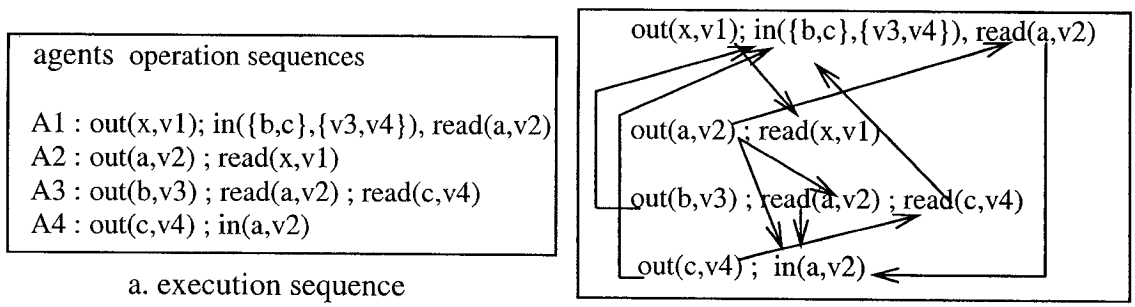


Figure 3-4 The View Model of a Multi-agent Example Based on Tuple Space

Definition of Sequential Consistency An execution is sequential consistent iff its global view is acycle.

The above definition in our model can be reasoned from the original definition of Lamport in Section 3.1. Given an acycle global view, we can obtain an interleaved execution (by iteratively selecting among the subset of operations which are not preceded by other operations in the remaining global view as the next operation) that does not contradict the program order. Thus the execution is sequential consistent. On the other

hand, the reverse is immediate.

Transformation of other Primitives In this section, we will discuss the effects of asynchronous primitives, reactive primitives and logic template primitives on the view model. The purpose of doing transformation is to provide a way to construct the view model of a multi-agent system based on the reactive tuple space simply and easily.

The semantic of an asynchronous primitive is different from that of the synchronous one for agents. However, from the point of view of accessing the tuple space, the kernel makes no difference between asynchronous and synchronous essentially. The only difference is that kernel may assign a system thread to do tuple access for an asynchronous primitive, and borrow the thread of a caller agent to do tuple access for a synchronous primitive. Thus, an asynchronous primitive has the same effect on view model as its corresponding synchronous one. It indicates that when constructing the view model, we can map an asynchronous primitive to its corresponding synchronous one. Figure 3-5 shows a transformation example, which maps a single asynchronous *read* to the single synchronous *read*.

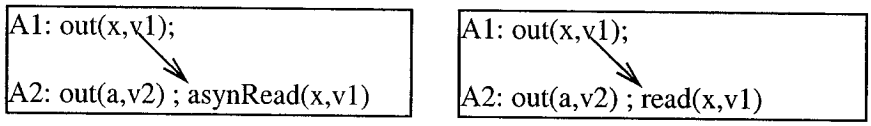


Figure 3-5 Transformation of Asynchronous to Synchronous *read*

As a tuple is written into a tuple space, the kernel will check whether there are some agent reactive behaviors have registered for the tuple. If so, the kernel will have a notification event to trigger the reactive behavior. The notification event includes a copy of the tuple. Thus, in order to model the effect of a reactive behavior in the view, we only need to insert a *read* tuple access event before the ordinary program order of the reactive behavior. For example, suppose agent *A2 out(a,v2)* triggers a reactive behavior notated by $R_1(A1)$ of agent *A1*. Figure 3-6 shows the transformation. Every reactive behavior has

an individual linear trace, here we just use the notation $R_1(A1)$ to represent the trace.

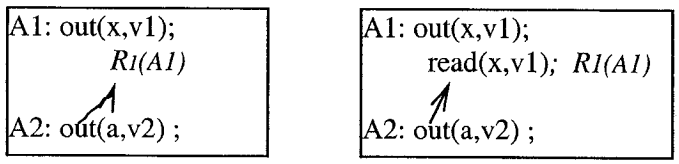


Figure 3-6 Transformation of Reactive Behavior

The semantic of the logic template primitive shows that it is actually made up of several single tuple access primitives. A mediator component will be responsible for collecting the matching tuples before returning the result to the caller agent. Although logic template primitives are in asynchronous form, we could think of them as the synchronous ones when modeling the execution. Therefore in the view model, a logic template primitive can be decomposed into several corresponding single tuple access primitives. For instance, the *logicIn({(x,v1),(y,v2)})* notates a logic template primitive invocation. It means that an agent retrieves a tuple *x* with value *v1* and a tuple *y* with value *v2*. Figure 3-7 shows the transform of a logic template *in* primitive invocation in view model. In other words, we can interpret a *logicIn* to consist of multiple *in* that happen to return those corresponding tuples. In the subsequent proof, for simplicity, we will only refer to events that correspond to the base types, namely *in*, *out*, and *read*, with the understanding that the other accesses can be formed from these types. It is also noteworthy that the reactive parts are factored out from this sequential consistency proof. Interesting enough, reactivity is associated strictly with triggering the reactions when a matching tuple is inserted and the correctness of an implementation protocol can be easily separated from sequential consistency considerations.

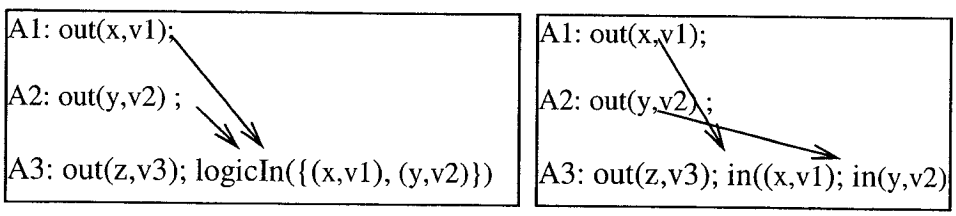


Figure 3-7 Transformation of Logic Template *in*

3.6.2 Correctness Proof

Lemma 1: In the view graph, whenever

(i) out(x,v) → in(x,v)/ read(x,v), then

out(x,v).commit must happen before in(x,v).commit /read(x,v).commit

(ii) read(x,v)→ in(x,v), then

read(x,v).commit must happen before in(x,v).commit

out(x,v).commit means the tuple x has been in the home tuple space, and it corresponds to the label out_commit in the protocol. in(x,v).commit means that invalidation of the tuple x has been done, and it corresponds to the label in_commit. read(x,v).commit means that a copy of the tuple x has been read from local cache or from the home tuple space, and it corresponds to the label read_commit1 or read_commit2 in the protocol.

Proof:

(i) First prove that out(x,v).commit happens before read(x,v).commit. If read(x,v) reads a copy from local cache, as labeled by read_commit1. Since the copy state is cached, obviously out(x,v) has been done, which means the statement labeled by the out_commit has been executed. If read(x,v) receives a copy from the home tuple space, as labeled by read_commite2. It also shows the statement labeled by the out_commit has been done. In detail, from Figure 3-3 state machine at server site, it shows that when the state is absent, the read miss message will not be handled until the tuple is written into the tuple space (receive the out message). If the state is uncached or cached (cached here means the tuple has been replicated to other client, not the one that sends the read miss message), it means that the tuple has already been in the tuple space. Hence, out(x,v).commit happens before read(x,v).commit.

Then prove that out(x,v).commit happens before in(x,v).commit. From Figure 3-3 state machine at server site, it shows that when the state is absent, the *in* request will not be

handled until the tuple is written into the tuple space (receive the out message). If the state is uncached or cached, it means that the tuple has already been in the tuple space. Hence, out(x,v).commit happens before in(x,v).commit.

(ii) This is directly shown through how to handle the *in* requests. From the protocol, when in(x.v).commit has been done (the statement labeled by in_commit has been executed. Specially, when the state of the tuple is absent or uncached, the execution of invalidation labeled by in_commit will invalidate nothing.), there will be no such copy in local cache and home tuple space because of the invalidation. From Figure 3-3 state machine at server site, it also shows that after a *in* request has been handled, the state will transit to absent state no matter which state it was before. Hence, a successful read(x.v).commit labeled by read_commit1 or read_commit2 must be executed before in(x.v).commit is done. Hence read(x,v).commit must happen before in(x,v).commit.

Theorem 1: The view graph under the replication protocol cannot contain a cycle. Hence the replication protocol implements the sequential consistency.

Proof:

Suppose a view cycle exists. Without loss of generality, assume the cycle runs through agent 1, agent 2 ... and agent k. There is a first and last event in each agent, say ei1 and ei2 are the first and last event of agent i in the cycle. Possibly ei1 = ei2. From the protocol, program-ordered tuple accesses are strictly maintained, i.e., ei1.commit happens before ei2.commit (whenever ei1 < > ei2). Furthermore, from Lemma 1, ei2.commit happens before e(i+1)1.commit Hence the view cycle also traces a happens-before cycle. This contradicts the happen-before relation which must be acyclic (causality cannot be cyclic). Hence the view cycle cannot exist. Hence the replication protocol implements the sequential consistency.

Chapter 4 Reactive Tuple Space Design

4.1 Introduction to JADE

A mobile agent platform usually realizes the features of a coordination model introduced in Section 1.1. Generally, a mobile agent is not bound to the system on which it begins execution. In other words, created in one execution environment, it can transport its state and code with it to another execution environment in the network, where it resumes execution. The benefits that mobile agents provide for creating distributed systems include [Lange99] reduction of network load and avoidance of network latency to an agent's progress. However, since mobile agents migrate autonomously in the network, they cannot reliably "know" the locations of their connection peers. Hence, some kind of tracking mechanism must be built into the platform to keep track of agent locations. This mechanism allows agents to communicate without knowing other agents' whereabout. Hence it is more complicated to manage the lifecycle of a mobile agent than that of a stationary agent. Therefore, when studying the impact of a mobile agent platform on the reactive tuple space design, we first need to concentrate on how an agent is modeled and its management (life cycle). For instance, an agent in the Grasshopper platform can have a deactivated state, which means this agent can be deactivated and swapped to the secondary store. The agent state may affect the replication decision. For a deactivated agent, the replication manager may choose not to replicate a tuple that is needed by that agent. Second, the design should address agent mobility. In particular, registration of agent reactive behavior must satisfy location transparency.

In this chapter, we will introduce JADE (Java Agent DEvelopment framework), a FIPA compliant mobile agent platform, based on version 2.61[Rimassa03, JADE], and the design of the reactive tuple space and its integration with JADE. Briefly, JADE can be

described from two different points of view. In the system level, JADE is a platform (also a middleware) which provides runtime supports for application agents. On the other hand, in the application level, JADE defines a framework to facilitate the development of multi-agent systems. It defines an agent model from which an agent developer can extend the abstract model to domain components by following specific business logic.

4.1.1 JADE Architecture

This section gives the description of JADE in the system level. Figure 4-1 shows the overall architecture of JADE platform. A JADE platform is composed of several containers. Every container runs on one Java virtual machine, and can host zero or more agents. The 'host' means that agents, as software components, can only run in the containers. Although a JADE platform can be actually distributed among several networks host, an agent will not observe the existence of the underlying network as JADE provides the high level abstraction called container to hide such complexity.

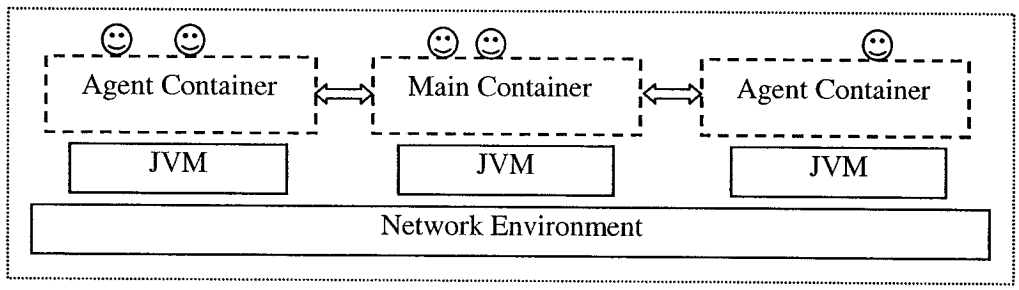


Figure 4-1 JADE Architecture

Container is a core concept in JADE. In fact, it is the container that provides runtime supports for agents. A container is actually a virtual host. Figure 4-2 shows the JADE container internals. In JADE, There are two kinds of interfaces exposed to an agent. The first one is the internal interfaces, which are defined by the container to provide system services to assist the agent implementation. For instance, when two agents need to interact with each other, they only need to call *send* and *receive* primitives to exchange messages, and the container will take care of how to transfer these messages. In brief, the

system service includes the lifecycle management, message service, mobility support, and resource allocation. The lifecycle management service controls the creation and destruction of agents. Mobility service determines how to serialize and de-serialize an agent during its migration. Resource allocation service maintains system resource such as thread pool and message buffer. The message transportation service will be described in detail later on. Considering implementation issues, these services are usually realized by various managers. For instance, intra-platform message service and inter-platform message service are implemented by *MessageManager* and *ACC* (*Agent Communication Channel*) respectively, and the mobility service is done by *MobilityManager*. An agent accesses the system services through its container. The container then delegates these service requests to the corresponding manager.

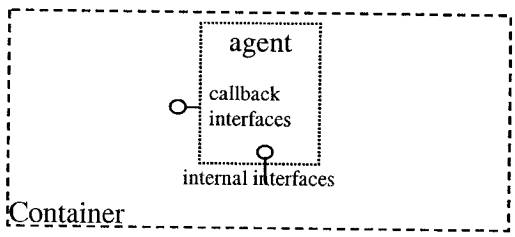


Figure 4-2 JADE Container Internals

The second one is the callback interfaces, which are hooks that will be implemented by an agent developer and called back by the container at run time. The callback interfaces can be further classified into two subcategories: system oriented vs. application oriented. The system oriented interfaces are intended to let an agent customize the system service. For instance, suppose that an agent is required to release all its resource before migrating to another container. Such a requirement can be done by overriding the *beforeMove* method. Then, the container will automatically free the resources allocated for the agent before it moves from the container. Thus, the default service provided by the container can be customized by an individual agent. On the other hand, the application oriented interfaces are intended to help an agent to program its business logic and they are

captured by the *agent behaviors* model in JADE, which will be illustrated in next section. For example, when an agent developer needs to program a buyer agent, what she needs to do is to analyze the business requirement, refine them into different tasks, and implement these tasks by extending the *behaviors* interfaces. Here, tasks can be understood as the functionalities that need to be done in order to meet its goal. Tasks have the same meaning as the so-called behaviors in a typical programming model. Thus, the development of a multi-agent system is simplified as ordinary object oriented design. However, an agent has no external interfaces exposed to outsides, since agents only know each other by identifier and the only way to coordination is through message passing.

There are two kinds of containers: main container and agent container. A platform can have several agent containers but only one main container. From the point of view of an agent, both containers provide the same services described above. However, in system level, the main container represents the platform and it has a global view of the system. For instance, the main container maintains a *Global Agent Descriptor Table (GADT)*, which maps every agent to the remote reference of the container where the agent lives. Therefore, the main container knows exactly the location of every individual agent. In contrast, an agent container has a *Local Agent Descriptor Table (LADT)*, which only records the information of its local agents and has no idea of location of the other agents. Therefore, the main container actually acts as a centralized mediator. When an agent migrates from one container to another container, or when an agent is destroyed in one agent container, such an event has to be notified to the main container so that it can refresh the *GADT*. When an agent sends a message to another agent located in a different agent container at first time, it has to contact the main container to get the proxy for the agent container where the receiver agent locates. Then it communicates with that container directly through the proxy.

4.1.2 JADE Agent Model

The purpose of introducing the JADE agent model is not to show how they abstract an agent, but to explain why we do not choose to build our reactive tuple space server in the agent level but in the system level.

Briefly, an agent is an active object and adopts a *thread-per-agent* concurrency model in JADE. This means that there is a single Java thread (the embedded scheduler) for an agent to execute all its tasks concurrently. An agent usually needs to do several tasks at the same time in order to fulfill its goal. For instance a seller agent negotiates with several buyer agents simultaneously. By letting an agent only have a thread, JADE tries to minimize the consumption of the system resource. This is in contrast to a thread-per-task model, where each task of an agent will be executed by a separate thread.

In order to realize such concurrency model, JADE provides an *agent behaviors* model. Statically, a behavior is an abstract class which exposes an interface called *action*. The action is what that is needed to meet the particular requirement as a task. An agent naturally can have multiple behaviors to model tasks. It can be understood that this behavior has the same meaning as the proactive behavior in our agent programming model. At runtime, the embedded scheduler inside the agent will take the various behaviors objects available and execute them in round-robin. However, the scheduler cannot save the stack frame for a behavior object. This means that once a behavior object is executed, it will not yield its control of the scheduler to another behavior object until it returns from its execution. In other words, if a behavior object is blocked or is in an endless loop during execution, all the other behaviors of the agent will never be scheduled. In addition, the behavior model and its round-robin scheduling policy may be unfair to individual behaviors. For instance, if one behavior is complex for performing long operations and another one is simple, then the complex one will be given more time.

The requirement of the reactive tuple space server is complicated. For example, it needs to synchronize multiple accesses to a single tuple space. If an agent is responsible for handling all the access requests, then an access process becomes sequential from the beginning to the end. This will eliminate the opportunity of segmenting the tuple space and having concurrent accesses to a single tuple space. On the other hand, one may think that we can have multiple agents take care of the accesses to a tuple space. However, it is impossible to decide the number of agents statically before knowing the workload. If the number changes dynamically, the cost of creating and destroying an agent is higher since an agent is more weight than a Java thread and also the main container must be notified of the agent creation and destruction events. The other requirements such as supporting agent reactive behavior and replicating tuples to some potential containers face similar problems. Therefore, in order to avoid the unfair scheduling and make the system efficient, we would not consider building the reactive tuple space server in agent level.

4.1.3 JADE Message Passing

Message passing is one of the core services that JADE provides to support agent interactions. Since the purpose of having reactive tuple space is to provide a dynamic sharing media for agents, it is necessary to have a good understanding of message passing internal.

JADE uses FIPA ACL specification to create messages. A FIPA ACL message contains a set of message elements, which mainly include *communicative act*, *sender*, *receiver*, and *content*. The communicative act element denotes the type of a message. FIPA already have defined some standard acts, for example the *query-if act* allows an agent to ask the receiver agent whether a proposition given in the content is true. The sender and receiver elements denote the identity of the sender and receiver agents. The identity is the agent name in JADE. The receiver may be a single agent name, or a sequence of agent names

in case of message multicast. The content element denotes the content of a message, which may involve specific ontology.

JADE has different message delivery strategies to deal with message passing between intra-platform and inter-platform. For intra-platform, there are two cases. If the agents that need to communicate live within the same container, JADE uses Java local call to exchange a message. On the other hand, if these agents live in different container, JADE uses Java remote call RMI to deliver a message. Massage encoding and decoding are just Java remote object serialization and de-serialization. Particularly at the first time that they communicate, the sender container has to first contact the main container to create a proxy of the container where the receiver agent lives. Then the sender container will cache the remote reference for later communication. In contrast to the inter-platform case, if an agent in a JADE platform wants to interact with another agent on another FIPA compliant platform, the protocols will be based on IIOP or HTTP. Message encoding and decoding is different from that of intra-platform. For instance messages can be transferred between platforms in XML form (for instance, XML encoding over HTTP). Messages for agents located in different platforms are all through the main container unless it is configured to let another container be the router.

In JADE, every agent has a message queue, which buffers the messages received from sender agents. JADE has so-called asynchronous messaging using push-to-receiver dispatch model. This means that a sender agent can send a message to a receiver agent any time as long as it knows the name of the receiver. The kernel will push the message into the message queue of the receiver agent. A receiver agent can try to receive a message at any time. The kernel will check the agent's message queue to see whether there is an available message. The advantage of providing every agent with its own message queue is to reduce the synchronization among agents in receiving messages, although it still includes synchronization between pushing a message into the queue and

pulling a message from the queue. Although every agent has its own queue to receive messages, there is only a shared queue in a container among agents for sending messages. This would potentially slow down message transportation.

4.2 Reactive Tuple Space Design

4.2.1 Design Principle

Efficiency is the primary non-functional requirement (or say quality attribute) when designing the reactive tuple space server. The ultimate goal is that the performance of the reactive tuple space would be competitive with that of message passing mechanism in JADE. However, the natural characteristic of the tuple space determines that it is not easy to achieve the goal. Since tuple space is time and space uncoupled, the system cannot predict which agents will consume a tuple dynamically although guessing is allowed. Hence, a tuple space cannot be partitioned into such granularities that every agent has a place to hold its own tuples as that of every agent having a message queue in JADE. Consequently, the synchronization among tuple space accesses would be more expensive than JADE message passing. For instance, it excludes concurrent executions to some extent. As one agent accesses the tuple space, some others may have to wait. In addition, it may take longer for searching a matching tuple as the number of tuples increases. Therefore, from functional analysis to system design, we shall take into account the performance requirement at every step. For example, at analysis stage we conclude that multiple tuple spaces are necessary in order to support agent clustering. In designing a single tuple space, it is desirable to partition it into small fractions to increase the concurrency among agents.

Another concern is the maintainability (modifiability). We need to decompose the whole system into several components (modules) to encapsulate volatile implementation details

behind stable interfaces. Thus it can localize the impact of design and implementation changes, and reduce the effort to understand and maintain existing software. It reflects so-called design for change. For instance, by having a client component (module) to provide agents stable interfaces in the form of primitives to access tuple space, then the change of the tuple space structure and its algorithms will not affect application. Design patterns [Carey00, Gamma94, Hofmeister99, Johnson98, Lavender95, Schmidt02] would be useful for the system evolution.

However, maintainability and performance requirement may conflict. The tuple space shell component (module) is an example, and the detail will be illustrated in Section 4.2.3.1. When a conflict occurs, maintainability is preferred because we believe in the philosophy that a good design should not bring bad performance. In addition, there usually are just several parts, for instance how to structure the tuple space and access it, that influence system performance significantly in a system. It is that 20% of the code that does 80% of the work. We therefore would like to recognize the important parts and optimize them during design stage without affecting the overall architecture. However, we may miss something. If we are not satisfied with the performance after testing, we can try to find what we miss with the help of tools such as *Java Profiler* and do optimization again. This strategy indeed helped us to improve system performance. For instance, we successfully optimized the message structures of transferring requests and responses among containers. It will be shown in Section 4.2.3.2.2 when we illustrate the design of the asynchronous tuple space access.

4.2.2 Reactive Tuple Space Architecture

Figure 4-3 is the reactive tuple space architecture in high level abstraction. It shows that the reactive tuple space is designed to be part of the services (infrastructures) provided by a container. The concrete modules that realize the abstractions will be illustrated in next

section. There are two layers in this abstract architecture: the tuple space shell and the tuple space kernel. Multiple tuple spaces are distributed over the network environment. The communication between the shell and the kernel is a local method call. The kernels located in different containers communicate with each other through RMI. The shell is in the agent level and supplies stable interfaces for agents. These interfaces reflect the proposed tuple space primitives. The shell actually separates concerns about the interfaces from their implementations in the kernel level. In addition, there is a connector in the shell. The connector takes charge of the exchange and transformation of data between the agent level and the kernel level. For example, it appends the agent name to the tuple space access request (primitive invocation) so that the server component can recognize which agent has issued the request. This information is useful when considering replication.

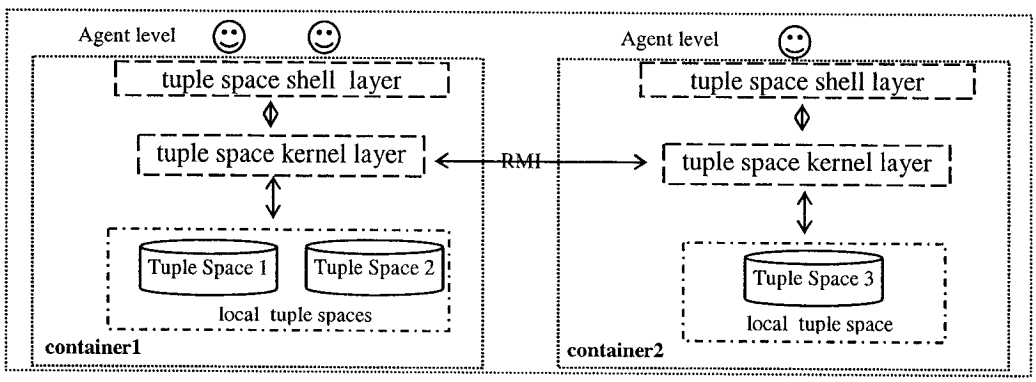


Figure 4-3 Reactive Tuple Space Architecture

On the other hand, the kernel is responsible for receiving requests from the shell, accessing local or remote tuple spaces synchronously or asynchronously, and then sending the responses back. If the tuple space is located on a remote container, the kernel will delegate the request to the remote kernel where the tuple space is located. The kernel also has a tiny framework to support agent reactive behaviors. The framework provides the reactive programmable interface called *IReactive* for agents and will trigger (execute)

these reactive behaviors as the specific events show up. In addition, the kernel also uses replication to make the system efficient.

4.2.3 Reactive Tuple Space Detail Design

In the following sections, we will explain the reactive tuple space design in detail. Figure 4-4 shows the overall picture of the modules [Booch99]. Modules are defined as packages in Java. The modules are organized hierarchically and loosely coupled. Objects insides a module are highly cohesive. However, the reason that there exists bidirectional coupling between the *TupleSpaceShell* module and *TupleSpaceReaction* module is that the agent reactive behaviors have to be notified. The *TupleSpaceShell* module servers as the shell in the architecture diagram, and the other modules belong to the kernel.

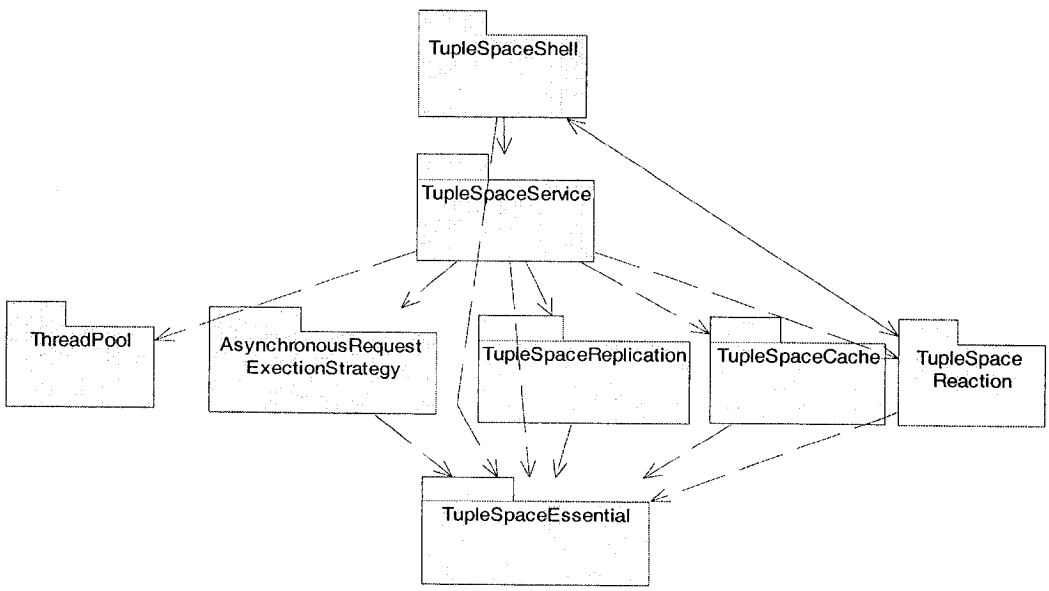


Figure 4-4 Reactive Tuple Space Modules

As we introduce the detail design, we will not describe these modules in an enumerative style. Instead, we focus on (i) the tuple space shell, (ii) the core components such as the reaction part defined by the kernel, and (iii) what we did in the kernel to try to make the system efficient. The runtime description then will be given. Because of the limitation of space, we may not show all the classes for every module. For a class, we may just list

part of the operations and attributes to show the design and its rationales. In addition, we do not show all the dependency relationship among classes. This should not affect our description. If there is a name conflict, we will further identify the class name with its package path.

4.2.3.1 Tuple Space Shell

As we see from Section 4.2.2, the tuple space shell is a lightweight layer that is integrated with JADE agent framework. It defines the entry for an agent to access tuple spaces. The tuple space shell is encapsulated in jade.core.ts.adapter package, and its class diagram is shown in Figure 4-5.

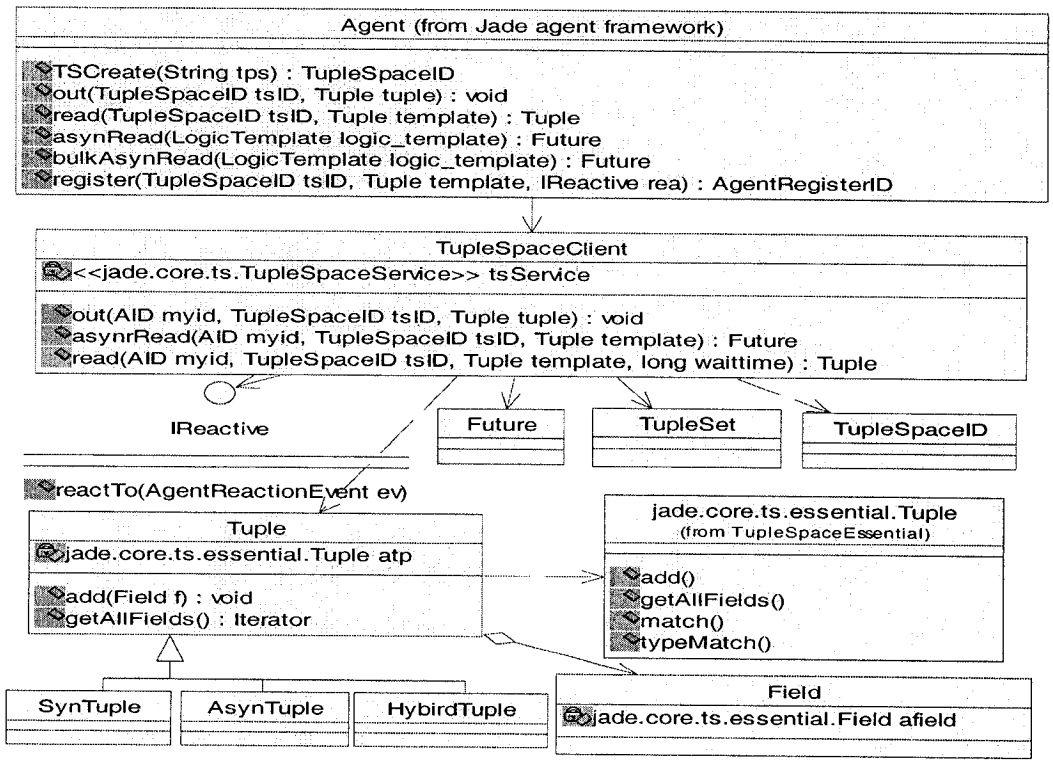


Figure 4-5 TupleSpaceShell Module Class Diagram

When we design the tuple space shell, the top priority is the separation of concerns between the interfaces and the implementations. In addition, the interfaces shall be clear and easy for the agent programmers to use.

From Figure 4-5, it shows that the tuple space primitives are embedded into the *Agent* class. The *Agent* class is the core of JADE agent framework. The details of the primitives are presented in Chapter 2. These primitives coexist with message passing. An agent can use both to interact with each other. For an agent programmer, learning how to use these primitives is as simple as learning how to use message passing.

There are many classes such as *Tuple* and *Future* assisting the primitives. This agent level *Tuple* class exposes the interfaces to an agent programmer. For instance, he/she can add fields into a tuple with *add(Field f)* method, iterate all the fields with *getAllFields()*. There is another kernel level *Tuple* defined in the *TupleSpaceEssential* module. The reason that we have two different *Tuple* classes is that it needs more methods to support system operations in the kernel level than that of the agent level. For example there is a *typeMatch*() method in kernel level *Tuple* class. It is used to decide whether two tuple templates are matched. This function will be used when building the index for tuples. If we only have a kernel level *Tuple* class, an agent programmer will be confused about how to use it. Therefore, in order to provide clear interfaces, we define the agent level *Tuple* class to hide the complexity of the kernel level *Tuple* class. An agent level *Tuple* object holds a reference to a kernel level *Tuple* object, and the real operations are actually on the kernel level object. Such design conflicts with the efficiency requirement since more object creations and destructions are involved. However, it will not lose performance significantly. Balancing the two considerations, we prefer to have a clear design. Take the *Future* class as another example. The agent level *Future* class has no set methods, and only the kernel level *Future* class has. The *Future* class is a placeholder for the result of an asynchronous primitive invocation. Such design only allows the kernel to assign a value to a *Future* object and prevents an agent programmer from doing such operations at the agent level. The other classes such as *Field*, *TupleSet*, *MultipleTupleSet*, *AgentRegisterID, AgentReactionEvent, etc* in this package are all based on this rationale.

In addition, there is an interface *Ireactive* supporting the agent reactive behavior through the tuple space. It provides only one method *reactTo(AgentReactionEvent ev)*. An agent programmer can program a reactive behavior by just implementing this method. In fact, the *Ireactive* is the hotspot (or say hook, or extension item) of the agent reactive framework, which will be illustrated in Section 4.2.3.2.3.

The transformations of data between the agent level and the kernel level are taken care by the *TupleSpaceClient* object. It works as a connector between the two different levels. In general, it will delegate the tuple access requests to the kernel *TupleSpaceService* object. As the result is returned from the kernel, it will transform the result to the corresponding agent level data. There is only one *TupleSpaceClient* instance in one container, and thus the singleton pattern is used.

4.2.3.2 Tuple Space Kernel

Figure 4-6 shows how the core components of tuple space kernel are statically structured. It is in the jade.core.ts package. We would like the kernel to have only a single point to expose its services for the shell. The *TupleSpaceService* is for this purpose. It is composed of several managers (usually active objcets) which will take different responsibilities in serving different requests. The *TupleSpaceService* actually works as a mediator. As it receives a request from the shell, it will delegate the request to the corresponding component after analysis. For instance, a reaction registration or deregistration will be handled by the *ReactionManager*. Asynchronous access requests will be handled by *AsynRequestManager* and *AsynResponseManager*. Replication issues will be handled by the *TupleSpaceReplicationManager*.

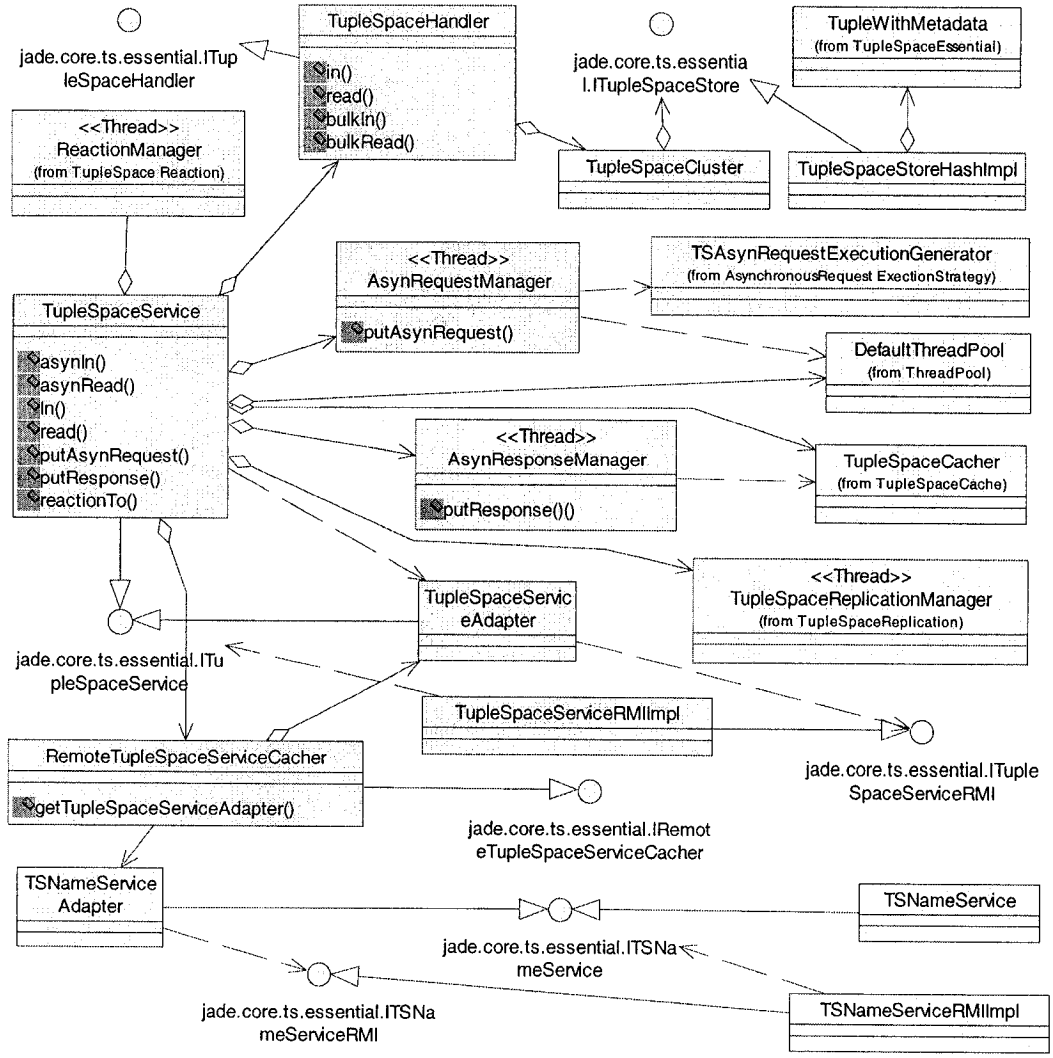


Figure 4-6 TupleSpaceService Module Class Diagram

In the following sections, we will describe three important design decisions: (i) the data structure of a tuple space, (ii) the asynchronous tuple space access, and (iii) the reactive behavior framework.

4.2.3.2.1 Tuple Space Data Structure

Since the tuple space has to be synchronized, we want to reduce the cost of the synchronizations among accesses with an efficient data structure shown in Figure 4-7. A

tuple space is organized as clusters of segments. The tuples that have the same number of fields are clustered together. A segment is then organized with a hash function on the first field of a tuple. The default hash function is simple. It first serializes the value of the first field to a string, and then gets the hash code of the string. However, since the hash function is based on the first field, the efficiency of the data structure depends on how to define and use a tuple. If most tuples can be distinguished by their first fields, it would be very efficient. Such data structure not only can reduce the cost of synchronization when doing an access, but also can promote concurrency among different segments. The *TupleSpaceHandler, TupleSpaceCluster* and *TupleSpaceStoreHashImp* shown in the Figure 4-6 together realize this data structure.

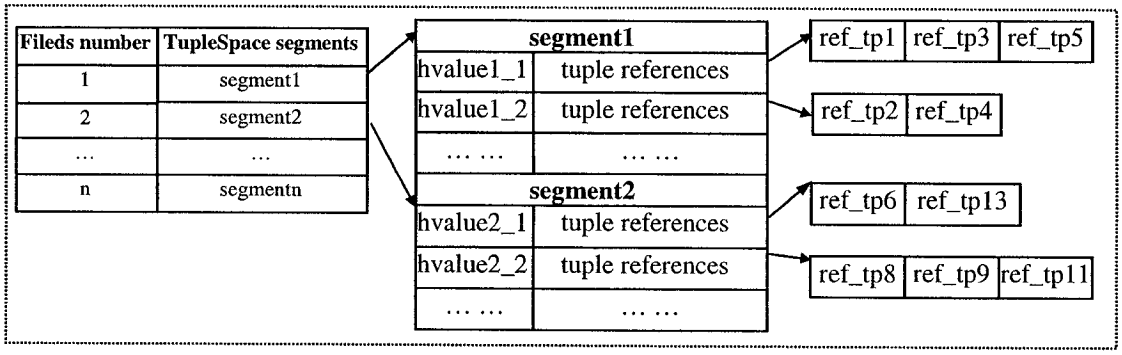


Figure 4-7 The Date Structure of a Tuple Space

4.2.3.2.2 Asynchronous Tuple Space Access

The asynchronous tuple space access is an important feature to improve the application performance. As an asynchronous primitive is invoked, the kernel will create a future object and return it to the agent. As we make design decision, it is preferable that the future object is returned as earlier as possible. The agent can continue its computation. Meanwhile, the kernel will have a system thread do the tuple space access. In order to optimize the system performance, we design that the kernel uses different ways to handle the asynchronous requests from local agents and remote agents. We distinguish these requests as the local asynchronous requests and the remote asynchronous requests.

Before showing the two different ways, we first need to describe how a system thread can do a tuple space access dynamically. Basically, the thread needs to know the entry of the method of doing a tuple space access. Section 4.2.3.3.4 will show that the entry actually is a programmed java *Runnable* interface. In Figure 4-8, *TSAsynRequestExecutor* implements the *Runnable* interface. It provides an abstract method *executionStrategy()* using template method design pattern. The abstract method then is implemented by various subclasses to define the algorithms (strategies) of doing various asynchronous tuple space accesses. These algorithms reflect how to handle the local and remote requests. It also uses strategy design pattern. The *TSRequest* actually provide the information that the corresponding algorithm will need. These algorithms server as the future operations specified in the formal specification in Section 2.3.

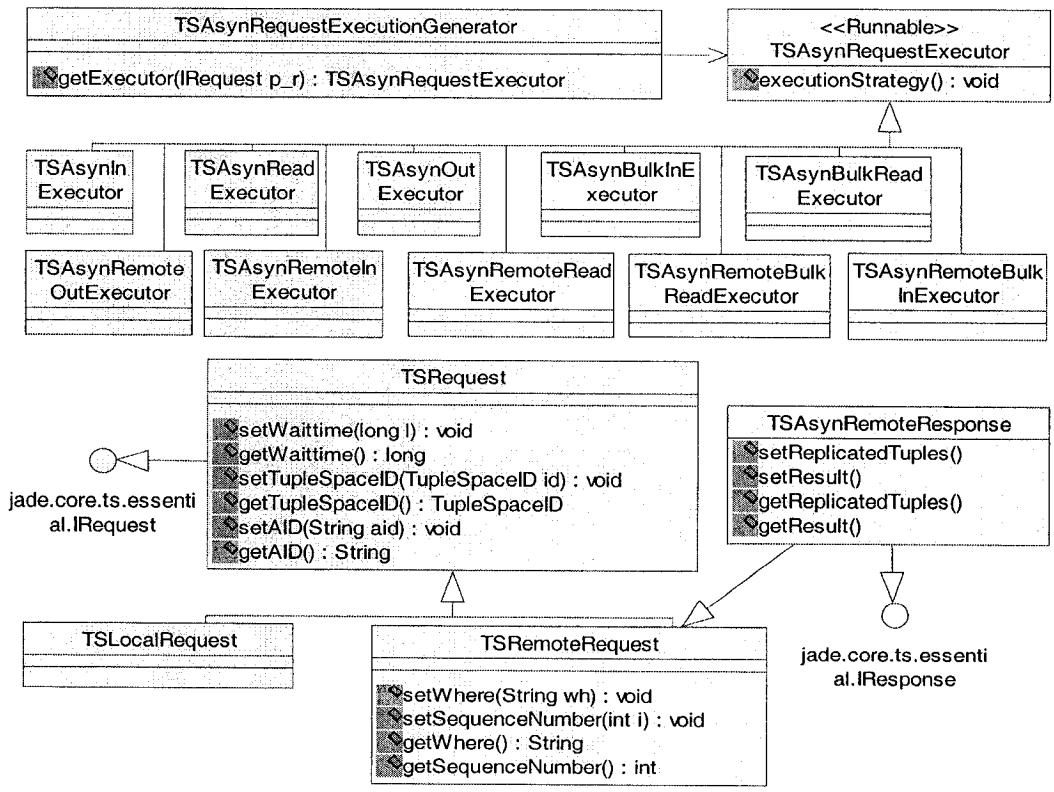


Figure 4-8 AsynchronousRequestExecutionStrategy Module Class Diagram

It is a little complicated to handle a remote asynchronous request since the future object and the tuple space accessed are in the different containers. The future object cannot be assigned a value directly. We design as follows to solve the problem. Two active objects *AsynRequestManager* and *AsynResponseManager* shown in Figure 4-6 will execute a remote asynchronous request coordinately. Specially, the remote asynchronous request will be put into a request queue of the container where the targeted tuple space is located. As shown in Figure 4-8, the remote request contains the information such as where the request is from, and a sequence number representing the individual request. The *AsynRequestManager* is responsible for generating a system thread with a proper execution algorithm according to the type of the request retrieved from the queue. With the information provided by the request, the algorithm knows where to return the response. The response contains the result and the request sequence number. The *AsynResponseManager* located in the requester container will be responsible for handling the response. According to the sequence number that the response provides, it will assign the result to the corresponding future object. The dynamic behavior of a remote asynchronous tuple space access will be illustrated in detail later.

In contrast, the characteristic of a local asynchronous request is that the future object and the tuple space accessed are in the same container (the same virtual machine). Thus, the value (for instance a matching tuple) can be directly assigned to the future object in the algorithm. In order to do so, the local request needs to contain the reference of the future object. Therefore, it does not require the *AsynResponseManager* to be involved.

In Figure 4-8, it shall be noted that a request contain two fields to show who (which agent) issues the request and from where (which container). At the beginning, we designed the 'who' as the agent identity (*AID*) and 'where' as the container identity (*ContainerID*). Both *AID* and *ContainerID* are from JADE, and have complex structures. As we trace the execution with *Java Profiler* tool during the performance test, it shows that the remote

calls take too much time because of object serialization and de-serialization. Therefore, we optimize these by modifying the type of 'who' and the type of 'where' to *String*, the performance is improved by around 25% since it reduces the cost of object serialization and de-serialization.

4.2.3.2.3 Reactive Behavior Framework

We have already shown the *IReactive* programming interface in Section 4.2.3.1. In this section we will illustrate the design of the reactive framework. First we will describe how to represent a reactive behavior and a reaction event. Then we will show how to detect an event and execute a reactive behavior. Figure 4-9 shows the class diagram of the framework.

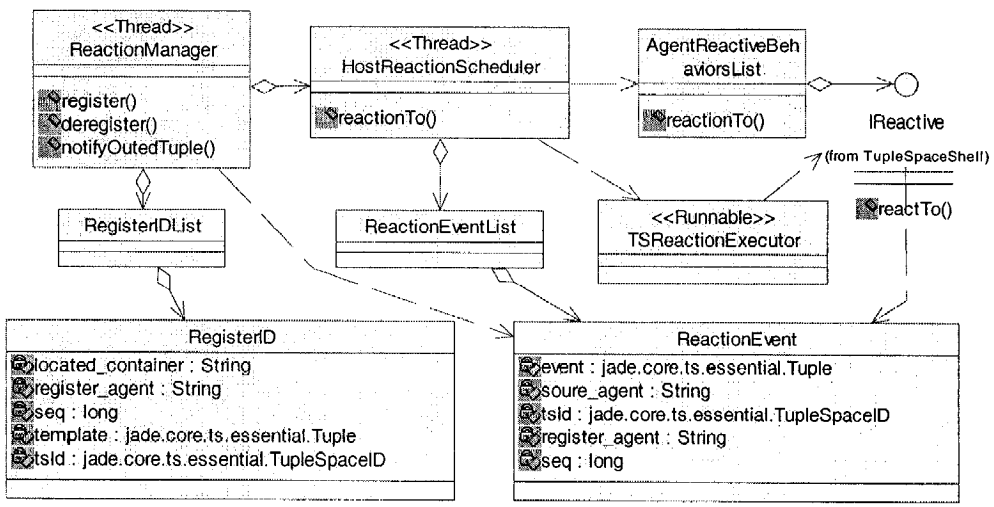


Figure 4-9 TupleSpaceReaction Module Class Diagram

A reactive behavior (registration information) is represented by a *RegisterID* object. The *RegisterID* contains information such as (i) the container name, (ii) the agent name, (iii) reaction sequence number, (iv) tuple template, and (v) tuple space ID. Such information can locate the exact reactive behavior. In brief, the container name identifies the *HostReactionScheduler*. Then the agent name locates the *AgentReactiveBehaviorsList*.

Then the sequence number locates the reactive behavior. In addition, the tuple template and tuple space ID together define the interested event (tuple).

The *ReacionEvent* object represents a reaction event. It contains the event (tuple) and the name of the agent that generates the event. It also has information from the *RegisterID* such as the agent name that registers for the event and reaction sequence number. Such information identifies which reaction needs to be notified.

The detection of an event is done by the active object *ReactionManager*. There is only one *ReactionManager* instance in a container. It keeps records of all the agent reaction registration information. The information will be stored in a list *RegisterIDList* that is indexed by tuple space ID and tuple template. For every newly inserted tuple, it will check whether some reaction behavior has registered for it. If so, it will create a reaction event *ReactionEvent*, and send it to the queue of the *HostReactionScheduler* identified by the container name form the *RegisterID*.

The invocation of a reactive behavior is triggered at the node where the agent is located. The advantage is to reduce the tuple space server side workload. In detail, the active object *HostReactionScheduler* takes the responsibility. When the *HostReactionScheduler* receives a *ReactionEvent* from its queue, it will find the corresponding reactive behavior. Then it generates a system thread to execute the reactive behavior. At this point, the kernel level *ReactionEvent* will be transformed into the shell level *AgentReactionEvent* object. Different reactions are executed in different thread contexts.

In addition, the *Agent* class is modified to have an attribute *AgentReactiveBehaviorsList*. It is responsible for managing all the reactive behaviors of an agent. The advantage of letting every agent have an *AgentReactiveBehaviorsList* is that it would be easy to handle agent migration. Before migration, an agent has to send a migration message to the *ReactionManager* and *HostReactionScheduler*, and then wait for the acknowledgements from the both. Thus, the *ReactionManager* can update its registration information and

will notify new events to the new destination container. The *HostReactionScheduler* will have an opportunity to finish all the reaction notifications for the agent.

4.2.3.3 Performance Consideration

In the previous section, we considered the performance issues while making design decisions. In this section, we still show some other efforts to try to make the kernel efficient. They are (i) tuple space replication, (ii) tuple space cache, (iii) caching the remote tuple space services, and (iv) the thread pool. The former two together reflect the design of the replication protocol.

4.2.3.3.1 Tuple Space Replication

In Chapter 3, we have shown the replication protocol. In this section, we will show the important design features for tuple space replication at the server side. Figure 4-10 shows the class diagram. First we will introduce the runtime and static knowledge that the replication manager needs in order to make decision on whether and where to replicate a tuple. Then describe the replication manager.

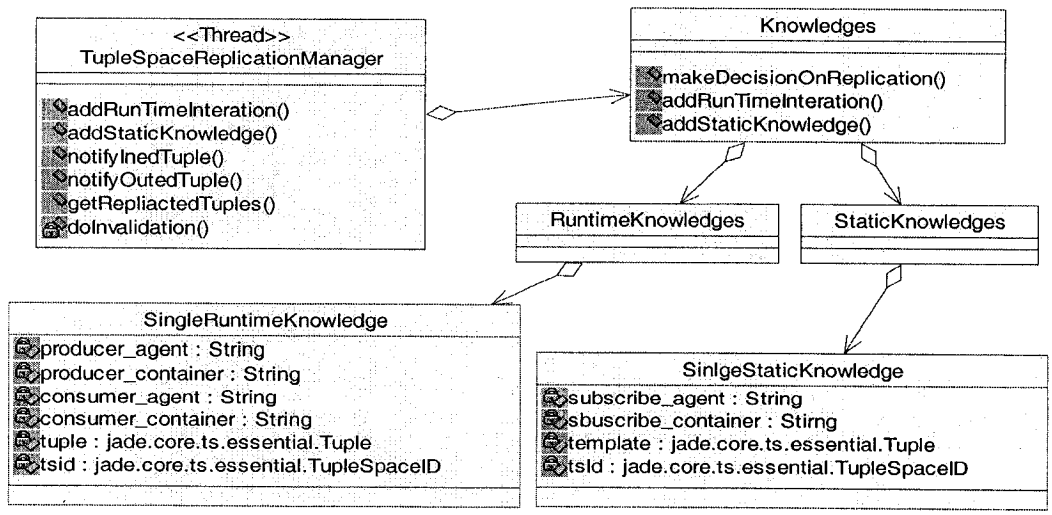


Figure 4-10 TupleSpaceReplication Module Class Diagram

The runtime knowledge is to track the interaction protocol between two agents. One step of the interactions between two agents is defined by a *SingleRuntimeKnowledge*. It contains information such as (i) producer name, (ii) producer container name, (iii) consumer name, (iv) consumer container name, (v) access operator used, (vi) involved tuple, and (vii) tuple space ID. All instances of the *SingleRuntimeKnowledge*s are contained in the *RuntimeKnowledges*. A search of the *RuntimeKnowledges* is optimized by maintaining a pointer that identifies the *SingleRuntimeKnowledge* that contains the most recent consumer. The static knowledge is constructed with the help of agents through *subscribeForReplication*() invocations. It is defined similarly as that of the runtime knowledge.

The *Knowledges* provides functionalities to decide whether a tuple needs to be replicated at other containers. The decision algorithm is simple: the priority is based on the static knowledge. If it shows that the some containers have subscribed the tuple, then the tuple will be replicated to these containers without considering runtime knowledge. However, if the *Knowledges* cannot decide with its static knowledge, it will check its runtime knowledge to find a container that has consumed a similar tuple most recently to accept the replica. We say that a tuple *A* is similar to a tuple *B* when every field of the two tuple matches or has the same type. The reason that we only choose a container to have the replica is to balance between the cost of invalidation and the return of local access.

The *TupleSpaceRepliactionManager* is an active object. When a tuple is inserted into a tuple space, it will decide based on its knowledge (through the *Knowledges* object) whether this tuple needs to be replicated to other containers. If so, it will keep a record of such a replication. These replications will be piggybacked with the responses to the same potential container.

Consider the invalidation process. Before a *HybridTuple* tuple is returned to the agent that issues a destructive read invocation, the *TupleSpaceRepliactionManager* needs to

invalidate all the replicas if the tuple has been replicated at other containers through the method *doInvalidation*().

4.2.3.3.2 Tuple Space Cache

In this section, we will show the design of tuple space replication at the client side. Figure 4-11 shows the class diagram. We decide that a container only has a cache to accept replicas from the other containers. The advantage is that the replicas not only can be managed easily, but also can be shared among agents. First we will introduce the entry of cache service. Then give brief description of the data structure of the cache.

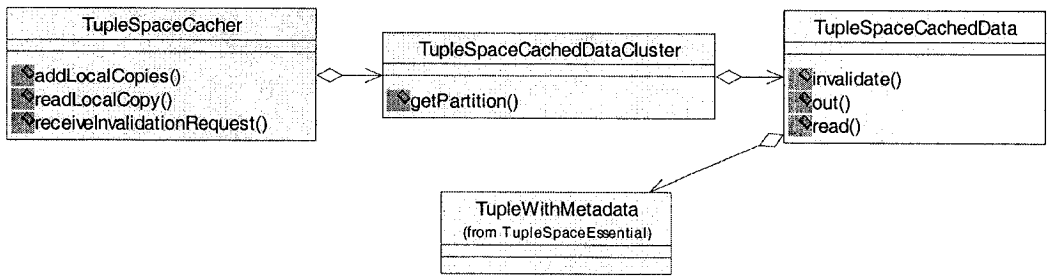


Figure 4-11 TupleSpaceCache Module Class Diagram

The *TupleSpaceCacher* provides the entry to access the cache. The cache can be queried, updated (adding replicas) and invalidated. The query and update are straightforward from the protocol described in the Chapter 3. For instance, when an agent issues an asynchronous read on a remote tuple space, the kernel will first access *TupleSpaceCacher* to check whether there is a matching local copy through the *readLocalCopy*().

The data structure of cache is organized as follows. The replicas are first classified by their tuple space ID. Then, the replicas that have the same number of fields are clustered together in a segment. Based on the assumption that the number of replicas from a tuple space is of limited size, we do not further partition the segments. The *TupleSpaceCachedDataCluster* and *TupleSpaceCachedData* together realize the structure.

4.2.3.3.3 Caching Remote Tuple Space Services

We already know from Section 4.2.3.2 that the *TupleSpaceService* component is the only entry to access the kernel service. Agents create and access a tuple space by name. The location of a tuple space is transparent to agents. The distribution of multiple tuple spaces is actually managed by the centralized *TSNameService* located in the main container. Without caching, every time when a kernel accesses a remote tuple space, it has to first get the stub of the corresponding remote *TupleSpaceService* from the *TSNameService*. Therefore the *TSNameService* would become a bottleneck especially when the system has massive remote tuple space access requests. In order to avoid such bottleneck, we design that every kernel has a *RemoteTupleSpaceServiceCacher* object shown in Figure 4-6, which will cache the stubs of all the remote *TupleSpaceServices* over the network. One stub is actually wrapped in the *TupleSapceServiceAdapter*.

4.2.3.3.4 Thread Pool

The thread pool provides threads to execute asynchronous tuple space access and agent reactive behaviors. By pooling threads, a thread usually can be created once and will last until the system terminates. This can reduce the overhead of thread creation and deletion. Figure 4-12 shows the class diagram of the *ThreadPool* module.

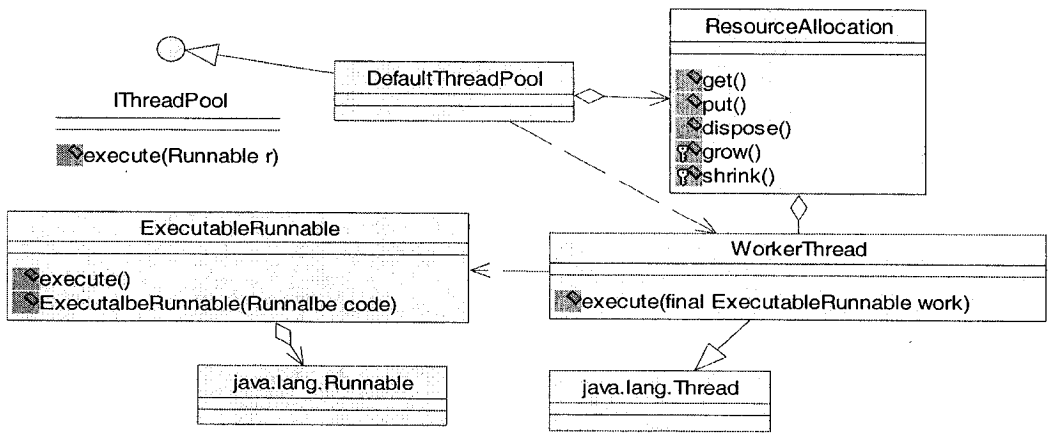


Figure 4-12 ThreadPool Module Class Diagram

In order to make the design clear, we do not program with Java thread directly. Instead, we define our own thread model to encapsulate the Java thread. It is captured through the *ExecutableRunnable* and *WorkerThread* classes. The *ExecutableRunnable* provides the code that will be executed by a *WorkerThread*. The code entry actually is a programmed Java *Runnable* interface. For instance, Figure 4-8 shows that the asynchronous access strategy *TSAsynRequestExecutor* implements the *Runnable* interface. Figure 4-9 shows that the reaction execution strategy *TSReactionExecutor* also implements the *Runnable* interface. The *TSReactionExecutor* encapsulates the agent programmable reactive behavior *Ireactive*.

On the other hand, the *WorkerThread* extends the Java thread. It will accept the *ExecutableRunnable* and execute the code. Most importantly, it needs to have the ability to manage itself. This means that after execution of the code, it can put itself back to the pool and wait for the next scheduling.

The pool is defined by the *ResoureAllocation*. It manages all the threads dynamically. It has a default pool size, which means that a certain number of threads will be created at system initial stage (startup time). The number of threads can grow or shrink according to the system workload. Finally, although the design is a little complicated, the thread pool component exposes only one method *execute(Runnable r)* to other modules.

4.2.3.4 Dynamic Behaviors

In this section, we will use sequence diagrams to illustrate the dynamic behaviors of the tuple space access. Here, we only show a remote asynchronous access scenario, and a reactive behavior detection and execution scenario. Although we cannot see the concurrency from sequence diagrams, it shows us how the kernel works in general. As we describe the sequence diagrams, we will point out where the concurrency exists.

4.2.3.4.1 Remote Asynchronous Access Scenario

We use *asynIn(tsid1,template)* primitive as an example to illustrate the dynamic behavior of a remote asynchronous tuple space access. It assumes that the tuple space is located in a remote container. Because of space limitation, we split one diagram into two as shown in Figure 4-13a and Figure 4-13b. One diagram describes what happens in one container. Figure 4-13a shows the dynamical behavior in the container that issues the access request. As an agent issue *asynIn(tsid1,template)* primitive, the access request comes to *TupleSpaceClient* object. *TupleSpaceClient* first transforms the request parameters into kernel level data, and then delegates the access request to the *TupleSpaceService* object. *TupleSpaceService* creates a *Future* object with a sequence number that represents this request. Then it asks the *AsynResponseManager* to remember the sequence number and the reference to the *Future* object. Since the tuple space is remote, *TupleSpaceService* will ask the *RemoteTupleSpaceServiceCacher* for the corresponding remote tuple space service adapter. After *TupleSpaceService* gets the remote adapter, it constructs a message called *TSAsynRemoteInRequest* and sends it to the remote tuple space service via its adapter. The *TSAsynRemoteInRequest* contains information such as the container name and sequence number. Then it returns to the agent with an agent level *Future* object that is not available to be used. The agent level *Future* object actually is a wrapper of the kernel level *Future* object.

After the access request has been processed by the remote tuple space service, the response will be put into the queue of the *AsynResponseManager*. Since the response has the sequence number, *AsynResponseManager* can assign the tuple retrieved from the response to the *Future* object that has the same sequence number. At this time, the *Future* object is available to the agent. In Figure 4-13a, there are two active objects: the agent and the *AsynResponseManager*. Both run concurrently.

Figure 4-13b shows the process in the remote container where the targeted tuple space is located. As the *AsynRequestManager* takes the request from its own queue, it will get the corresponding asynchronous execution strategy *TSAsynRemoteInExecutor* from *TSAsynRequestExectionGenerator*. Then *AsynRequestManager* calls *DefaultPool* to execute the strategy. It will not wait for the end of the execution. Instead it will proceed to the next request. The *DefaultPool* schedules a worker thread from its pool and assigns it the strategy. The worker thread does the actual execution. It first accesses *TupleSpaceHandler* to retrieve a tuple, and the worker thread may block until there is a matching tuple. After retrieving the matching tuple, the worker thread will construct a response object, and send it back to the requester container. Afterwards, the work thread will put itself back to the pool. In Figure 4-13b, there are also two active objects: *AsynRequestManager* and *WorkerThread*.

4.2.3.4.2 Reactive Scenario

We will illustrate how a reaction event is notified. It assumes that the detection of an event and execution of the corresponding reactive behavior happen in different containers. The dash line splits the diagram into two parts as shown in Figure 4-14. The left part shows the dynamic behaviors of notifying a remote agent the event in one container, and the right part shows the dynamic behaviors of executing a reactive behavior in another container.

The left part shows that when an agent writes a tuple, the *ReactionManager* will check whether any agent has registered for it. If so, the *ReactinoManager* will create a *ReactionEvent* object and perform a notification. Since the notified event has to be sent to a remote container, the *ReactinoManager* first needs to get the corresponding tuple space adapter. Then it sends the *ReactionEvent* object via the adapter through the *reactionTo*()

method. In fact, the *ReactionEvent* is put into the queue of the *HostReactionScheduler* in the remote container.

The right part shows the *HostReactionScheduler* retrieving the *ReactionEvent* from its queue. It first needs to find the corresponding reactive behavior from the *AgentReactiveBehaviorsList* of the registered agent. Then it constructs a *TSReactionExecutor* with the reactive behavior and asks the *DefaultPool* to execute it.

There are three active objects: *ReactionManager, HostReactionScheduler,* and *WorkerThread.* The *HostReactionScheduler* and *WorkerThread* are in the same container, and the *ReactionManager* is in another container.

4.3 Integration with JADE Platform

Here we do not want to describe how to integrate the reactive tuple space with JADE platform since it will involve too much detail. Instead, we list what should be done for proper integration. When considering integration, we follow the principle of minimal change to the JADE source code. The first change is to extend the interfaces of the JADE *Agent* class so that an agent can access tuple spaces. The second is to handle agent mobility so that when an agent migrates from one container to another container, reactive behavior registration information can be updated by the reaction manager. Thus it can keep location transparency for agent reactions. Finally, the initialization and termination handling of JADE is modified in order that tuple space services can start and stop properly.

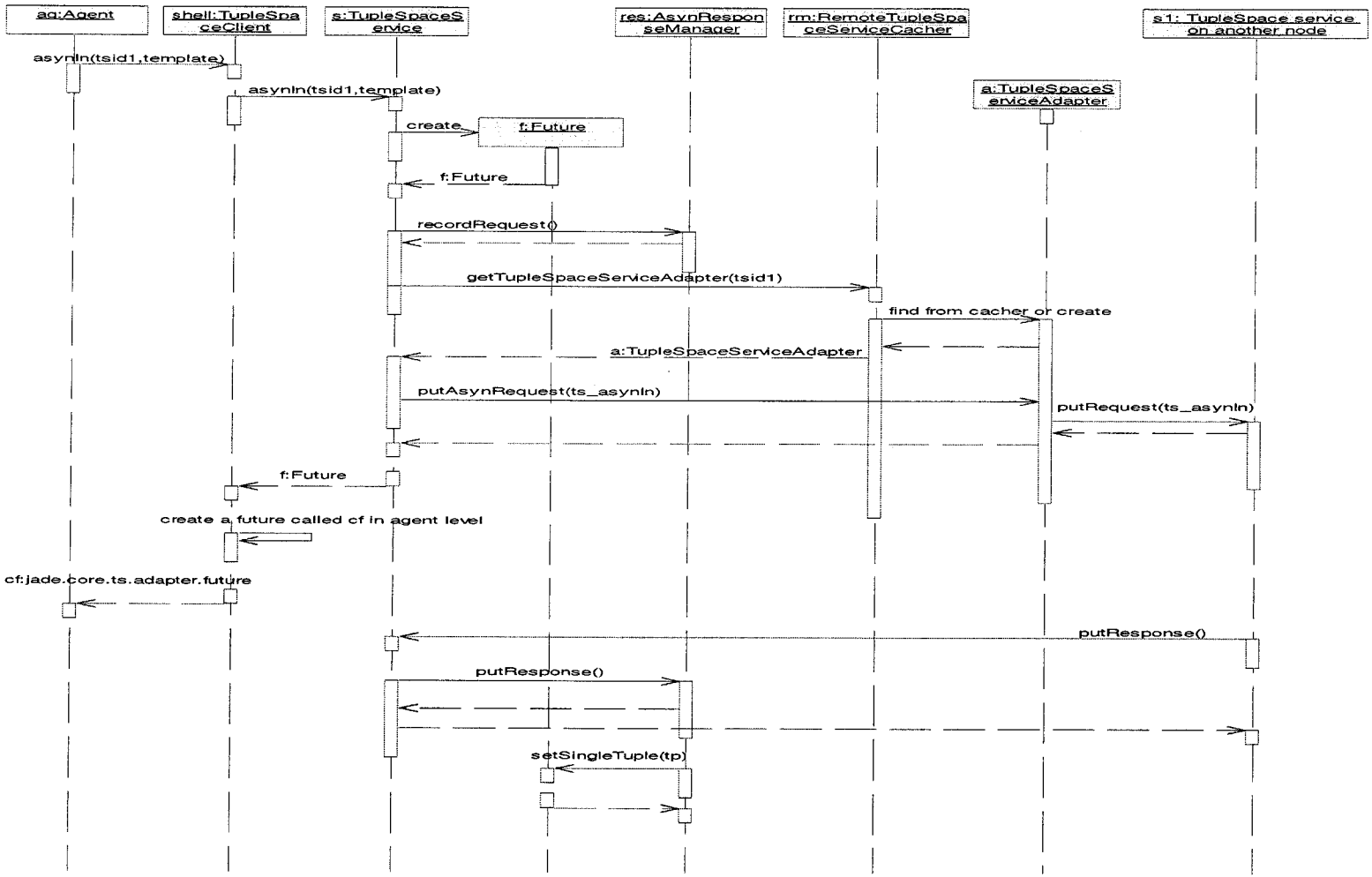


Figure 4-13a The Sequence Diagram of Asynchronous Remote *in* (Request Container)

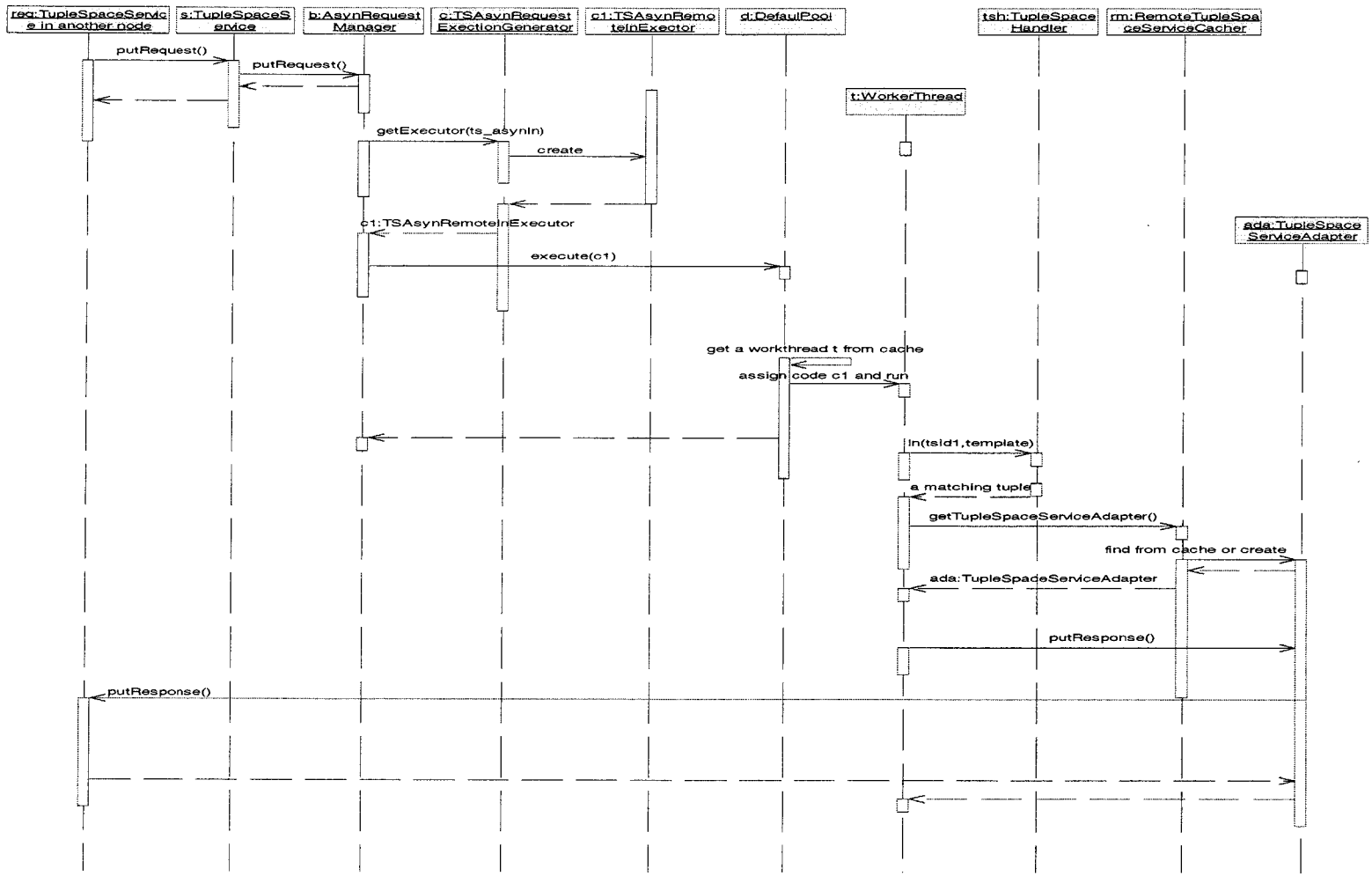


Figure 4-13b The Sequence Diagram of Asynchronous Remote *in* (Remote Container)

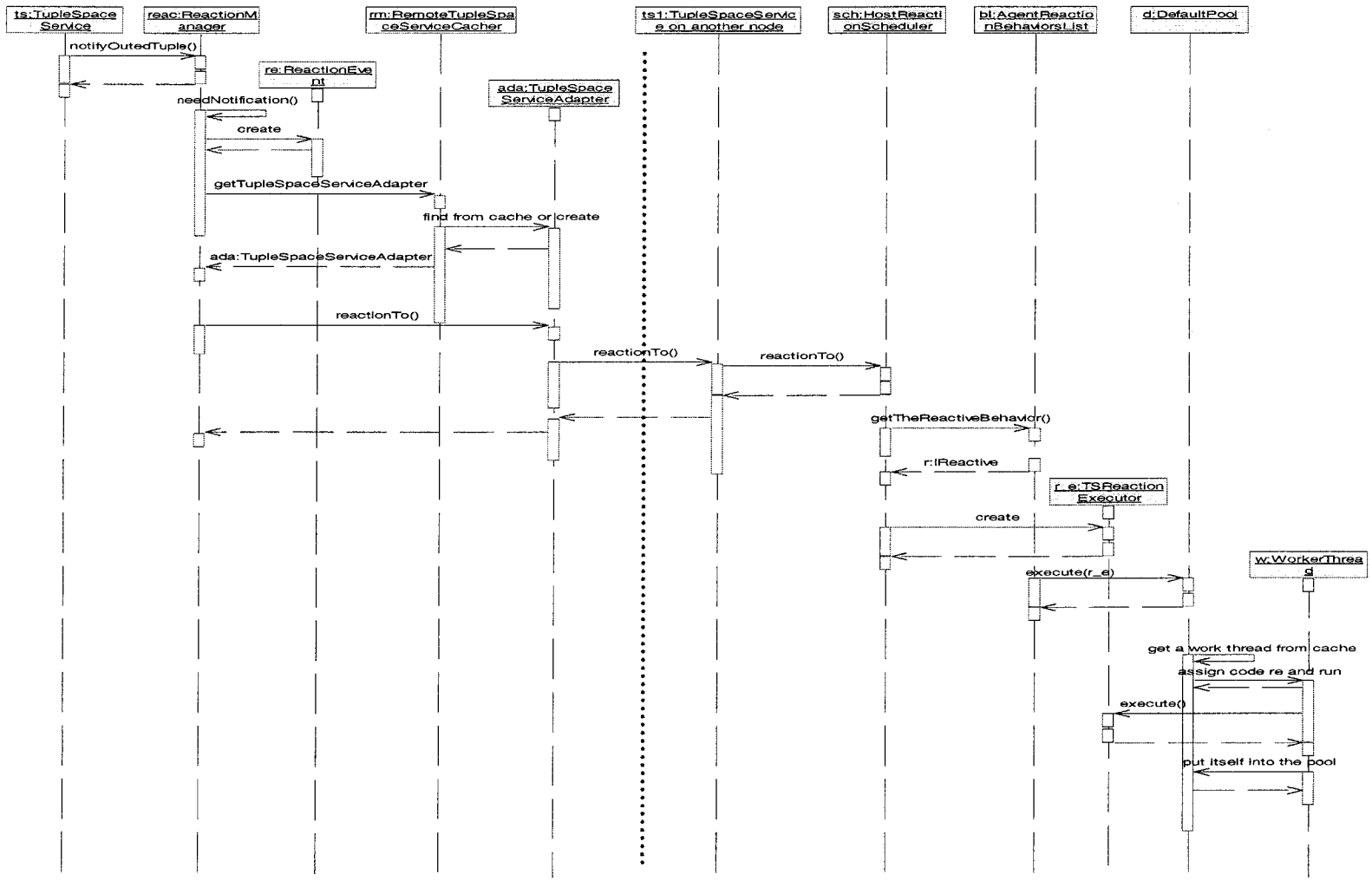


Figure 4-14 The Sequence Diagram of a Reactive Behavior Notification

Chapter 5 Performance Test

5.1 Statement

In order to evaluate the performance of the tuple spaces, we compare it with that of JADE ACL message passing as both are in the same platform and based on Java RMI.

Performance evaluation is conducted on information transfer latency, system bandwidth, and comparison with simulation of dynamic coupling using ACL. As shown in Chapter 1, the ability of the tuple space in supporting dynamic interaction is a key advantage of tuple space. In simulating dynamic coupling, ACL is used to simulate a mailbox agent to hold shared messages to be dynamically picked up by application agents. The objectives of these tests are to show: (i) the tuple space is almost as good as ACL in message latency, (ii) interaction bandwidth supported through tuple space(s) is compatible with that through ACL, and (iii) dynamic coupling is better supported by tuple space.

5.2 Performance Test

ACL and tuple space test cases are tested on the JADE platform that spans four nodes (computers). One of the nodes holds the main container *M1*, and each of the others holds an agent container named *C1*, *C2*, and *C3* separately. The experiments were conducted when little other network activities exist.

5.2.1 Latency

5.2.1.1 Test Case

ACL Test Case A sender agent sends a message to a waiting receiver agent, and then waits for an acknowledgement from the latter. There is only one round of message exchange between the sender agent and the receiver agent. The reason that we choose

only one round will be illustrated in Section 5.2.1.3. Latency is given by the difference between the time of receipt of the acknowledgement and the time of sending the message.

Tuple Space Test Case A producer agent writes a synchronous tuple into a tuple space for a waiting consumer agent, and then waits for an acknowledgement tuple from the latter. Both agents use synchronous primitives. Latency is measured at the producer agent as the difference between the time of receipt of the acknowledgement and the time of inserting the synchronous tuple.

5.2.1.2 Test Result

Both cases are tested with different background agents as the test parameter. Since the deployment of agents (and tuple space) affects the test results, we intend to keep the deployment fair to both cases. The sender (producer) agent and receiver (consumer) agent are in *C1* and *C2* respectively. The deployment of background agents is as follows. For the case with 10 agents, two pairs are in *C1* and *C2,* one pair in *C1* and *C3*, one pair in *C2* and *M1*, one pair in *C3* and *M1*. Every pair has a sender (producer) and a receiver (consumer). As we double the number of background agents, it follows the same deployment strategy. In addition, for tuple space, the background agents also use synchronous read (*in*) to retrieve a tuple, and the tuple space is located in the main container *M1*. The test results are shown in Table 5-1 and Figure 5-1.

5.2.1.3 Analysis

In the case of ACL, the sender agent first needs to get the proxy of the container where the receive agent is located from the main container, and then send the message through the proxy. The same holds for the receiver agent in order to send back an acknowledgement. Message latency goes up as the number of background agents increases because messages are sent from one container to another container sequentially

(since all the agents in a container share a same sending buffer). The buffer needs to be synchronized.

Background	0 agent	10agents	20agents	40agents	80agents
ACL (ms)	15.3	22.7	37.8	88.9	249.8
TS (ms)	15.5	24.3	41.2	96.1	232.1

Table 5-1 Latency

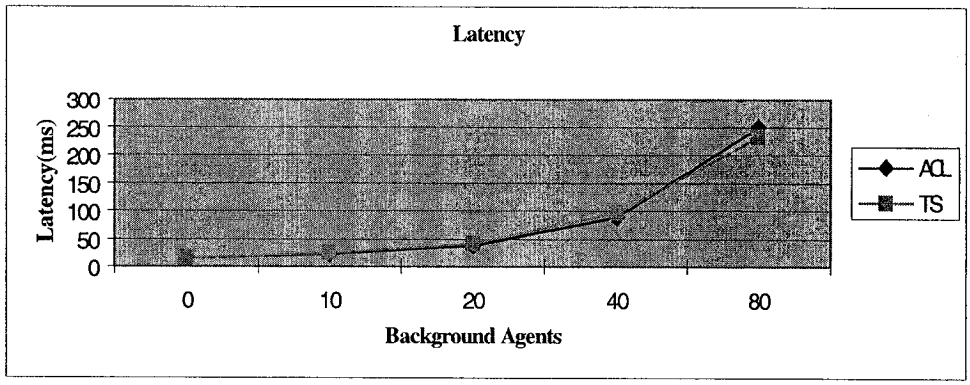


Figure 5-1 Latency

For the case of tuple space, agents from different containers can access a tuple space concurrently. The stage of the sender agent getting the acknowledgement is partially overlapped with the stage of the receiver agent writing the acknowledgement into the tuple space. The tuple space also needs to be synchronized. However, when tuple space is of small size, such synchronization does not take much time as the number of the background agent goes up. In fact, the size never exceeds half the number of agents because of the synchronous read in this test.

The test results involve a single round of message exchange. If message exchange runs multiple rounds, it can be predicted that ACL will perform better than tuple space. It is because that for ACL, after the fist time of message exchange, the rest will only needs one remote call. For tuple space, it always needs two remote calls even if the two become partially overlapped. However, if we change the location of the tuple space from the main container *M1* to an agent container such as *C1*, this will ensure a message will involve a single remote call. In such a case, the performance of tuple space may become

compatible with ACL again.

5.2.2 Bandwidth

5.2.2.1 Test Case

ACL Test Case In order to test bandwidth, every pair of sender and receiver agents use distinguished messages, and perform 1000 transactions between themselves. During each transaction, a sender agent sends a message to the corresponding receiver agent and waits for an acknowledgement from the latter.

Tuple Space Test Case The test cases in tuple space are the same as the ACL counterpart, especially the message size. Synchronous read (*in*) primitives are used.

Bandwidth is measured as follows. Total transaction time **Ta** measures the total elapsed time of all 1000 transactions in every pair of agents. Average time of single transaction **t** is equal to **Ta** divided by the total number of transaction. As there are many transactions from different pairs going on concurrently, the measurement of time **t** cannot reflect this. In fact, the value of **t** is smaller than the real one. Even though, it does not affect the description of the problem significantly. The bandwidth **B** is equal to 1 second divides **t.**

5.2.2.2 Test Result

For ACL, a scenario of 10 agents involves the following distribution. Two pairs are in *C1* and *C2*, one pair in *C1* and *C3*, one pair in *C2* and *M1* and one pair in *C3* and *M1*. As we double the number of agents, it follows the same deployment strategy.

The tests in tuple space involve two groups, (i) single tuple space, and (ii) dual tuple space. The deployment of agents follows the same strategy as ACL. In the single tuple space scenario, the tuple space is located in the main container *M1*. In the dual tuple space scenario, tuple space *A* is located in *M1* and tuple space *B* is located in *C1*. Agents in container *C1* and *C2* interact through tuple space *B*, and agents in *C1* and *C3* also

interact through tuple space *B*. Agents in container C2 and C3 interact through tuple space *A*. It is to maintain a fair distribution of remote calls during message exchange. Table 5-2 contains the collected data. Table 5-3 and Figure 5-2 show the bandwidth.

	10agents	20agents	40agents	80agents
ACL (ms)	13705	25469	53186	163209
Single-TS (ms)	16547	32187	68469	154094
2-TS (ms)	12203	24397	55796	125341

Table 5-2 Total Transaction Time

Calculate the bandwidth (number of transactions per second):

	10agents	20agents	40agents	80agents
ACL	365	392	376	245
Single-TS	302	311	292	260
2-TS	410	410	358	319

Table 5-3 Bandwidth

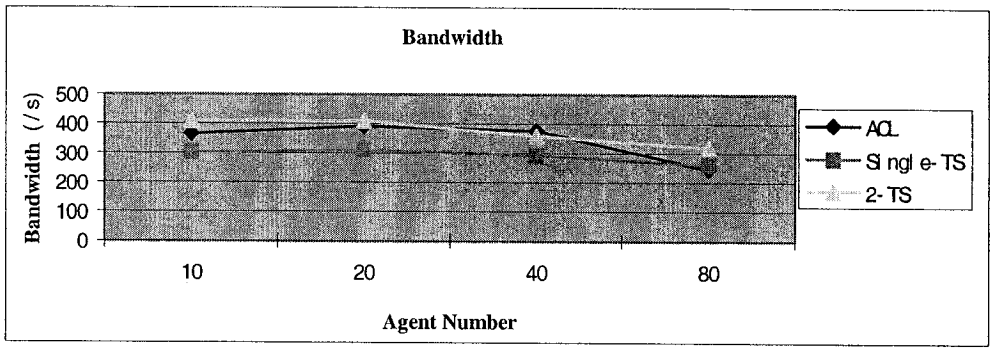


Figure 5-2 Bandwidth

5.2.2.3 Analysis

For ACL, we already know that messages are sent from one container to another container sequentially, and all the agents in a container share one sending buffer that needs to be synchronized. Figure 5-2 shows that as the number of agent increases, the number of messages in the sending buffer will also increase. This will lead to congestion and reduce the bandwidth.

For single-TS, the tuple space is centralized. This reduces concurrency among accesses from agents. Although the tuple space is segmented as we see from the design in order to

reduce the granularity of synchronization, ACL still has better performance in general.

To compare the performance of ACL and dual tuple space, consider the case with 10 agents. The same deployment of agents is used in both cases. In ACL, when an agent sends a message, the kernel needs to synchronize the sending buffer for putting or getting a message, to switch the thread context from the agent to the message manager, and then to send the message to the destination container through a remote call. The receiver agent will receive the message from its own private queue. On the other hand, in the dual tuple space case, there is no thread context switching since agents use synchronous primitives. In addition, the agents in different container can access a tuple space concurrently. The stage of writing a tuple and the stage of retrieving the tuple become partially overlapped. There are three pairs of agents that communicate through tuple space *B*. Thus tuple space *B* affects the test result significantly. The tuple spaces need synchronization as well. Besides this, the size of the tuple space is a very important factor. If it is very big, the process of searching for a tuple may take a long time. Fortunately, the size in this test case is small. Therefore, we could see that the dual tuple space performs as well as ACL, and even better in bandwidth. However, it can be expected that if we increase the number of nodes (computers) and agents in the dual tuple space case, its performance will slow down no matter how we deploy the two tuple spaces since there are only two points over the network to assist the message exchanges.

5.2.3 Dynamic Coupling

5.2.3.1 Test Case

ACL Test Case In order to simulate dynamic coupling, we design an agent that acts as a mailbox. A sender agent sends a shared message to the mailbox. A receiver agent sends a request message to the mailbox asking for a suitable message. Afterwards, it will send an acknowledgement to the sender agent. The latency is measured as in the earlier cases.

We only collected the data based on one round of message exchange for the same reason as that of Section 5.2.1.

Tuple Space Test Case The producer agent writes an asynchronous tuple in a tuple space for dynamic sharing. The consumer agent reads the asynchronous tuple from the tuple space, and then sends an acknowledgement through the tuple space. Latency is measured as in the earlier cases.

5.2.3.2 Test Result

Both cases are tested with different background agents as the test parameters. The sender (producer) agent and receiver (consumer) agent are in container *C1* and *C2*. The background agents and their deployments are the same as those in Section 5.2.1. In order to measure the cost, the sender (producer) agent, receiver (consumer) agent, and mailbox agent are controlled to start up simultaneously. The tuple space and the mailbox agent are in the main container *M1*. Test results are shown in Table 5-4 and Figure 5-3.

Background	0agent	10agents	20agnets	40agents	80agents
ACL (ms)	35.4	54.7	88.7	243.3	685.2
TS (ms)	16.0	45.6	76.5	164.7	341.9

Table 5-4 Dynamic Coupling Latency

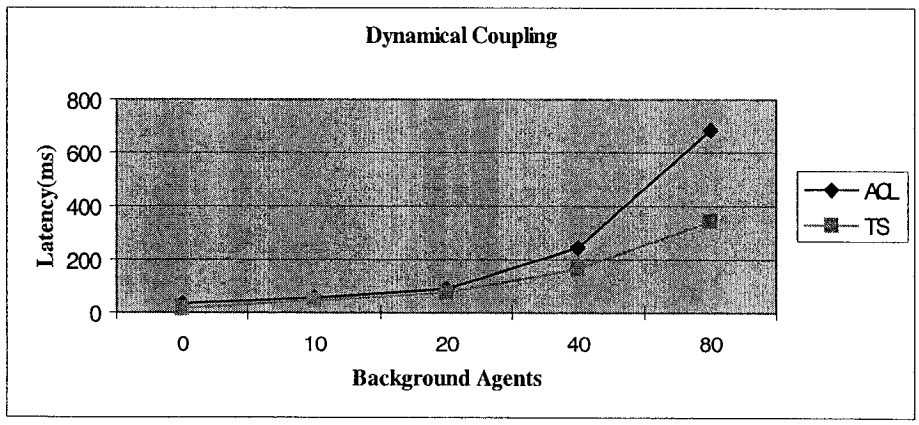


Figure 5-3 Dynamic Coupling Latency

5.2.3.3 Analysis

The runtime process of the ACL test case is as follows. The sender agent sends a message

to the mailbox agent for sharing, and the receiver agent sends a request to the mailbox. The mailbox then checks for the message and sends it to the receiver agent. Then the receiver agent sends an acknowledgement to the sender agent. However, the stage of the sender agent sending a message to the mailbox agent is overlapped with the stage of the receiver agent sending a request to the mailbox agent. Therefore, it can be understood that the process has three stages. In addition, we already know the exchange of message from one container to another container is sequential, and the sending buffer needs to be synchronized.

For tuple space, the process is as follows. A producer agent writes a tuple into tuple space and the consumer agent retrieves the tuple. Then the consumer agent writes an acknowledgement tuple which will be eventually picked up by the corresponding producer agent. Concurrency among these accesses exists. Therefore the whole process actually involves only two stages. As long as the tuple space size is small, dynamic couplings among multiple pairs of agents do not incur significant influence on each other.

Chapter 6 Conclusion

6.1 Conclusion

Software agent is an interesting paradigm in building distributed system. This thesis focuses on building an efficient reactive tuple space coordination media, which is integrated with the JADE platform. Thus an agent can choose to have an interaction through either message passing or tuple space. The existence of the reactive tuple space is not intended to replace the message passing. However, it means to provide another choice to facilitate the development of the multi-agent software systems.

During the requirement analysis phase, we have reviewed other related tuple spaces and propose the reactive tuple space model. To an agent programmer, the reactive tuple space provides synchronous and asynchronous primitives to implement proactive behaviors, and reactive primitives to implement reactive behaviors. A formal description of the reactive tuple space is presented using the state machine model.

In order to make the reactive tuple space efficient, a replication protocol is also provided to maintain copies of selected tuples at some nodes. In this protocol, it hides the access latency by making the process of invalidation work at background. Using the view model, the correctness of the protocol is also proved.

Efficiency is a top priority as we design and implement the reactive tuple space. For instance, the distribution of multiple tuple spaces over the network enables different agent groups to execute concurrently. A tuple space is segmented in order to reduce the granularity of synchronization. The invocation of a reactive behavior is triggered at the node where the agent is located. A tuple space server is only responsible for reactive event detection so as to reduce the server side workload. Compared with message passing in JADE, the experiments conducted show that tuple space can provide competitive

performance in terms of interaction latency and bandwidth when tuple space is of reasonable size. This is especially true in dynamic coupling.

6.2 Lessons Learned and Future Work

However, additional efforts can be made to improve the system efficiency. The property of a tuple space requires that the multiple accesses have to be synchronized. It means that multiple accesses have to be serialized. Although we have segmented a tuple space, this seems to be inadequate. In fact, a tuple space can be partitioned into several nodes over the network to promote concurrency. In addition, a proper partition policy can also help to achieve the access locality.

In the current implementation, an asynchronous tuple access will consume a system thread. So does the execution of a reactive behavior (a remote synchronous tuple access will also consume a thread, but it is done in the Java RMI layer). The advantage of such a solution is that it promotes concurrency. However, if a multi-agent system involves a lot of asynchronous tuple accesses and reactive behaviors invocations, it can reach the thread saturation point of a Java virtual machine. We recognize that the system resource is an important factor that affects application performance. If kernel can monitor the consumption of system resources, a system thread then can be assigned several tasks dynamically before the system reaches saturation. A task here represents an asynchronous access or the execution of a reactive behavior. This will help to avoid system performance from dropping precipitously.

Especially in case of the single processor system, we may have another choice to handle the tuple accesses at the server site. For one tuple space, we only have one thread be responsible for scheduling and executing all its access requests (no matter synchronous or asynchronous) at server site. The thread functions similarly as the *AsynRequestManager*, but it does not generate any other thread. The execution of a request can never be blocked.

A request will be rescheduled if it cannot find the matching tuple. Therefore, it eliminates the overheads of context switching and thread synchronization since only one thread accesses a tuple space. On the contrary, the multi-thread strategy (one thread per request) in current implementation requires the Java virtual machine to schedule different threads and synchronize these concurrent accesses. In addition, the single thread strategy also reduces the resource consumption significantly and avoids much competition with the agents.

Another effort is to find an efficient way to tune the performance [Ji98]. For instance, the current performance test analysis is by way of reasoning through the source code. Although *java Profiler* tool is used, it cannot measure system internal states. However, we may want to measure the runtime internal states so as to give a more accurate explanation and thus easier to improve the performance. These states may include, for example, the number of threads in the 'ready' queue, and how long a thread has been waiting for a condition. It also reflects design for testability.

Bibliography

[Aglets] Aglets, http://sourceforge.net/projects/aglets

[Ahamad95] M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, P.W. Hutto, "Causal memory: Definitions, implementation, and programming", Distributed Computing, 1995, pp.37–49.

[Arbab98] F. Arbab, "What Do You Mean, Coordination", Bulletin of the Dutch Association for Theoretical Computer Science (NVTI), March 1998.

[Bernstein96] P. Bernstein, "Middleware", Communications of the ACM, Vol. 39, Issue 2, 1996.

[Booch99] G. Booch, J.Rumbaugh, I.Jacobson, "The Unified Modeling Language User Guide", Addison Wesley, 1999.

[Cabri00] G. Cabri, L. Leonardi, F. Zambornelli, "Mars: A programmable coordination architecture for mobile agents", IEEE Internet Computing, 2000.

[Carey00] James Carey, Brent Carlson, Tim Graser, "SanFrancisco Design Patterns: Blueprints for Business Software", Addison-Wesley, 2000.

[Carriero85] N. Carriero, D. Gelernter, "The S/Net's Linda Kernel", Proceedings Symp, Operating Systems Principles, December 1985.

[Carriero90] N. Carriero, D. Gelerner, "How to write parallel programs: A first course", MIT Press, 1990.

[Carter91] J.B. Carter, J. K. Bennett, W. Zwaenepoel, "Implementation and performance of Munin", In Proceedings of the 13th Symposium on Operating System Principles, October 1991, pp.152-164.

[Ciancarini96] P. Ciancarini, "Coordination Models and Languages as Software Integrators", ACM Computing Surveys, Vol. 28, No. 2, June 1996, pp.300-302.

[Ciancarini98] P. Ciancarini, A. Omicini, F. Zambonelli, "Coordination model for multi-agent systems", from www.agentlink.org.

[Corradi95] A. Corradi, L. Leonardi, F. Zambonelli, "A Scalable Tuple Space Model for Structured Parallel Programming", Working Conference on Massively Parallel Programming Model, Berlin, October 1995, pp.120-128.

[Emmerich00] W. Emmerich, "Software Engineering and Middleware: A Roadmap", ACM Proceedings of the Conference on the Future of Software Engineering, May 2000.

[FIPA] Foundation for Intelligent Physical Agents (FIPA), http://www.fipa.org/

[FIPA_OS] FIPA_OS, http://www.nortelnetworks.com/products/announcements/fipa/

[Fok03] C.L. Fok, G.C. Roman, G. Hackmann, "A Lightweight Coordination Middleware for Mobile Computing", Technical Report WUCS-03-67, Washington University, Department of Computer Science and Engineering.

[Gamma94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994.

[Gelernter85] David Gelernter, "Generative communication in Linda", ACM Transactions on Programming Languages and Systems, 1985, pp.80-112.

[Gelernter92] D. Gelernter, N.Carriero, "Coordination Languages and their Significance", Communications of the ACM, 1992, pp.97-107.

[Gharachorloo90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Henessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors", Proceedings of the 17th annual international Symposium on Computer Architecture, 1990, pp.15–26.

[Girard99] G. Girard, H.F. Li, "Evaluation of two optimized protocols for sequential consistency", In Proceedings of the 32nd Annual Hawaii International Conference on System Sciences, January 1999.

[Grasshopper] Grasshopper, http://www.grasshopper.de/

[Gray96] J. N. Gray, P. Helland, D. S. P. O'Neil, "The dangers of replication and a solution", In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Canada, June 1996, pp.73-82.

[Hofmeister99] C. Hofmeister, R. Nord, D. Soni, "Applied Software Architecture", Addison Wesley, 1999.

[JADE] JADE, http://sharon.cselt.it/projects/jade/

[JavaSpaces] JavaSpaces Service Specification, http://wwws.sun.com/software/jini/specs

[Ji98] M. Ji, E.W. Felten, K. Li, "Performance Measurements for Multithread Programs", Proceedings of the 1998 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pp.161-170, 1998.

[Johnson98] R.E. Johnson, B. Foote, "Designing Reusable Classes", Journal of object oriented programming, June 1998, Volume 1, pp.22-35.

[Keleher92] P. Keleher, A.L. Cox, W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory", In Proc. 19th Annual Int'l Symp. on Computer Architecture, May 1992, pp.13-21.

[Kemme00] B. Kemme, G. Alonso, "A new approach to developing and implementing eager database replication protocols", ACM Transactions on Database Systems, Volume 25, No. 3, September 2000, pp.333-379.

[Kendall98] E.A. Kendall, P.V.Murali Krishna, "An Application Framework for Intelligent and Mobile Agent Systems", ACM Computing Surveys Symposium on Application Frameworks, ed. M. Fayad, D. C. Schmidt, ACM, 1998.

[Kendall99] E.A. Kendall, "Role models-patterns of agent system analysis", BT Technology Journal Vol.17, No. 4, October 1999.

[Kranz94] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, B.H. Lim, "Integrating Message-Passing and Shared-Memory: Early Experience", in Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1994, pp.54-63.

[Lamport78] L. Lamport, "Time, clocks, and ordering of events in a distributed system", Communications of ACM, July 1978, 21(7):558-565.

[Lamport79] L. Lamport, "How to make a multiprocessor computer that correctly executes multi-process programs", IEEE Transactions on Computers, 1979, C-28:690-691.

[Lange99] D.B.Lange, M.Oshinma, "Seven Good Reasons for Mobile Agents", Communications of the ACM, 1999, 42(3):88-89.

[Lavender95] R.G. Lavender, D.C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming", Pattern Languages of Programming, Illinois, 1995.

[Li89] K. Li, P. Hudak, "Memory Coherence in Shared Virtual Memory Systems", ACM Trans. Computer Systems, 1989, 7(4): 321-359.

[Li03] H.F. Li, G. Girard, "View Consistencies and Exact Implementations", Parallel Computing, Vol 29, No 1, January 2003, pp.37-67.

[LIME] LIME, http://lime.sourceforge.net

[Malone94] T. Malone, K. Crowston, "The interdisciplinary study of coordination", ACM Computing Surveys, March 1994, pp.87-119.

[Picco99] G.P. Picco, A.L. Murphy, G.C. Roman, "LIME: Linda Meets Mobility", 21th International Conference on Software Engineering, May 1999.

[Picco03] G. P. Picco, A. L. Murphy, G.C. Roman, "Developing Mobile Applications: A LIME Primer", 2003, available http://www.cse.wustl.edu/mobilab/Publications/index.htm

[Riehle98] D. Riehle, T. Gross. "Role Model Based Framework Design and Integration", In Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98), ACM Press, 1998, pp.117-133.

[Rimassa03] G. Rimassa, "Runtime Support for Distributed Multi-Agent Systems", Ph. D. Thesis, University of Parma, January 2003.

[Rowstron96] A. Rowstron, A. Wood, "Solving the Linda multiple rd problem", In Coordination Languages and Models, Proceedings of Coordination '96 P.Ciancarini and C. Hankin, Eds, LNCS Springer Verlag, 1996, pp.357-367.

[Rowstron98] A. Rowstron, "WCL: A coordination language for geographically distributed agents", World Wide Web, Volume 1, Issue 3, 1998, pp.167-179.

[Schmidt02] Douglas C. Schmidt, Stephen D. Huston, "C++ Network Programming: Mastering Complexity Using ACE and Patterns", Addison Wesley, 2002.

[Shrivastava99] S.K. Shrivastava, M.C. Littl, "A method for combining replication with caching", Reliable Distributed Systems, Proceedings of the 18th IEEE Symposium, October 1999, pp.316-321.

[Shoham93] Y. Shoham, "Agent oriented programming", Artificial Intelligence, 1993, 60:51-92.

[Snyder02] J. Snyder, R. Menezes, "Using Logical Operators as an Extended Coordination Mechanism in Linda", Proceedings of Coordination 2002, York, England, April 2002.

[TSpace] IBM Tspace, http://www.alphaworks.ibm.com/tech/tspaces

[Weihl93] William E. Weihl, "Specification of Concurrent and Distributed systems", Chapter 3 of "Distributed Systems", Addison-Wesley, 1993

[Wiesmann00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, G. Alonso, "Database Replication Techniques: a Three Parameter classification", Proc. of 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000), Nürnberg, Germany, 2000.

[Wood00] M. F. Wood, S. A. DeLoach, "An Overview of the Multi-agent Systems Engineering Methodology", Proceedings of the First International Workshop on Agent-Oriented Software Engineering, June 2000, pp.207-220.

[Zeus] Zeus, http://more.btexact.com/projects/agents/zeus/

[Zhang04] Y. Zhang, "A Tuple Space Based Agent Programming Framework", Master of Computer Science Thesis, Concordia University, June 2004.