A Soft Real-Time Self-Planned Multi-Agent Framework

De Jin

A Thesis
in
The Department
of
Computer Science and Software Engineering

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

September 2005

Your file Votre référence
ISBN: 978-0-494-34593-1
Our file Notre référence
ISBN: 978-0-494-34593-1

Abstract

A Soft Real-Time Self-Planned Multi-Agent Framework

De Jin

This thesis presents a real-time multi-agent framework targeted for soft real-time applications. Such an application involves missions each with a soft deadline. A mission is decomposed into phases. The work in each phase can be performed differently, depending on the remaining time. A tradeoff between the quality of the result and the run time requirement is permitted. To make use of this tradeoff, our framework provides a planner that dynamically manages the selection of solution to be used in a phase, based on its knowledge of the deadline and runtime system state. Therefore, the planner may select a faster solution with inferior results in order to meet the mission deadline. This model leads to a real-time self-planned multi-agent framework that is presented as a set of application programming interfaces for the application designer. An implementation of this framework on the Real-Time JVM is also presented. Use of the framework is illustrated with an example intelligent security monitoring system. Finally, an evaluation of the overhead of the planner and its effect on the overall deadline miss rate is reported. Preliminary experimental studies reveal that the planner overhead is rather insignificant, and the self-planned approach can potentially improve both the quality of the result and the deadline miss rate.

Acknowledgments

I would like to express my gratitude and respect to my supervisors Dr. Hon F. Li and Dr. Rajagopalan Jayakumar for their invaluable guidance, encouragement and support during the whole period of my research work. Dr. Li and Dr. Jayakumar not only teach me knowledge at class, during meetings and through emails, they also try their best to enlighten me of how to do research, how to analyze and solve a problem.

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

Finally, I wish to thank my parents and my wife and my forthcoming baby.

Table of Contents

List of Figures

List of Tables

List of Scripts

Chapter 1

Introduction

1.1 Motivation

In recent years, the increasing use of personal computers, Ethernet applications and so on is rapidly making reactive systems (that depend strongly on an external stimulus or a set or sequence of external stimuli) more open, distributed and cost-effective. Hence, reactive distributed applications such as multimedia, complex information retrieval, and system monitoring and controlling are becoming increasingly common and necessary. Multi-agent systems [1] are systems composed of multiple interacting computing elements known as agents. Agents are computing subsystems with two important characteristics: *Autonomy* — an ability to be active without relying on direct and continuous intervention of its environment; and *Sociality* — an ability to interact with other agents in ways analogous to human interactions in society for cooperation, coordination, negotiation, and so on. These features of the agent-oriented paradigm allow for a greater level of decentralization, a desirable characteristic in an open distributed environment. So, the multi-agent model can be meaningfully applied to capture the presence of the abovementioned applications because of the extensibility, flexibility, ease of use and naturally distributed nature of this paradigm.

An important characteristic of many open reactive applications is that they have soft

real-time constraints and flexible, rather than strict, functional requirements. Soft real-time constraints mean that failure to meet a deadline is not necessarily considered to be a failure of the application or system; and flexible functional requirements mean that acceptable results are diverse and may differ in quality aspects. These characteristics are useful in managing agent activities in the resulting application design. For example, when an agent does not have enough time to complete its task, we can either assign more resources (CPU, Memory, Bandwidth, etc.) to it, or simplify its task at the expense of a lower quality level of its results. Dynamic allocation of more resources to an agent may not be feasible in a resource-tight environment and the additional resource management also will incur non-negligible management overhead. Hence managing the quality of results or services of agents may be a reasonable alternative strategy to be applied in such applications.

Under *quality-time tradeoff*, an agent can perform the task faster or slower and try to meet the deadline eventually. Quality-time tradeoff is widely used in soft real-time/flexible output systems. This principle can make soft real-time applications more adaptable to their run-time environments. Based on the available remaining time and resources, these systems produce the output with different qualities. The goal of using such principle is to optimize the system response in the presence of soft deadline expectations. Hence, this principle can be well used in managing/scheduling agents to complete their tasks involved in common or distinct missions.

When designing a distributed soft real-time multi-agent system, such as the

abovementioned applications, one approach is to build it from scratch by incorporating the quality-time tradeoff features in agent designs. Obviously such an approach will complicate the underlying agent design required to provide the functional outputs to the application. Not only that an agent has to perform its mission but also manage its course of action depending on the remaining time available. A reasonable alternative is to develop a soft real-time agent management framework that has built-in quality-time tradeoff management and agent scheduling services so that different applications can be built on top of it without having to custom design each quality-time tradeoff detail. This thesis is dedicated to the design of such a framework and the demonstration of its use.

In both the real-time system research [2–7] and the real-time AI research [8–16] communities, there are numerous reported uses of quality-time tradeoff to handle soft real-time constraints. Nevertheless, there are relatively few systematic research efforts on practical methodologies for applying this principle in a soft real-time application development process. For example, research work from the traditional real-time community is mainly process-based, which, as we mentioned before, lacks the ability to deal with open distributed environments. Additionally, most research work from the real-time AI community focuses more on individual, so-called sophisticated, agent and the coordination of a small group of these agents to handle specific application requirements. The flexibility and extensibility of their solutions are limited. So to develop a general solution, a new soft real-time agent model along with a framework is needed. The following are important focuses in our research quest:

(i) An agent programming model that combines all meaningful mainstream technologies from relevant research communities.

(ii) A simple but effective real-time multi-agent self-management framework to support a wide spectrum of soft real-time multi-agent applications.

With these two focuses, we hope to develop a model that is flexible and easy to use, and also will not incur much degradation in performance in the resulting implementation. In this thesis, an agent-programming model called SPMM is proposed. SPMM not only provides a soft real-time agent-programming model integrating multiple technologies related to agent-based systems, but also provides an easy to use programming environment for implementing the resulting design. In particular, SPMM provides a role-based agent behavior model in abstraction, a model that supports the analysis and design of multi-agent systems from the perspective of agent-oriented software engineering. As illustrated through a case study on an intelligent security monitoring application in this thesis, SPMM demonstrates flexibility and extensibility in dealing with different kinds of applications and also helps developers to better achieve the aforementioned goal — low deadline missing rate, high-average-quality output and high system utilization.

1.2 Related Work

This section presents some important related research reported in the literature.

1.2.1 Traditional Real-Time System

In order to cope with changing load and failure conditions and to maximize system utilization, the principle of quality-time tradeoff is widely used in the traditional real-time community. Depending on different assumptions on task properties and available resources, this principle can be applied at different stages of the lifecycle of a real-time task. For example, admission control based on the resource requirements applies the strategy in managing task admission, and on-line negotiation applies this strategy in managing task execution. However, no matter where the strategy is applied, it is commonly assumed that a soft real-time task can have different quality levels associated with different deadlines and resource requirements. The following reviews several instances of applying quality-time tradeoff.

1.2.1.1 On-line Admission Control with different QoS levels

Admission control manages the set of tasks to be supported by the system. Traditional on-line admission control assumes that a task arrives with a fixed QoS requirement (and therefore a fixed resource requirement); and based on resource availability, a task may be admitted or rejected. However, in on-line admission control [2] with QoS modeling, a task has several different output qualities each of which is associated with the required execution time and resources. Intrinsically the model assumes that task requirements are deterministic and known in advance. Task arrival time is unpredictable. The system admits a newly arrived task only if it is feasible; that is, the requirements of one of the

quality levels can be met by the system state.

Although on-line admission control with different QoS levels allow the system to dynamically manage its commitments to an open environment, it does not have the provision to manage subsequent changes of system state that may jeopardize its commitments. In particular, subsequent to admission, due to inaccuracy of both task requirements and runtime resource estimates (including network bandwidth and computational bandwidth), an admitted task may miss its deadline unless further quality degradation and management actions are taken. Providing quality consideration in admission control is a first step but not a complete solution to manage tasks; especially those that may have a longer time span or more complex mission requirements.

1.2.1.2 On-line QoS Negotiation

On-line QoS negotiation extends admission control with dynamic changes of the QoS levels of a task when the QoS that was accepted during admission becomes difficult to achieve. Such changes include increasing the QoS level of a task when the system is underutilized or decreasing the QoS level of a task when the system is overloaded. Such a strategy can improve the overall performance of the system and maximize the system utilization. The following is a typical example of on-line QoS Negotiation strategy [3].

QoS negotiation: This strategy involves a mechanism for QoS (re)negotiation as a way to ensure graceful degradation in case of overload, failures, or violation of pre-run-time assumptions. In this model, a distributed application involves a single

overall mission (e.g. flight control, shipboard computing, automated manufacturing and so on). A mission is composed of a set of tasks, each of which requires a set of resources/services. Tasks can arrive at any node of the system. There are no precedence constraints among different tasks.

There are several QoS levels for each task. These levels are specified upon task arrival as alternative acceptable performance levels for the task. Each QoS level has different resource requirements and a corresponding benefit (called reward) of executing the task at that QoS level. The higher reward the task has, the more important it is. Each QoS level may involve a set of modules that may be different from the one in the other QoS level. Within a given QoS level, finishing the set of modules by the deadline accomplishes the task.

A node can accept a task in which case a QoS contract is said to be signed by promising to execute the task at one of its specified QoS levels. Alternatively, the task may be rejected in which case no contract is signed. There is a penalty if the task is rejected and a different (higher) penalty if the QoS contract is violated. The latter allows a node to break the contract unilaterally and offer compensation.

At each node, there is a local scheduler, which schedules the tasks for maximum reward. It upgrades the task with the largest reward differential between QoS levels when the system is underutilized, and downgrades the task with the smallest reward differential when the system is overloaded. Underutilization is measured by idle time, while overload is measured by deadline misses. When tasks execute at a level that is lower than the

maximum QoS level declared in their contract, the system is said to have an unfulfilled potential reward (UPR). A global scheduler periodically migrates tasks from nodes with high UPR to nodes with low UPR to balance the load and gain higher global reward.

1.2.1.3 Feedback Control Real-Time Scheduling

In order to meet performance constraints in open environments where both load and available resources are difficult to predict, this strategy assumes that task execution times are unknown or variable. In other words, this solution meets performance guarantees without accurate knowledge of task execution parameters. Unlike ad hoc algorithms (e.g. the previous two strategies) based on intuition and testing, this solution has a basis in the theory and practice of feedback control scheduling. Based on a set of control equations, it applies control theory to attain scheduling guarantees. Dynamic changes of QoS are the common factor between this approach and the earlier examples. Depending on the feedback from system load and resource state, QoS applied to different tasks is adjusted to ensure schedulability of the tasks. An example is the Distributed Feedback Control real-time Scheduling (DFCS) [4] described below.

DFCS: In DFCS, a distributed application operates in an open environment where both the system load and the available system resources are difficult to predict. The application uses two sets of metrics: primary metrics that are maintained at specific levels, such as global or local deadline missing rates; and secondary metrics that are variable, such as CPU utilization.

The application involves a set of tasks that arrive at nodes in arbitrary patterns. For each task T_i, there are N different QoS service levels ($N > 1$). Task T_i running at service level q, $0 \leq q \leq N$, has a deadline $D_i[q]$ and execution-time $C_i[q]$. The required CPU utilization, $J_i(q) = C_i[q]/D_i[q]$, of the task is a monotonically increasing function of the service level q, which means that a higher QoS will require more CPU utilization.

In this solution, two controllers (a distributed controller and a local controller) are introduced in order to dynamically adjust the QoS levels of tasks based on the changing environment. The responsibility of the Distributed Feedback Control System is to limit the global missing rate of the application, control the overall service level of the system, and balance the system load by migrating tasks among nodes. It includes two main parts: a miss rate controller, which takes care of system overload; and a utilization controller, which manages the utilization of the system. On the other hand, the responsibility of a Local Feedback Control (LFC) system is to control the miss rate of locally admitted tasks, and to control the secondary metric. Three important parts are included in the local controller: a miss rate controller, which takes care of local overload; a utilization controller, which manages the local node utilization; and a Service Level Ratio Controller (SLR), which addresses the CPU utilization as the secondary metric.

The distributed controller manages the local controllers via the QoS set point. The entire control system of the local node becomes an actuator of the distributed controller to control the state of one local node. The local controller manipulates its actuators in the LFC to achieve the target QoS set point.

1.2.1.4 Summary

In summary, we would like to pinpoint the key differences between the above strategies and their main drawbacks when these models are used to develop new soft real-time applications in open environments.

Comparison of Different Strategies: The above-mentioned approaches involve a tradeoff between quality and time. All of them intend to deal with dynamic systems with some unpredictable features, such as unknown task arrival patterns or unpredictable system load or resource status. On-line admission control focuses more on controlling a task before its execution rather than during its execution. It admits a new task by matching one of the different resource requirements of the task with the current available resources. Also, it assumes that resources can be reserved for a task and the admitted task can execute without violating a priori assumed load and failure conditions. However, on-line QoS negotiation and Feedback Control scheduling propose mechanisms to ensure graceful degradation in case of overload, failure, or violation of pre-run-time assumptions. Moreover, in Feedback Control scheduling, the performance of the real-time system is modeled in some coarse-grained manner that represents the relation between aggregate QoS and aggregate resource consumption. This is as opposed to QoS negotiation, a fine-grained model that requires knowledge of individual task execution times. Additionally, in Feedback Control scheduling, feedback is used as a primary mechanism to adjust resource allocation in the absence of a priori knowledge of resource supply and demand. However, as an optimization-based QoS adaptation technique, QoS negotiation

assumes accurate models of application resource requirements.

Key Issues: Generally, all of these three approaches belong to process-based strategy. In such strategies, the design process involves functional decomposition of the system. It is unnatural to handle the inherent complexity of open and dynamic systems. In addition, these approaches focus more on individual tasks. A general assumption is usually made that there is less dependency between different tasks. In other words, these approaches have a relatively weak ability to deal with systems having complex task structures and dependencies. For a real-time system in an open environment, however, this is usually the case. Its tasks normally have sophisticated data or functional dependencies. So, these approaches will face difficulties when they are directly used for the abovementioned soft real-time applications.

1.2.2 Real-Time AI System

Traditionally, artificial intelligence techniques have not been utilized in real-time environments due to their highly unpredictable performance. Generally this is a result of the types of problems that AI research focuses on — the problems with complex algorithms and facing unpredictable environments. A major step forward in real-time AI research is represented by the use of approximate algorithms. To date, real-time AI research has been interested in two main types of approximate algorithms: Anytime Algorithms and Multiple (Approximate) methods.

1.2.2.1 Anytime Algorithm

An anytime algorithm [10–12] is an iterative refinement algorithm where a "default" answer is first generated and then refined through multiple iterations. It is also true that the quality of the solution increases proportionally to the amount of time (number of iterations) the algorithm is executed. In addition, anytime algorithms always produce a result regardless of when they are interrupted.

Figures 1-1 and 1-2 provide a detailed example taken from Zilberstein [11] of an anytime algorithm and its performance profile. Figure 1-1 is a simple anytime algorithm for solving the Traveling Salesman Problem (TSP). This algorithm quickly constructs an initial tour, registers that result (making it available should the algorithm be halted), then repeatedly chooses two random edges and decides if switching them will result in a better tour. Figure 1-2 shows the runtime and quality results obtained by running this algorithm on randomly generated inputs and stopping it at a randomly generated time.

1.2.2.2 Multiple and Approximate Methods

The multiple method approach [8, 13–16] does not rely upon continuous processing to solve a problem. Rather, a set of methods is available to solve a task. Each method has different characteristics that make it more or less appropriate given the current conditions. Every method solves the same problem, but varies in the amount of time it needs to produce the result and the quality of the result. There is a quality-time tradeoff between

methods where a shorter execution time is achieved through reducing the quality of the result.

```
AnyTime_TSP(V, iteration){
        T = Init_Tour(V)
        cost = COST(T)
        Register_Result(T)
        for i = 1 to iteration{
            e1 = Random_Edge(T)
            e2 = Random_Edge(T)
            diff = COST(T) – COST(Switch(T,e1,e2))
            if diff > 0 then{
                T = Switch(T,e1,e2)
                cost = cost - diff
                Register_Result(T)
        }
        Signal(Termination)
        Exit()   }
```

Script 1-1 Sample Anytime Algorithm from Zilberstein [11]

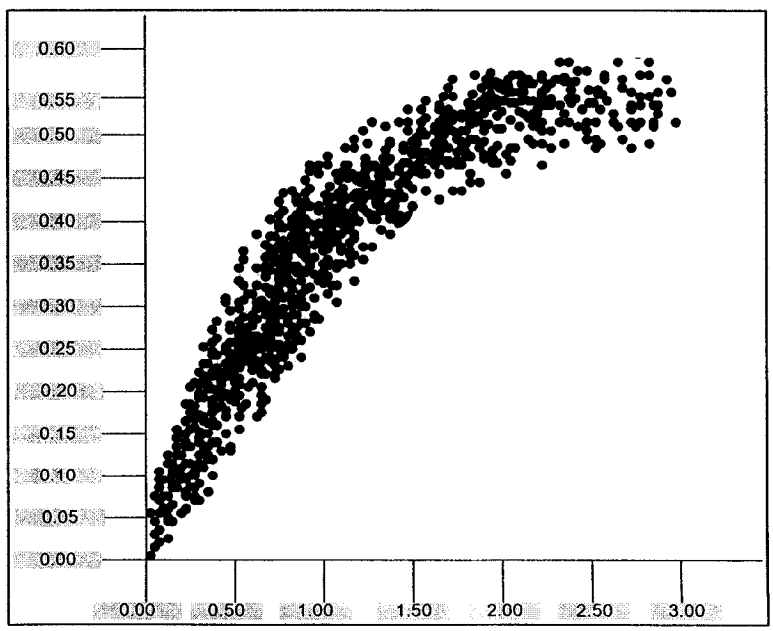

Figure 1-1 Runtime and quality results of this algorithm from Zilberstein [11]

Figure 1-3 provides an example from Decker et al. [14] of multiple methods for a sensor interpretation application. This figure shows two grammars describing characteristics of hypotheses necessary to identify a particular vehicle type. In the case on the left a number of intermediate objects are hypothesized from the low-level sensor data, then combined together to hypothesize a vehicle object. In the case on the right, the intermediate levels are bypassed and a vehicle is hypothesized based just on the low-level sensor data. This second method is able to hypothesize the vehicle more quickly, but with reduced certainty and precision.

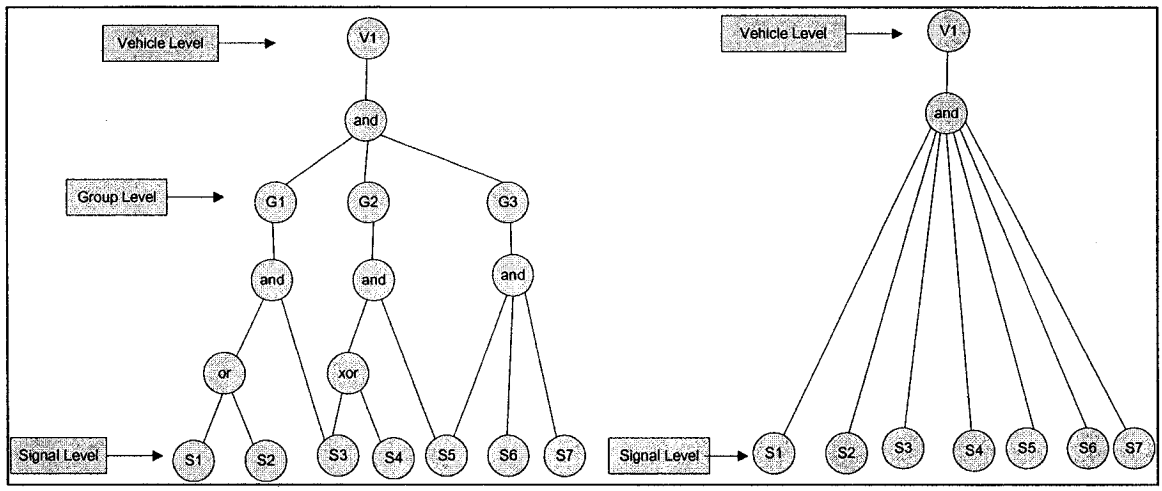

Figure 1-2 Multiple methods for a sensor interpretation application from Decker [14]

1.2.2.3 Summary

This section presents the key differences between the strategies that trade off quality with time in real-time AI applications. Also, it discusses the main drawbacks of these strategies when they are applied in developing new soft real-time applications.

Comparison of Different Strategies: In real-time AI systems [9], the focus is usually on

high-level goal achievement. For this reason, often in real-time AI, loose definitions of real-time are used. One common (and usually implicit) definition is that the system statistically (e.g., on average) achieves the required quality value by the required time, but no guarantee is made about any particular task in a lower granularity. Additionally, the anytime algorithm and approximate processing approaches have the characteristic that the system nearly always produces some quality value for tasks, although the value achieved may or may not be useful, given the criteria for high-level goal achievement.

Anytime algorithms have the advantage of always delivering some (feasible) result. So, in some sense, they are a naturally usable strategy. Another potential advantage of anytime algorithms is that a single version and hence the same piece of code is sufficient to handle the degradation required in the absence of adequate time or resources. On the other hand, the multiple method approach does not rely on the existence of an iterative refinement algorithm that produces incrementally improving solutions as the runtime increases. Instead, it involves different methods or solutions to handle the task at hand. These approaches should serve as alternative or complementary strategies, depending on the actual application needs.

Key Issues: Anytime algorithms assume that subsequent iterations produce incrementally better solutions. However, for most soft real-time applications working in an open environment, this assumption may be too strong. In some cases, it is difficult to find algorithms whose expected quality improves in a predictable, monotonic fashion in such systems. Another assumption that anytime algorithms rely on is that they can work

effectively in all environmental situations. This may not always be the case. Many of these applications face different and dynamic environments. It is hard to find one algorithm that works well in all kinds of environments.

The multiple methods approach is based on the assumption that the execution properties, such as duration, quality and dependence, associated with tasks are fairly predictable. In other words, this approach needs a complete and fine-grained task model. For many soft real-time agent-based systems in open environments, however, it is difficult to have complete knowledge of task properties during design time because these systems normally are large scale and subject to change at launch time. Moreover, this approach is based on a centralized task control mechanism that is responsible for global task planning. For a large-scale system with complex task structure and a great number of agents, the resulting performance may be compromised.

1.2.3 Real-Time AI System vs Traditional Real-Time System

The above-mentioned strategies from both real-time AI community and traditional real-time community are actually alike in terms of objectives, the applied principle and the methodology of performing quality-time tradeoff. All these strategies aim to develop real-time systems with strong capability to support the emerging generation of complex and dynamic applications with timing constraints. When more time and resources are available, the system produces higher quality results or services.

However, the detailed strategies from the real-time AI community are slightly

different from the ones from the traditional real-time community. Traditional real-time systems research has focused on developing low-level runtime system mechanisms to support predictable execution of traditional periodic control tasks that have only minor data dependencies. On the other hand, real-time AI systems research focused on more complex tasks that involve sophisticated dependencies. Normally, for real-time AI applications, the selection of suitable sub-tasks involves complex reasoning and planning algorithms. The difference between the characteristics of these sub-tasks can be dramatic. However, by simply loading different sets of modules of a task at run-time, traditional real-time applications have limited capabilities to change the characteristics of the task.

1.3 Contributions

This thesis proposes a self-planned multi-agent mission model (SPMM) involving multiple agents that integrate many key attributes in previous models and parallel computing in order to address the needs of soft real-time applications in an open distributed system. The following are intrinsically incorporated:

(i) An application consists of multiple types of missions that can be invoked periodically or sporadically. The expected deadline of a mission is known a priori.

(ii) Each mission consists of many phases; each phase has multiple solutions. A solution can be obtained by functional decomposition into a task graph or data decomposition involving parallel tasks. Functional decomposition leads to multiple possible solutions with differing runtime needs and resulting qualities.

Data decomposition can be performed at different granularities of data, leading also to quality-time tradeoff.

(iii) Admission control is exercised at mission invocation time.

Under SPMM, self-planned quality-time tradeoff is applied at every phase of a mission with the ultimate goal of meeting the soft deadline constraint of the mission. To facilitate application development, a corresponding self-planned agent framework (SPAF) is proposed and implemented. The framework consists of two parts: a soft real-time agent platform that provides key agent services, and a self-planner that performs quality-time tradeoff dynamically during the lifetime of a mission instance. A role-based agent behavior model [18, 19] is used in the framework. Through this combination, SPAF provides a complete solution to assist designers in developing application agent codes. The planner provided by the framework has an application interface that allows the developer to specify important attributes used by the planner in optimizing the quality-time tradeoff at runtime.

The rest of the thesis is organized as follows. Chapter 2 gives a detailed description of our soft real-time multi-agent model. Chapter 3 presents the design of our soft real-time multi-agent framework. Chapter 4 describes the usage of the framework through a sample application – Intelligent Security Monitoring System. In Chapter 5, we evaluate the performance and effectiveness of our framework and analyze the test results. Finally, in Chapter 6, we conclude our thesis with an indication of future researches.

Chapter 2

Self-Planned Multi-Agent Mission Model (SPMM)

This chapter introduces the details of our soft real-time Self-Planned Multi-Agent Mission Model (SPMM). A multi-agent system is characterized by the use of multiple types of agents playing different roles in order to fulfill the needs of their application-specific missions. Naturally agents interact in social groups throughout their lifetime. Various role models [18, 19] have been proposed in multi-agent design. Role design in agent-oriented software engineering bears resemblance to functional decomposition in parallel programming. By decomposing the application needs into roles and assigning them to agents, an abstract design can be concretized before passing it to the developer for agent code generation. Since our interests lie in open system applications in which the external environment provides the stimulus that triggers the launching of missions, periodically or sporadically, an application is modeled using missions each involving multiple agents at different phases. The model should be able to support varying degrees of complexity in agent dependencies within a phase and between phases.

Consider an intelligent security monitoring system. In this application, multiple security devices are installed in many places of a building. For example, surveillance cameras are installed in the lobby and in the hallways, image scanning devices are

installed in front of rooms for access control. All the data captured by the security devices are processed by the application on the server side. Periodically, the surveillance cameras will take the image of the scene and send the data to the server for image analysis. Additionally, the image scanning devices on the doors perform the image scanning and send the data, such as fingerprint image, to the server for personal identity validation. Due to the heavy computation work of image analysis, a single server can hardly finish the whole computation task on time, so a distributed multi-server system is needed to distribute and parallelize the computation of the task. The aforesaid features of a multi-agent system are desirable for the above application. So, a multi-agent system on the server side processes all the image analysis/validation requests. There are some interesting features to observe.

(i) In the system, each monitoring device is linked to a complex task (referred to a mission) on the server side. For example, a surveillance camera in the lobby is linked to an image analysis mission on the server side; an image scan device on a door is linked to a fingerprint validation mission on the server side. Because each device works independently and concurrently, each mission on the server side is also performed independently and concurrently.

(ii) The monitoring devices periodically capture the image information of the scene or sporadically accept a request from the user. So the related missions on the server side are invoked sporadically or periodically.

(iii) All these missions are time-bounded. For instance, the lobby monitoring mission

runs periodically, so an execution of the mission should be finished before the next period. Also, a person requesting a fingerprint validation should not wait for too long for passing through the door. So, the fingerprint mission should also be time-bound. However, missing a deadline will not cause system failure. So each mission has a soft deadline.

(iv) Multiple copies (instances) of the same mission may run on the server side. For instance, multiple copies of the fingerprint validation mission may be executed to serve multiple fingerprint validation requests from different doors.

(v) All the security devices can be used only after they are properly installed in the right place. So, their related missions can be invoked only after being created correctly.

(vi) For each mission, several phases exist during the execution. Completing all the phases accomplishes the mission. For example, the image-processing mission can include the following phases: image sampling, feature extraction, matching with database records, and decision resolution.

(vii) Each mission includes heavy computation work and a single agent can hardly finish the whole mission. Hence, each mission involves multiple agents playing different roles and collaborating in different phases (milestones). In other words, each mission is served by a group of agents that are distributed among the servers.

(viii) Multiple solvers (alternative solutions) exist for each phase. Certainly, image processing can be performed at different granularity. Feature extraction can

demand different levels of precision or features. Matching can be done with multiple classification techniques and voting schemes.

Aiming to develop a framework to support applications with features similar to the above, a self-planning strategy is incorporated in a mission to optimize quality-time tradeoff dynamically at runtime. Hence, a planner is invoked in the model for selecting an appropriate solver (set of agents) to perform the next phase depending on the remaining time and the available resources. We wish to use a generic planner rather than an application-specific one to manage/schedule the agents. So, the self-planned multi-agent mission model (SPMM) involves two essential components: the planner and applicant agents and their interface relationship. Each mission involves interactions between these two components. However, adding a generic planner into an application introduces an overhead to the application. A large overhead of the planner will weaken the ability of the application to meet the deadline rather than strengthen it.

So, in the development of a framework with the generic planner that uses quality-time tradeoff as a means to manage/schedule the agents to complete each mission, the following sub-problems remain as important focuses.

(i) A generic model for mission and alternative solutions is required for the planner to work and for the application designers to use.

(ii) The continuity of agents and mission state (consistency) in mission reconfiguration presents an original challenge. Due to the generic planner, additional agents and mission state switching is needed in managing the changes

of a plan. The effectiveness and efficiency with which this reconfiguration can take place affect the success of the design and hence the application.

(iii) The planner should be as thin a layer as possible. Hence a complex solution may not be desirable.

We explore the agent-programming model in the following sections and explain how the model handles the above sub-problems. In Section 2.1, we present the SPMM model and its components (application agents and planner agent). We explain the rationale behind a set of important modeling assumptions, the model of alternative solutions, and the quality model using a rank function. Section 2.2 presents the interface between an application agent and a planner agent and the rationale behind it. Finally, Section 2.3 compares SPMM with similar strategies.

2.1 Application Agents and Planner Agent

This section presents the modeling of an application involving missions and agents under the control of a generic planner agent. Upon invocation, a multi-agent mission is launched. The completion of the mission is under the management of a generic planner agent that aims to optimize quality-time tradeoff, involving possibly different choices of tasks or granularity of data processing.

2.1.1 Application and Mission

Each system contains two types of agents: application agents and a planner agent. Unlike the planner agent, application agents are application specific. Each application agent is

characterized by its capability in playing the roles that it has been assigned. Once created, an application agent remains available to support successive launches of a mission until the application terminates. Hence a multi-agent application system contains a pool of application agents necessary to support the different types of missions required. Launches of a mission can be periodic or sporadic, as described earlier. A planner agent performs the self-planning required to optimize the quality-time tradeoff in each phase of a mission. It can be designed once and used for many applications. The following assumptions are made regarding a mission:

(i) *Mission deadline is known a priori; a mission consists of phases, each with an approximate deadline that can be dynamically adjusted at runtime by the planner.*

In reality, software is designed to mimic human strategies and behaviors. Hence, in this thesis, the generic planner mimics what a humanized solution would do in handling time. Let us consider an example in the real world. A group of software developers have to finish a project in four months. How can the team leader manage the progress of the project? A typical scenario is like this. At the beginning, several important phases with idealized deadlines are defined for the project and each developer is assigned different roles in different phases. For example, divide the development of the project into four phases: Architecture design (3 weeks), Detailed design (4 weeks), Implementation (5 weeks), and Testing (4 weeks). The time in parenthesis is the idealized time for each phase. During the development of the project, adjust the deadline of each phase based on

the real progress of the project. For instance, when the time taken in the detailed design phase is longer than expected, in order to complete the project within the deadline, what the team leader can do is to finish the rest of the phase faster than planned and compensate for the lost time by using more efficient means to do the implementation and testing. So, in order to mimic what a humanized solution would do in handling time, a generic planner needs to have model-based mission (task model) with time estimates and have a rough plan about how much time is allowed in each phase (in ideal situations). With estimated time, the planner can manage the actual time of a mission within the targeted idealized schedules. With time estimates on both phases and involved tasks in each phase (performed by each agent), the detection of phase delays and adaptation to future phases can be performed rather easily with proper design support. This certainly facilitates one of our design objectives: make the self-planner as thin as possible. Moreover, in the applications such as the abovementioned security monitoring system, the time for each phase and the involved tasks can possibly be estimated rather accurately with a given runtime environment, at system set-up time. These estimates can be readjusted whenever there are changes in the runtime system and/or environment.

(ii) *Importance of mission deadline is quantifiable using a ranking function.*

Different missions serve different purposes. Importance of mission deadlines can be ranked based on the inherent application requirements associated with these missions. For example, in the security monitoring system, suppose room A

is more important than room B. Then the importance of the fingerprint validation mission for room A should be ranked higher than that for room B.

(iii) *Concurrent missions are independent. Delay in one does not affect the other.*

Two missions launched concurrently are assumed to be logically independent and do not require direct cooperation between them, despite they may share some logical or physical resources. If they have potential resource conflicts, the planner may try to reduce the conflicts by choosing different solvers with different resource requirements. Obviously, the process of choosing solvers will have some impact on mission executions and deadlines. For example, it may decrease the overall quality of the mission. Also, the planner may fail to find suitable solvers and eventually the mission may fail to meet the deadline. However, our model does not make absolute guarantees on mission deadlines. One implicit criterion is that the system will statistically (e.g., on average) achieve the required quality value by the required time, but no guarantee is made about any particular mission.

(iv) *Multiple solvers exist for a phase, with known resource (CPU, memory and communication) requirements.*

However, the model does not assume that every phase can have multiple solutions whose resource requirements are all known a priori. The model allows some phase to have only one solution as long as not every phase has only one solution. If the phase with one solution is delayed, it is possible that the planner can tolerate the delay in the following phases by choosing simpler solutions with

smaller execution times and make the mission meet the deadline eventually. It is not necessary that all the resource requirements of a solution are known a priori. The more knowledge about the resource requirements the planner has, the better the planner can pick a suitable solution based on the runtime environment.

2.1.2 Alternative Solution

We use alternative solutions with different qualities and execution times to optimize quality-time tradeoff in each phase of a mission. The ability of our strategy to tolerate time failure of a mission depends on how many alternative solutions the mission has and how well these solutions are defined. In other words, the more alternative solutions a mission has and the more diversified these solutions are, the stronger ability our strategy has to tolerate deadline missing failure of the mission.

One obvious strategy is that alternative solutions refer to different algorithms. Based on their own domain knowledge, developers define different solutions using alternative algorithms for each task. For example, for a face identification task in an intelligent security monitoring system, there can be various face recognition algorithms, such as LDA (Linear Discriminant Analysis), Bayesian Classifier, Gabor Wavelet Algorithm and Elastic graphs, etc. Different algorithms can be used to accomplish the task. However, this approach may have a major drawback. Sometimes, multiple appropriate algorithms may not be available, and even if they are available, their quality-time tradeoff may not be apparent or deterministic. In such cases, we use a data decomposition strategy [20] as

a means to parallelize the work to meet deadline requirements. Using the same algorithm, multiple agents can work on different parts of the data. The number of agents and the granularity of the data provide the planner with a means to optimize system response. Indeed, the strategy of using multiple solutions is related to functional decomposition in traditional parallel programming. Upon functional decomposition, each function can be solved using a set of potential solvers, with varying degrees of performance implications. The following is the detailed description of our model of alternative solutions.

All the solutions can be divided into two possibilities.

(i) *Parallelization by distribution of work to a set of identical agents (Category A).*

This follows from the traditional strategy of data decomposition whereby the computational workload is partitioned among a set of k agents. These agents need little coordination during the computation. In such a case, when time is running short, there can be two methods to save time: creating more agents; or requiring each agent to perform less work (data granularity is increased). When the available resource is small, creating more agents will not always perform well. So, we emphasize on the second method. In other words, we emphasize on granularity, not concurrency. So, each agent is given smaller data size in order to complete its work faster. Data reduction is achieved by compressing data into a coarser granularity.

(ii) *Parallelization by function (Category B).*

This follows the other traditional strategy of functional decomposition

whereby the overall computation is partitioned into a corresponding set of functions, each assigned to a distinct agent. Typically these agents interact in order to complete the overall work. In such a case, the different algorithms may lead to different solutions derived from functional decomposition. These solutions may differ in resource needs, and in quality of results. Again, the choice of a solution may be made at runtime depending on the available resources and time.

2.1.3 Quality and its Ranking

The term "quality" is defined for this thesis as the variance between the returned result and the optimal result for a given task. This is a very task-specific definition. For example, in a real-time information retrieval system (for example, when doing a search on Google), an optimal result can be defined as a link having 95% or higher accuracy ratio. These optimal results are very specific to the requests and reject universal classification. Either the developer or system administrator must define them. With the notion of a pre-defined "optimal result" for a task, the variance between the actual result and the optimal result can be used as an objective metric to statistically compare the qualities of different solutions.

Quality ranking is to assign different numbers to different solutions to indicate their difference in the quality of results. Such a ranking can only happen within a phase and is performed by developers or system administrators during the design time. The solutions in different phases cannot be ranked together. The quality ranking of a returned result can

be a constant value Q or a quality ranking function $Q(x)$ (x refers to the quality ranking of the optimal result). For instance, in the above information search task, the optimal result that has a 95% accuracy ratio can be assigned a value of 10 (this actual value is defined by the developer). Then a search result that has a 75% accuracy ratio can be assigned a value of 7.9 using the function $Q(10) = ((95\% - 75\%) / 95\% * 10)$. The total quality ranking of a mission is represented by the sum of the quality rankings of all selected solutions that were completed successfully in their phases.

2.1.4 Planner

The planner is responsible for selecting solutions based on its knowledge of the runtime system and the expected needs of the mission. The former is runtime knowledge while the latter is based on design time knowledge. The planner is generic and mission-independent. It is an integral part of the solution provided to a mission. Under the assumption that missions are independent, so are their associated planners. The planner has knowledge of the multiple solvers available for each phase of the mission, the relevant runtime parameters including the elapsed time, the deadline constraints and available system resources, and the dependency between agents in successive phases of a mission. Based on a generic model interface, the planner can be activated upon the launching of a mission to guide the mission to proceed as safely as possible without compromising quality unnecessarily.

Figure 2-1 shows the interactions between a periodic mission, its planner and the

application agents executing the phases of the mission. In the figure, 'launch' refers to an invocation of the mission. Upon initialization, the system starts with a pool of application agents and a planner agent for each type of mission. Successive launches of a specific mission involve the associated planner and application agents. A mission migration into a new phase is represented by the selection of the application agents by the planner. An application agent returns to a dormant state when its assignment is completed. On the other hand, migration to a new phase is synchronized by the successful completion of all application agents involved in the current phase. Subject to the actual time experienced, the solvers for the next phase will be chosen by taking quality and remaining time into consideration.

2.2 Interface between the Application and the Planner

Through the interface, application agents and planner agent cooperate to complete a mission. However, the planner is not application specific. Intrinsic to the planning model is a generic interface between the planner and the application so that all relevant knowledge about the solvers and application agents are specified for each application design. This interface contains a generic set of parameters that are detailed in this sub-section.

2.2.1 Design Interface

Through the following design interface, an application provides static mission information to the planner upon mission initialization.

(i) An application involves a set of periodic and/or sporadic missions. A *mission* has three properties: *deadline*, *period* and *importance ranking*.

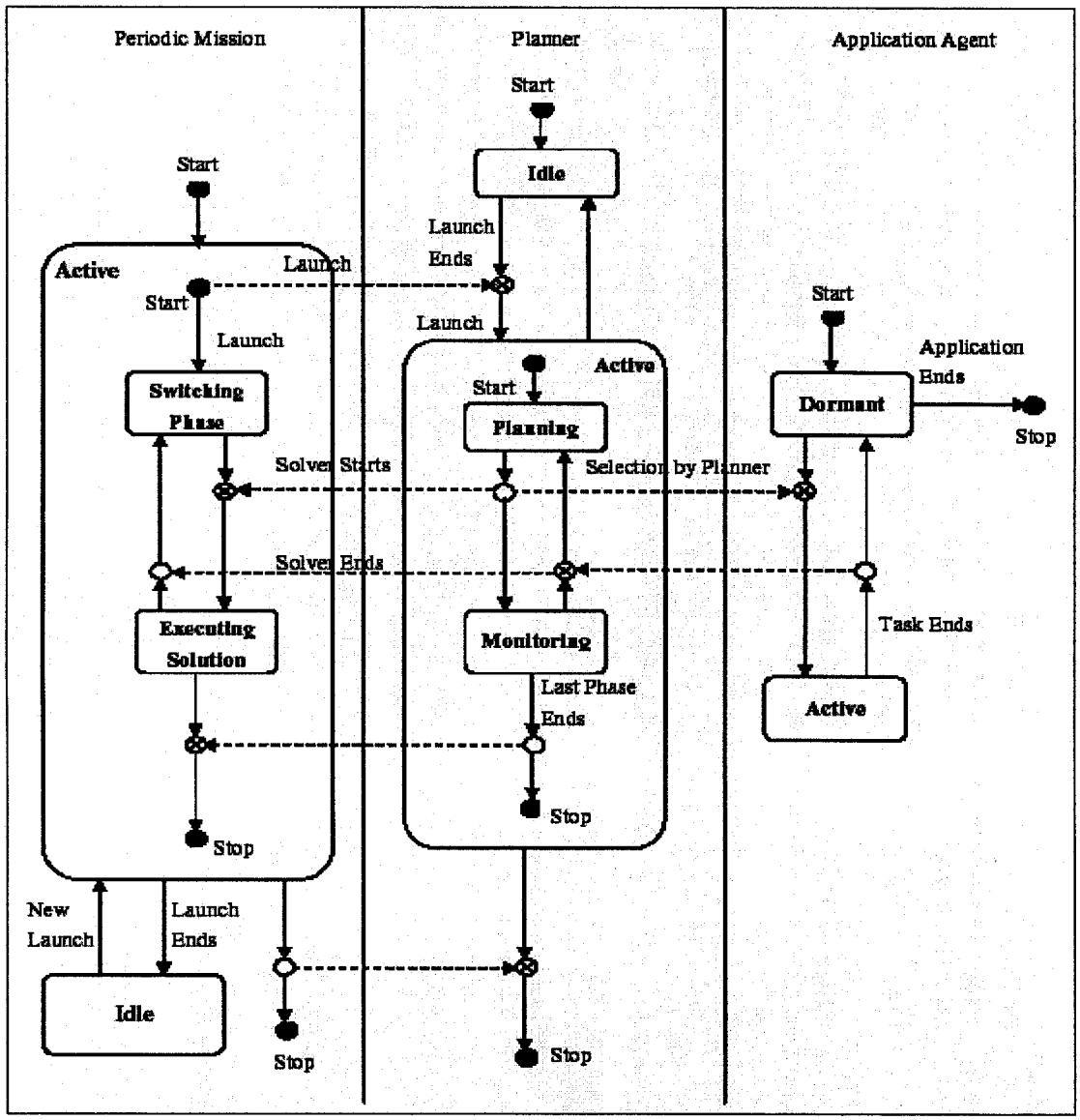

Figure 2-1 State Diagram of the Self-Planned Multi-Agent Model

(ii) A mission is composed of a sequence of *phases*, each involving several application agents working cooperatively to solve a sub-problem, referred to as a *task*.

(iii) Each task has multiple alternative *solutions*; completing any of them will finish the task. The system has a set of recognizable resources including: *Host capacity*; *Memory capacity*; and *Communication bandwidth*. Each solution belongs to one of two categories: parallelization by distribution of work (Category A); parallelization by function (Category B).

Parameters for a solution in Category A:

- *Agent name* and *relevant invocation parameters*, including an explicit *data size* parameter
- *Recognizable resource needs*
- *Expected time function* (if it is not constant)
- *Quality ranking function* (if it is not constant)

Parameters for a solution in category B:

- *Agent names* and *relevant invocation parameters*
- *Recognizable resource needs*
- *Expected time*
- *Ranking of quality of results*

2.2.2 Runtime Interface

The runtime interface exists during mission switching from one phase to the next. As we mentioned before, the performance of the runtime interface directly affects the overhead of the planner. So, the goal of this interface design is to make mission switching as fast as

possible. In addition, results derived from a phase are needed in a future phase. This forwarding of knowledge can be based on the continued survival of the agents or passive objects shared between phases.

Survival of agent through phases means two possible solutions: the next phase choice simply involves selection of data to be given to that agent or the agent is an artificial entity that can be assigned actual work (code) to be executed. The former places some design constraints on the application, while the latter requires the agent to behave like a kernel thread that can be loaded to execute an application thread. However, the application agents in the former solution only have to pick up the data instead of picking up data and code in the latter solution. So, the application agents in the former solution may switch faster and may have residual context that is useful to a mission.

Passive objects allow agents to pick up new state information and be launched. Such agents carry the (algorithmic) knowledge and may be available (in standby mode). When work is available, the planner assigns it to the agent(s). This kind of solution needs some sort of centralized control for these passive objects, such as using a repository manager. Nevertheless, in some situations with heavy message passing during phase switching, this solution can cause a serious performance bottleneck.

Based on the above analysis, the following assumptions are made regarding the runtime interface between application and planner.

(i) *A set of application agents will survive through two successive phases, which are responsible to forward mission status to next phases.*

(ii) *The surviving agents are decided by the application developers at the design time.*

(iii) *An application agent contains the code for all the roles that it will perform. Based on the parameters received from planner, it simply performs the role reflected by the parameters.*

Upon completion of a phase, an application agent has to synchronize with the planner (by reporting and waiting for further details for the next phase). Two possibilities may occur:

(i) The agent has a few possible next phase entry points, selected by the planner. Each entry point is associated with the details of the roles that agent is going to play. Two different entry points may differ only in terms of the system scale (such as the number of other agents working with it). They may also differ in actual code (role) being played.

(ii) The agent released by the planner returns to a dormant state waiting for the next mission call.

2.3 Comparison with Other Strategies Using the Same Principle

This section compares our solution against those cited in Chapter 1 and analyzes their similarities and differences.

2.3.1 Similarities

SPMM involves on-line planning for tasks, using alternative solutions with different quality to trade off mission time. The planning decisions depend on the estimated task execution times. In comparison, these features are somewhat similar to on-line QoS

negotiation, except that this is done within a mission and is self-induced. In addition, SPMM employs on-line admission control to further enhance its performance. On the other hand, our strategy involves the uses of phases. A mission can be divided into a sequence of phases. Finishing all the phases accomplishes the mission. Within each phase, a choice of solutions involving either data parallelism or functional parallelism is usually available. This is quite different from the anytime algorithm approach but bears resemblance to the multiple approximation approach used in AI applications.

2.3.2 Differences

Although our strategy is similar to some existing approaches as mentioned in the previous sub-section, it is also significantly different from these strategies when details are examined. SPMM differs from the previously mentioned strategies in two aspects:

(i) SPMM is an agent-based model, so it naturally inherits the advantages of agent-based models in handling distributed applications in an open environment. For example, the intelligence of agents allows for a greater level of decentralization; this is a desirable characteristic in a distributed environment. In addition, agent-based designs are well structured and the mission model is inherent in such applications, unlike a simple process- or thread-based model. Interactions among related agents until the mission is accomplished provide a simple mechanism to model the decomposition involved in parallel system design.

(ii) The task planning in SPMM focuses on the activities of a group of agents. In other words, it focuses on planning for a group of threads. However, the strategies from the real-time community mainly pay more attention to the control of a single thread. Such a feature means that these strategies have difficult in handling the soft real-time applications that normally involve highly distributed tasks with complex structure and dependency.

Additionally, SPMM differs from the anytime algorithms in two aspects:

(i) Instead of assuming the existence of iterative refinement algorithms, SPMM assumes that a mission can be separated into a sequence of phases. By incorporating this key assumption, we can proceed to handle problems where anytime algorithms do not exist.

(ii) SPMM does not assume that one algorithm can work well in all situations (another key assumption of anytime algorithms) because this is not always the case for an application in an open environment. SPMM supports different algorithms for different run-time environments.

In comparison, there are two main differences between SPMM and the multiple method approach in real-time AI community:

(i) SPMM does not rely on a complete task model of the system. SPMM performs the task planning only based on estimated time, quality and resource requirements of a solution and does not consider the detailed agent activities within this solution. Compared with detailed and complete model of agent activities in

multiple methods, SPMM is more flexible to be used in application development.

(ii) SPMM integrates the use of both functional and data decomposition to support alternative solutions in a mission. Different algorithmic solutions represent different functional decompositions in solving the sub-problem. Differences in data decomposition may involve different degrees of concurrency (hence speed) or differences in data granularity (and hence speed as well). The former involves differences in agent codes and the latter involves only differences in data sets.

In conclusion, by integrating mainstream technologies from real-time systems, real-time AI applications and parallel computing, SPMM presents a complete model for developers to better develop soft real-time multi-agent systems by trading off quality with time. An effective marriage of these techniques is aimed to overcome the shortcomings of each individual technique.

Chapter 3

Design of Self-Planned Multi-Agent Framework

This chapter presents the Soft Real-Time Self-Planned Multi-Agent Framework (SPAF), which is an implementation of SPMM and provides an application development environment where the planning part is built in the platform. We choose Java as the programming language of our platform because of its portability, ease of use, security and popularity. Real-time applications usually need to control the system resources, such as memory, hardware ports, software/hardware interrupts, etc. Also, real-time applications need precise control of system time and execution of threads. However, the standard Java Virtual Machine (JVM) has weak support for these features. With standard Java, the programmer cannot control the memory, access the hardware ports or accept interrupts from the operating system. Additionally, the garbage collection mechanism that is beyond the control of programmers makes the execution time of Java program unpredictable, a drawback that cannot be accepted by real-time application developers. Moreover, the scheduling of a Java thread in JVM is not truly priority-based. In other words, JVM does not guarantee that a thread with the highest priority is always run first. With these drawbacks, the standard Java is not suitable for our development. In order to have the advantages of Java and at the same time overcome its disadvantages, we adopt real-time Java [21] in the implementation of our platform because it has the following features

compared with standard Java:

Feature	Description
Accessing hardware	Real-Time Java allows byte level access to physical memory, mapping objects into RAM or flash memory, dealing with virtual memory, and implementing device drivers, memory-mapped I/O, and low-level software.
Scheduling	The RealtimeThread class extends the semantics of the Thread class for real-time. An instance of the Scheduler class implements a scheduling algorithm, providing flexibility to install an arbitrary scheduler that requires at least 28 priorities. Base scheduling is preemptive and priority-based.
Synchronization	The semantics of the synchronized keyword has been extended to avoid priority inversion, and waiting queues are priority ordered.
Asynchronous event handling	Real-Time Java provides two classes: AsyncEvent and AsyncEventHandler to treat events. An AsyncEvent object is like a POSIX signal or a hardware interrupt, and can have a set of handlers associated. An AsyncEventHandler has a SchedulingParameters object associated, which controls its execution.
Resource management	In Real-Time Java, prior to starting a task, MemoryParameters must be assigned to it. These parameters are used for both the scheduler (to control admission) and the Garbage Collector (to satisfy all tasks allocation rates). The scheduler can generate an exception to reject workloads, when a task exceeds a resource limitation, or when a task allocates memory faster than the garbage collection budget allows.
Dynamic memory management	Real-Time Java supports scoped (objects have a limited lifetime), physical (objects in faster memory) and immortal (objects are not collected) memory. The GarbageCollector class provides methods for getting information about the Garbage Collector behavior (e.g., getOverhead(), getReclamationRate(), and getPreemtionLatency()).

Table 3-1 Features of Real-Time JVM

To develop our soft real-time self-planned agent framework, we need an agent platform that can support key agent services such as agent lifecycle management and naming service, and that provides a transparent agent communication service. So far, however, there are no such platforms that can run on real-time JVM. So, the design of

SPAF involves the design of the agent management platform that can work on real-time JVM; and the design of soft real-time self-planned agent framework above the agent management platform.

3.1 Agent Management Platform

Our agent management system for agent lifecycle management, naming services and agent communication follows the FIPA standard [22] and is not much different from other agent platforms such as Jade [23], except for the communication sub-system.

3.1.1 Architecture

Our agent platform implements the basic agent management model, includes the Agent Management Service (AMS) that manages the lifecycle of agents and agent naming service and offers a Message Transport Service (MTS) supporting transparent communication between agents. All agent communication is performed through message passing, where FIPA ACL is the language to represent messages. The design of the agent management architecture is similar to that of Jade except for the communication sub-system. Jade adopts Java RMI as the main communication approach between two hosts. However, real-time JVM only supports socket communication, so we redesign the communication sub-system to be suitable for our objectives.

The software architecture is based on the coexistence of several JVMs and the communication relies on Java Sockets between different VMs and event signaling within a single VM. Each VM is a basic container of agents that provides a complete run time

environment for agent execution and allows several agents to concurrently execute on the same host. The platform uses default real-time JVM. In other words, the platform does not include any modification of the JVM. In principle, the architecture also allows several VMs to be executed on the same host; however, this is discouraged because of the increase in overhead and the lack of any benefit. Each agent container is a multithreaded execution environment composed of one thread for every agent plus system threads spawned by the Message Deliverer for message dispatching. The main container runs management agents and represents the whole platform to the outside world.

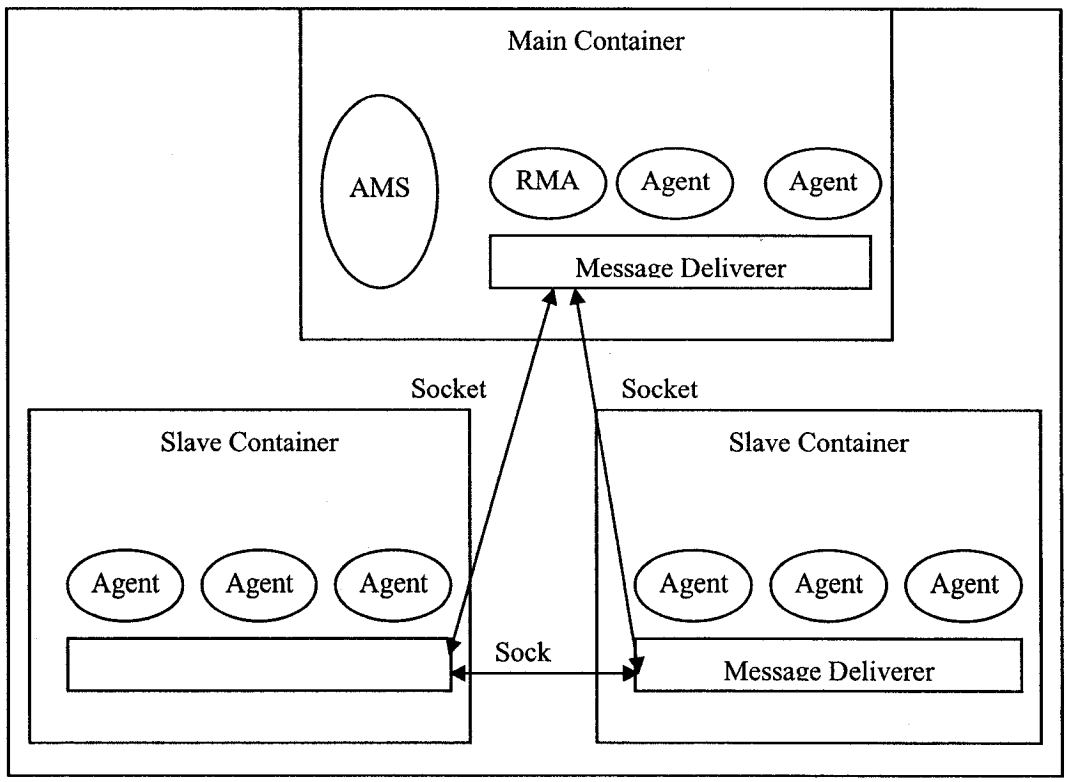

Figure 3-1 Architecture of Agent Management Platform

A complete agent platform is then composed of several agent containers as shown in Figure 3-1. Distribution of containers across a computer network is allowed. Each agent

container controls the life cycle of agents by creating, suspending, resuming and killing them. Besides, it deals with all the communication aspects by dispatching incoming ACL messages, routing them according to the destination field (receiver) and putting them into private agent message queues. For outgoing messages, the Agent Container maintains enough information to look up receiver agent location and choose a suitable transport to forward the ACL message. The agent platform provides a User Interface (UI) for the remote management, monitoring and controlling the status of agents, allowing, for example, to stop and kill agents. The UI itself has been implemented as an agent called RMA (Remote Monitoring Agent). All the communication between agents and this UI and all the communication between this UI and the AMS is done through ACL. Figure 3-2 shows this User Interface of our platform.

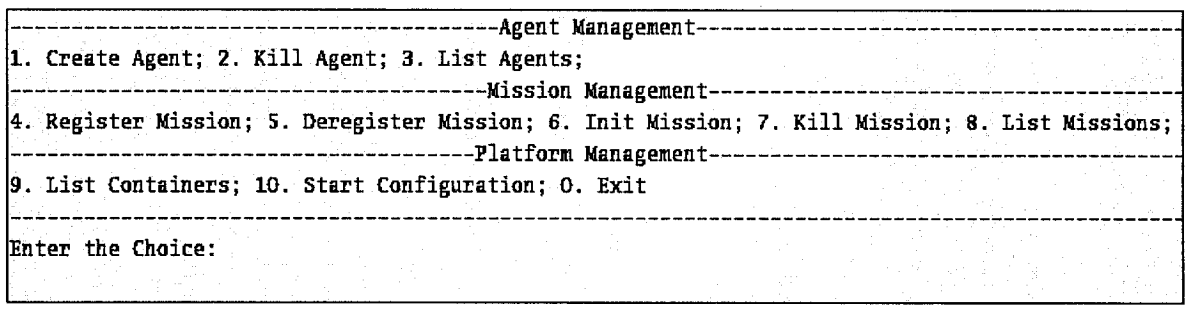

Figure 3-2 User Interface of the Agent Platform

3.1.2 Communication sub-system

The main container maintains a table of all containers along with their host names and ports. When a new main container begins to execute, it creates a Message Deliverer (a system thread) listening to a user specified TCP/IP port; then it starts an AMS agent and

an RMA agent. When a new container begins to execute, it creates a socket connection with the main container based on the main container's host name and port. Through this socket link, the new agent container registers itself with the main container and is added to the Agent Container Table. This new container notifies its main container whenever an agent is created or terminates, in order to keep the agent global descriptor table consistent. The message dispatching between the sender agent and receiver agent are through their own agent proxy. An agent proxy keeps a reference of the receiver agent object if the receiver and the sender are in the same host or keeps the host name and port of the remote container where the receiver is located if they are in different hosts. For the application, the communication between two agents is based on their name only, in other words, without considering on which host they are located.

As mentioned in Chapter 2, the generic planner manages the changes to the mission future. However, additional message passing between application agents and planner introduces a large overhead to task planning. Minimization of this overhead is important to the real-time agent platform. Hence, the Message Deliverer in each container caches the socket connection to other containers once a message is sent to them. The destination container creates a system thread to handle and maintain this connection. So, unlike RMI, this mechanism avoids the creation of a socket connection every time a message must be delivered, which is a very time-consuming operation.

When an agent sends a message, the following different cases are possible.

(i) If the receiver agent lives in the same agent container, the Java object

representing the ACL message is passed to the receiver using an event object, without any message translation (for example, the message sent by Agent1 to Agent2 in Figure 3-3).

(ii) If the receiver agent lives on a different container, the ACL message is sent using a socket by the Message Deliverer (for example, the message sent by Agent3 to Agent2 in Figure 3-3; in this case, a cache hit is supposed to occur, so the main container is not contacted).

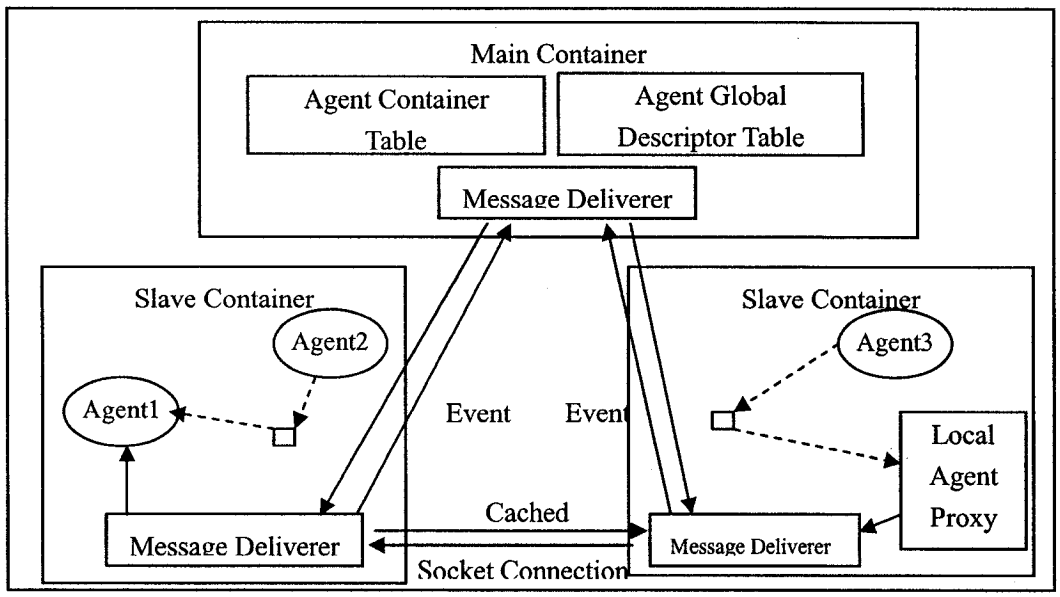

Figure 3-3 Intra-platform Communication Sub-system

3.2 Soft Real-Time Self-Planned Multi-Agent Framework

The agent management platform provides basic agent management service and communication service. Moreover, in order to create a complete real-time multi-agent framework, we should design a real-time agent framework above this platform. This framework supports mission management (mission lifecycle management, mission

admission control and mission time consistency checking) and mission execution management (task planning and task execution monitoring).

Within the framework, each mission instance has its own task planner in order to avoid the task-planning bottleneck and reduce task planning overhead. The manager agent equipped with task planner and application agents of each mission instance form an agent group. After a mission instance is started, the communication between agents only happens within the agent group (manager and selected application agents). In other words, no service agents are involved in mission execution because service agents shared by all agents can easily become the performance bottleneck of task planning. The different instances of a mission do not share the same application agents. They involve different instances of the same application agents. This is to avoid planning conflict raised by two task planners. Creating agents during task execution is time consuming, so we assume that all the application agents are created and started before the mission instance is started. The importance value of a mission is automatically mapped as the priority of the real-time threads of its agents that are involved in this mission by the middleware.

This framework also provides a resource-monitoring service for mission admission controlling, for task planning and for locating agents during the creation of these agents. This service periodically collects and calculates the resource status data of each host and updates the local resource-monitoring file on each host. In other words, each host has a copy of the current resource status data of all the hosts in the platform. This file is accessible to all the task planners residing in the same host. The reason for separating

resource data collection from task planners is that the remote resource data collection and calculation are relatively time-consuming. If the task planner performs this work every time when it needs resource data, its planning overhead will increase dramatically.

The design of this framework involves four aspects: real-time service agents, a set of protocols used by service agents to perform mission management and task planning, general task planner, and general application agent.

3.2.1 Real-Time Service Agents

All the services that the framework provides are designed as service agents. They include the following three Agents.

(i) *Mission Management Agent (MMA):* The MMA has the following functions: register/deregister mission, init/kill mission instance and mission time consistency checking.

(ii) *Manager Agent Equipped with Task Planner:* The Manager Agent performs mission execution management and task planning, including mission admission control, starting mission instance, choosing a suitable solution for a phase, starting the selected solution and handling deadline-miss exceptions.

(iii) *Snmpclient Agent:* Each container has an Snmpclient that periodically collects and calculates resource status data (e.g., CPU load, memory and bandwidth) of all hosts and updates a local resource file with these data.

All the agents are implemented as real-time threads. So, they can take advantage of

the features of a real-time JVM. The following class diagram (Figure 3-4) shows the design of the agents (service and application agents) in this platform (agent management platform and real-time agent framework).

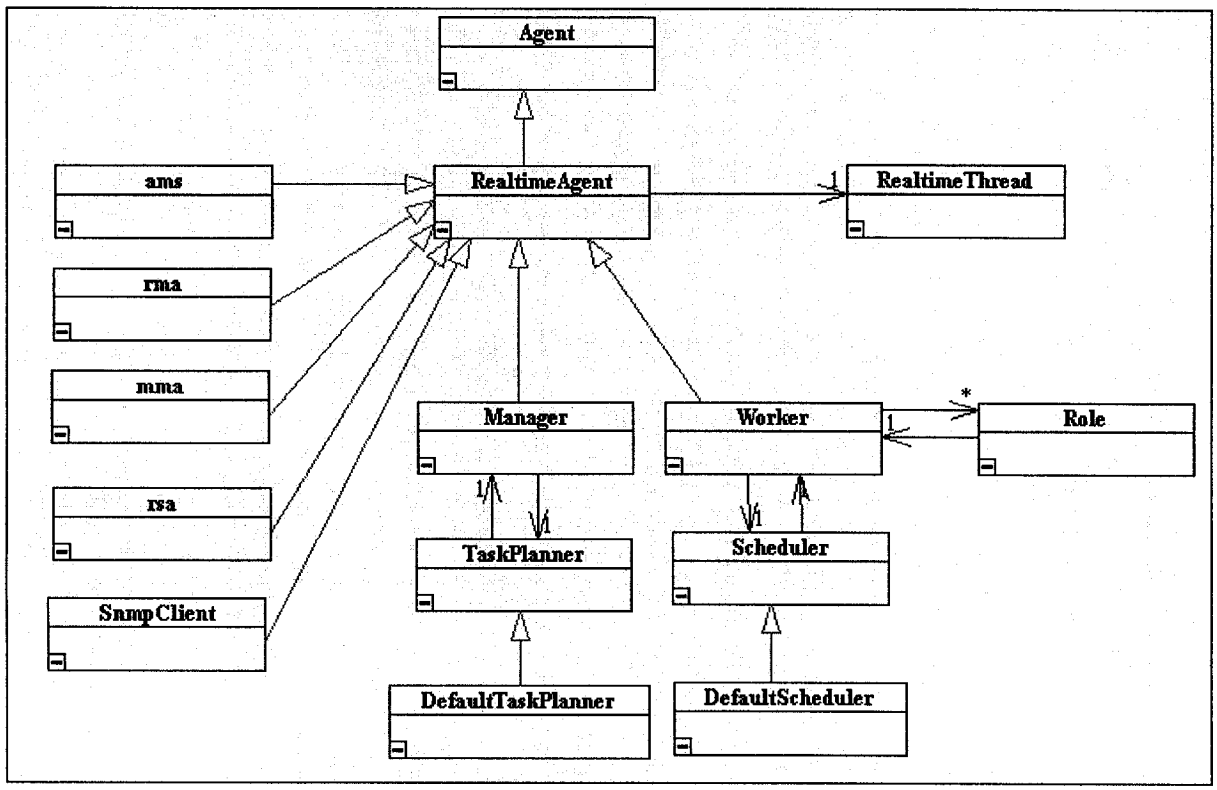

Figure 3-4 Class Diagram of Agents

3.2.2 Task Planner

As an internal component of a Manager agent, the task planner performs task planning for each phase of a mission and handles deadline-missing exceptions.

3.2.2.1 States of the Planner

The state diagram of the planner agent is shown below.

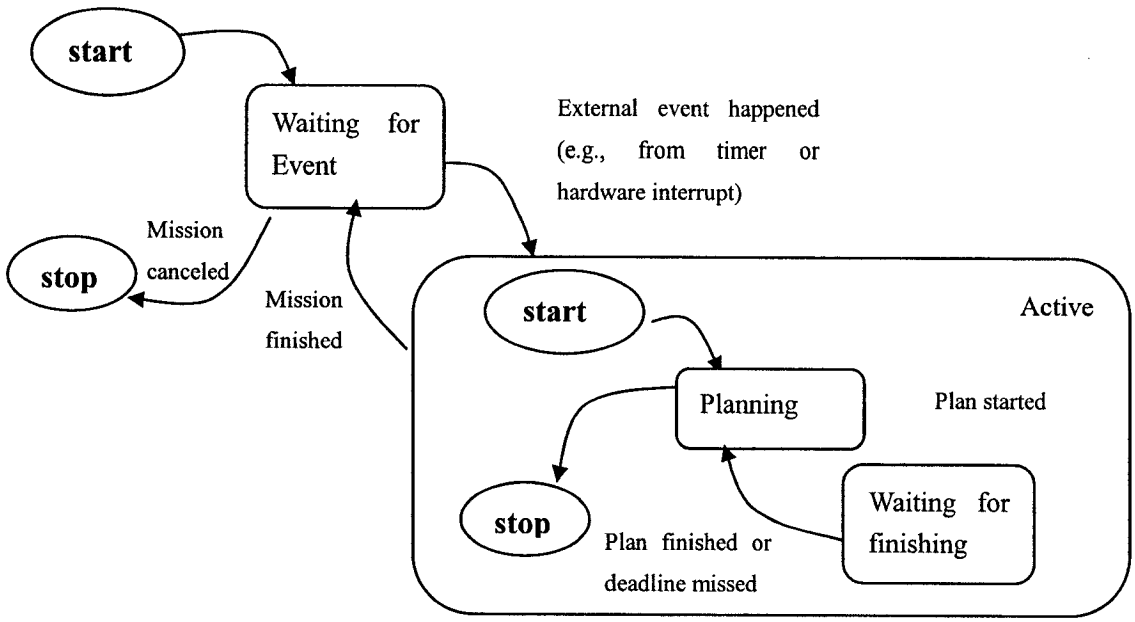

Figure 3-5 State Diagram of Task Planner

3.2.2.2 Input for the planner

(i) Static Input

The following mission information is provided by the developer and submitted to the planner before the mission is started.

Object Name	Attribute	Description
Mission	deadline	Indicate the deadline of a mission.
	period	Indicate the interval between two periodic events.
	importance	Identifies the importance of a mission within its application (range from 1 to 10).
	phaseSequence	A sequence of phase objects.
Phase	index	The sequential number of a phase within its mission.
	deadline	The deadline of a phase.
	MET	Minimum execution time of a phase.

	solutionSet	A set of solutions.
Solution	agentNameSet	A set of agent names and relevant invocation parameters.
	expectedTime	The expected execution time of this solution, which can be a constant or a function of data size parameter.
	quality	The expected quality of the result, which can be a constant or a function of data size parameter.
	resourceNeeds	A set of resource needs.
AgentName	agentName	The name of an agent.
	invocationParameters	The invocation parameters for the agent.
InvocationParameters	roleName	The name of the role that an agent will play.
	relatedRole	The related role parameters.
	sizeParameterSet	data size control parameters (optional).
	others	User-defined parameters (optional).
ResourceNeeds	hosts	The maximal value of the CPU utilization of some hosts. The solution will be selected only if the real CPU utilization is under this value.
	memory	The required remaining memory capacity of some hosts. The solution will be selected only if the real remaining memory is greater than this value.
	bandwidth	The required bandwidth of some hosts. The solution will be selected only if the real bandwidth is greater than this value.

Table 3-2 Static Input Parameters for Task Planner

(ii) Dynamic Input

Provided by the application: the mission status at the end of a phase. After finishing its work in one phase, each agent reports to the planner whether the task is completed or not, and the names of the roles that it can play in the next phase.

Obtained by the planner: the elapsed time of the mission, and the current system resource status (CPU load, memory or bandwidth)

3.2.2.3 Planning algorithm

```
Public Plan plan(Mission mission, Phase currentPhase, float elapsedTime, Vector systemLoad,
                 Vector potentialRoles )
{
    // if the deadline of current phase is missed, choose the cheapest plan according to
    // potential roles and try to finish it as soon as possible
    if (currentPhase.deadline <= elapsedTime) then {
        //choose the cheapest plan
        Plan selectedPlan = getCheapestPlan(currentPhase.solutions, potentialRoles);
        //if the plan was already selected, return null
        If(selectedPlan == null || selectedPlan == currentPlan) then
            Return null;
        else{
            currentPlan = selectedPlan;
            return currentPlan;
        }
    }
    else { //if the deadline is not missed.
        //calculate the remaining time of the mission.
        remainingTime = mission.deadline – elapsedTime;
        //using proata reasoning, get the appropriate time for the current phase
       assignedTime = getProataTime(remainingTime, currentPhase, mission)
       //if the assigned time is not greater than the Minimum Execution Time of the current
       //phase, choose the cheapest plan according to the potential roles.
```

```
if( assignedTime <= currentPhase.MET) then {
    Plan selectedPlan = getCheapestPlan(currentPhase.solutions, potentialRoles);
    if(selectedPlan == null || selectedPlan == currentPlan) then
      Return null;
    else{
        currentPlan = selectedPlan;
        return currentPlan;
      } }
 else{
    //if the assigned time is greater than the Minimum Execution Time of the current
    //phase, choose the appropriate plan according to the potential roles, time, quality
    //and available system resource
    //choose all the solutions involving potential roles that reported by the application
    //agents (Solution Set 1)
    Set selectedPlanSet = getPlansInvovlingPotentialRoles(currentPhase.solutions,
    potentialRoles)
    //from Solution Set 1, choose the solutions whose expected execution time is not
    //greater than the assigned time (Solution Set 2)
    selectedPlanSet = getPlanswithLimitedTime(selectedPlanSet, assignedTime);
    //from Solution Set 2, choose the solutions whose resource needs is satisfied by the
    //current system resource status (Solution Set 3)
    selectedPlanSet = adjustPlansAgainstResource(selectedPlanSet, systemLoad);
    //from Solution Set 3, choose the solution with the highest quality
    While (!selectedPlanSet.isEmpty())
        Plan selectedPlan = getPlanwithHighestQuality(selectedPlanSet);
        if (selectedPlan == currentPlan) then {
            SelectedPlanSet.remove(selectedPlan);
        else {
            currentPlan = selectedPlan;
            return currentPlan;
        }
    }
    // No plan is selected
    Return null;
} } }
```

Script 3-1 Planning Algorithm

3.2.2.4 Output from the planner

The output from the planner can be null, which means no solution can be selected, or it can be a solution for the current phase including a set of agent names and the invocation parameters (the name of role that the agent will play, role allying parameters, data size control parameters if the solution belongs to Category A, and other user-defined parameters) for each agent.

3.2.3 General Application Agent

A general application agent is a partially implemented application agent. It captures the common features of all application agents in our real-time agent model. Only by extending the general application agent, application developers can easily develop their own application agents. Based on our real-time agent model, the following features of an application agent can be observed.

The design of the application agents is role-based. An application agent can carry multiple roles and at a time it can only play one role. In other words, it can play another role only after it finished the current work. An application agent knows all the dependencies among its roles. That is, it knows which role has to be executed first and which one should be played later, or concurrently. Based on the parameters from the task planner, it decides which role it performs next. An application agent is almost stateless at the end of a mission. In other words, it does not retain the knowledge that is pertinent to a previous mission except perhaps in statistical data relevant to a future mission. However,

within a mission, the agent can retain the knowledge of the roles that it has performed, and this knowledge also can be shared by other roles that it will perform. An application system is constructed with a pool of application agents. During the initialization of the application, all its agents are created and started. Then these agents automatically enter dormant state and wait for instructions from the task planner. Different instances of a mission involve different agent instances.

3.2.3.1 General Application Agent Model

The agent state diagram is shown below.

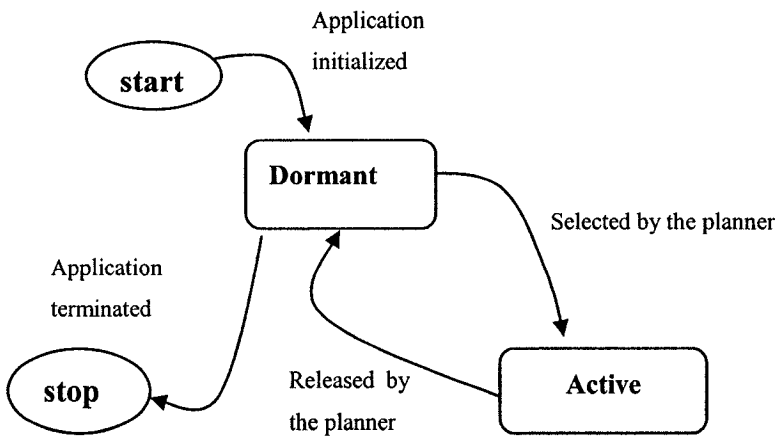

Figure 3-6 State Diagram of General Application Agent

The main functions of the General Application Agent are:

(i) Load multiple roles at the initial time of the agent,

(ii) Play one role at one time,

(iii) Know the dependencies of its roles and which roles can be played next,

(iv) Schedule and launch the role by taking parameters from the planner,

(v) Provide storage mechanism for roles to share knowledge within an agent,

(vi) Coordinate with the planner, and

(vii) Provide agent life cycle management and communication support.

3.2.3.2 Design of the General Application Agent

The architecture of the general application is shown in Figure 3-7. The general agent takes care of the life-cycle management of an agent. That includes creating, initializing, deactivating, activating and terminating the agent. An application agent has a role pool to keep all the roles that it will play. In order to support message-passing-based communication, an inbox of ACL messages is designed. It keeps all the incoming messages. The internal scheduler is responsible for coordinating with the task planner and scheduling the execution of roles. A list of role objects and their names are kept in an agent. In a fingerprint verification mission, for example, an image refiner agent may play the role slave refiner whose object is 'Slave_Refiner1' that is an instance of the Slave_Refiner role class. The role name ('slave_refiner') and the object of the role (Slave_Refiner1) are stored in the role list of the agent that can invoke this role by referring to its name in the list. Role dependency knowledge includes a set of parent-children role pairs. Each role pair involves the name of a parent role and the names of a set of children roles that are identifiable based on the name of their parent. The run-time knowledge shared among the roles is retrievable based on the name of the role that outputs the knowledge. This knowledge is a set of variable and value pairs.

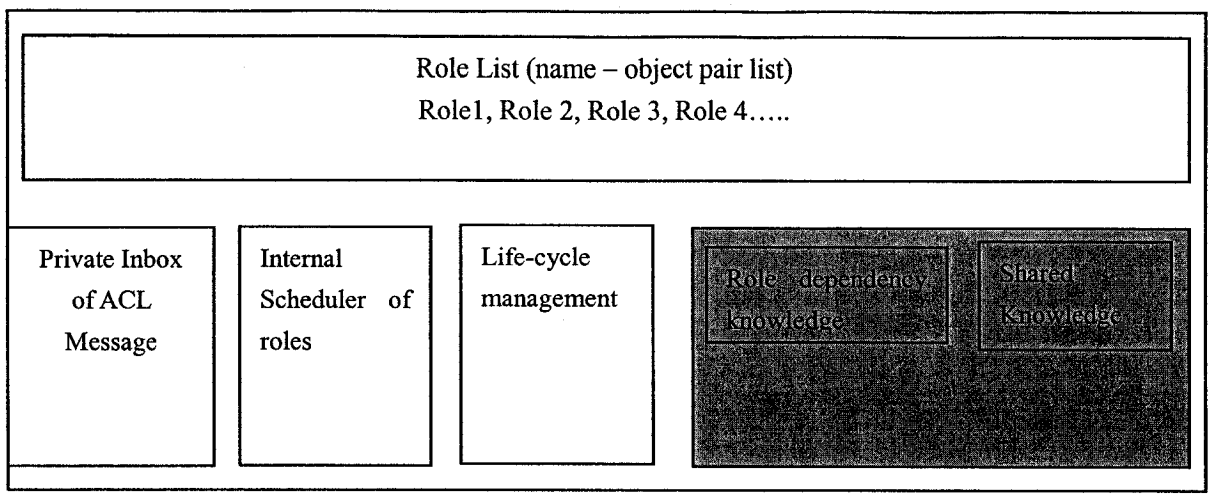

Figure 3-7 Architecture of Application Agent

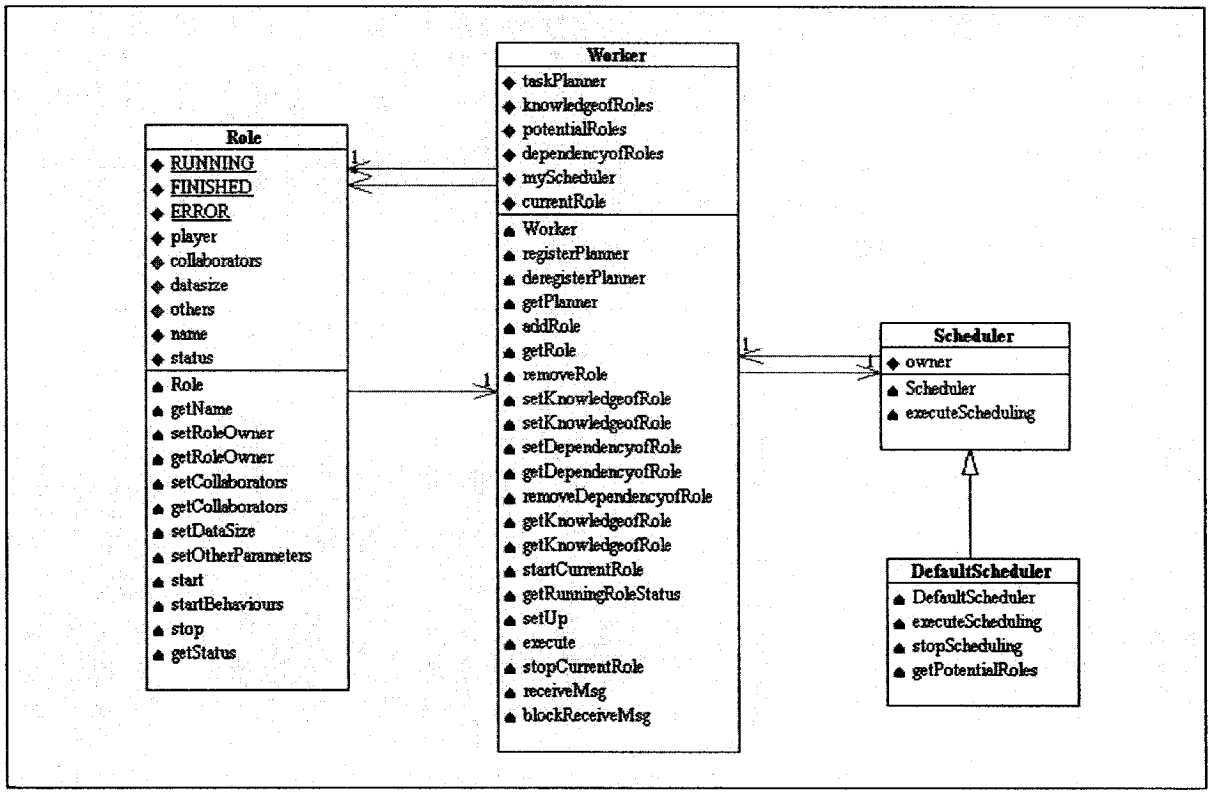

Figure 3-8 Class Diagram of Application Agent

The class diagram of the general application agent is shown in Figure 3-8. A default scheduler is already implemented in this platform. Developers can implement their own scheduler by extending the Scheduler class. This gives more flexibility to developers.

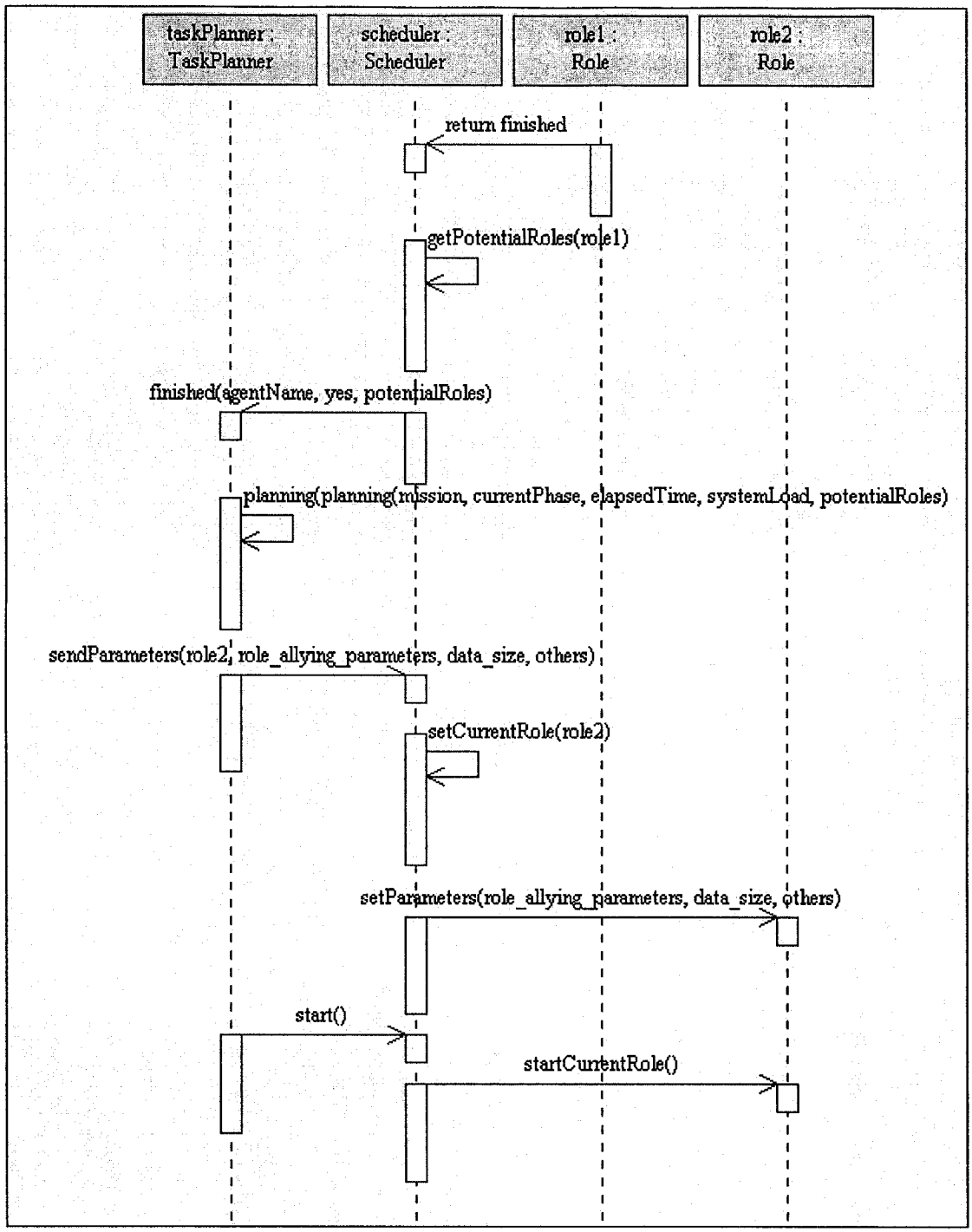

Figure 3-9 Switching from one role to the next

The most important part of the general application agent is the internal scheduler, which is responsible for starting/terminating the execution of roles by taking parameters

from the task planner and reporting the execution result of a role to task planner. The switch from one role to the next role is shown in Figure 3-9.

3.2.4 Protocol for Main Services

This section includes the description of the important protocols in the platform.

3.2.4.1 Process Plan Exception

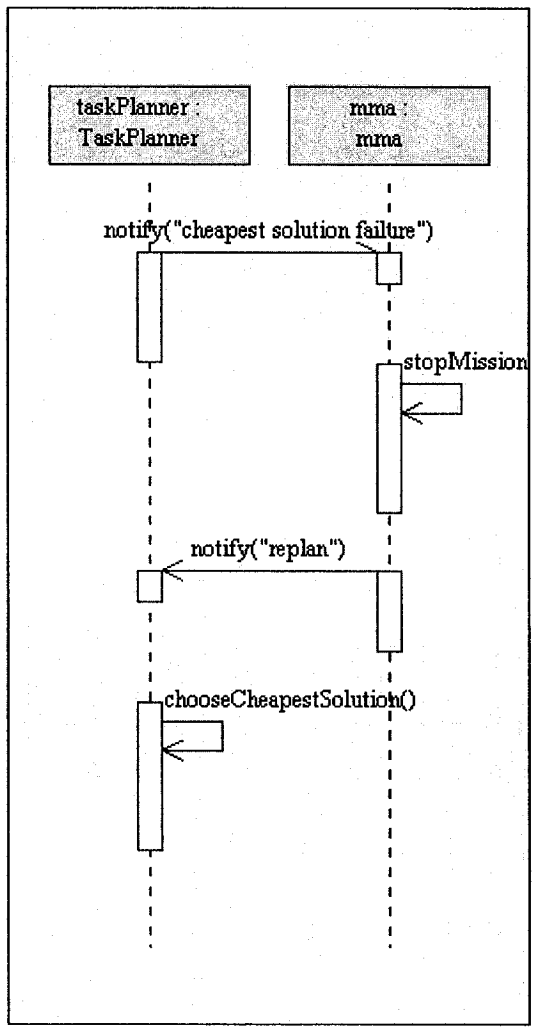

Figure 3-10 Planning exception processing

3.2.4.2 Initializing a Mission Instance

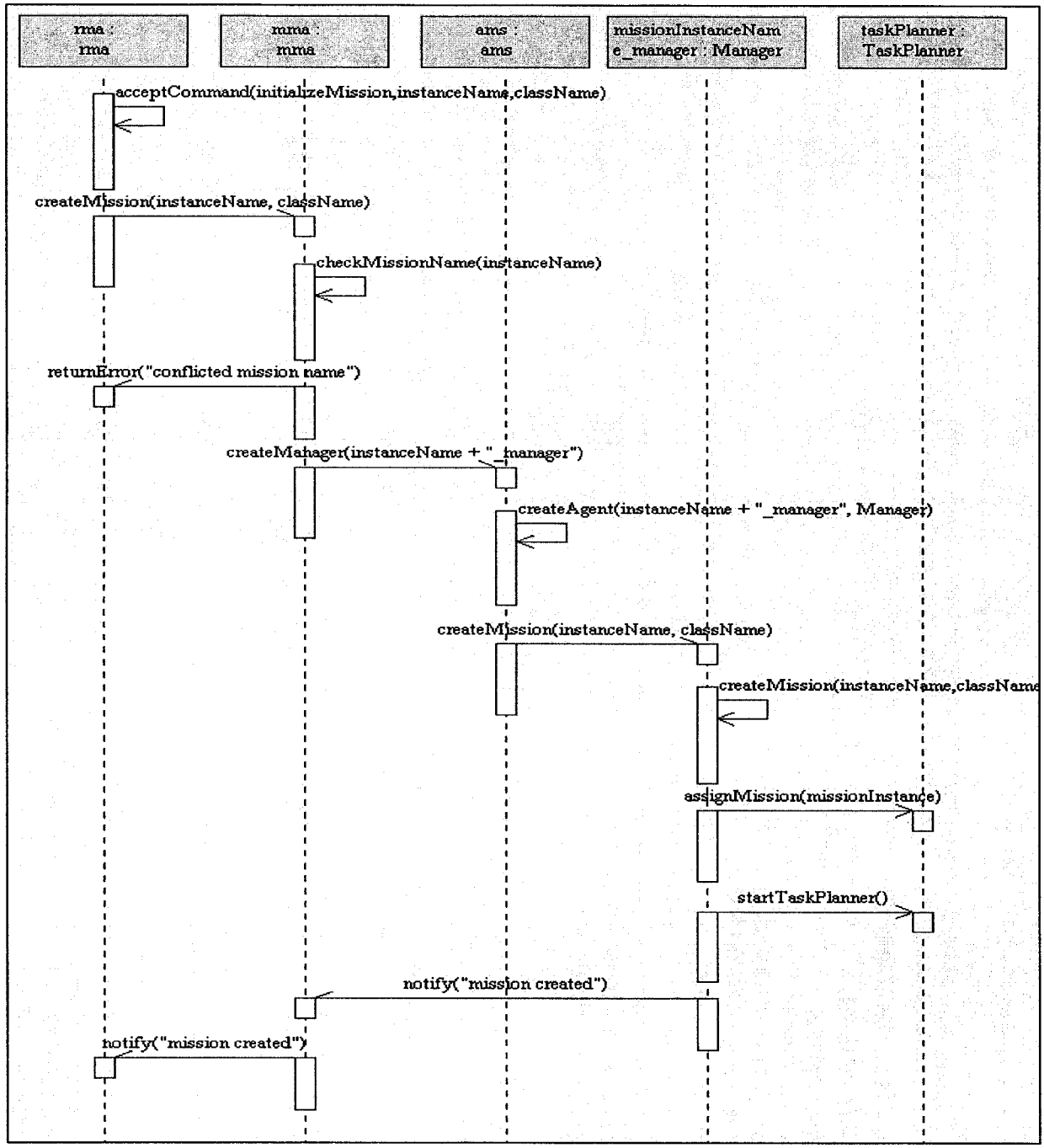

Figure 3-11 Initialization of a mission instance

3.2.4.3 Switching from One Phase to the Next One

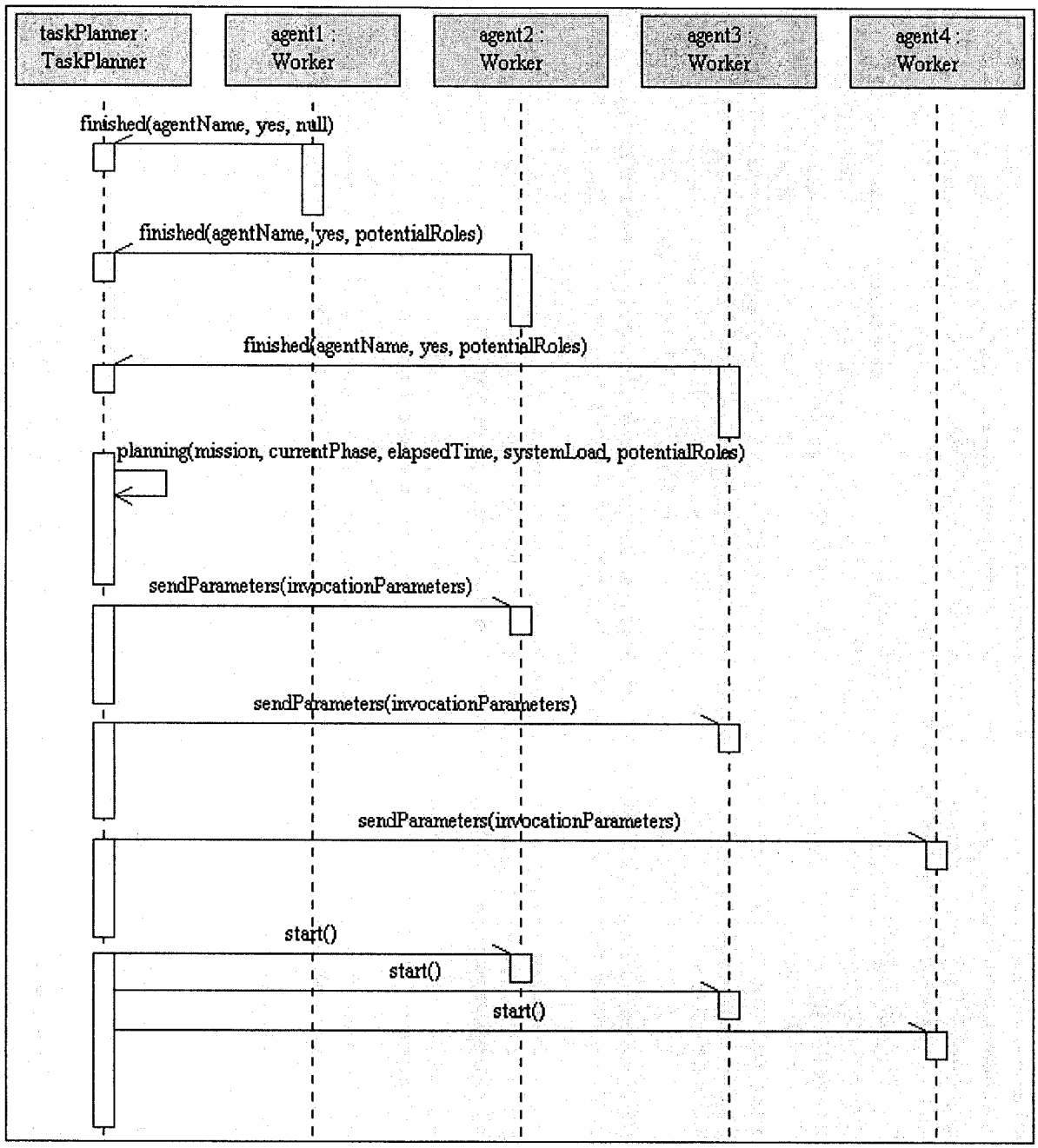

Figure 3-12 Switching from one phase to the next

3.2.4.4 Kill Mission

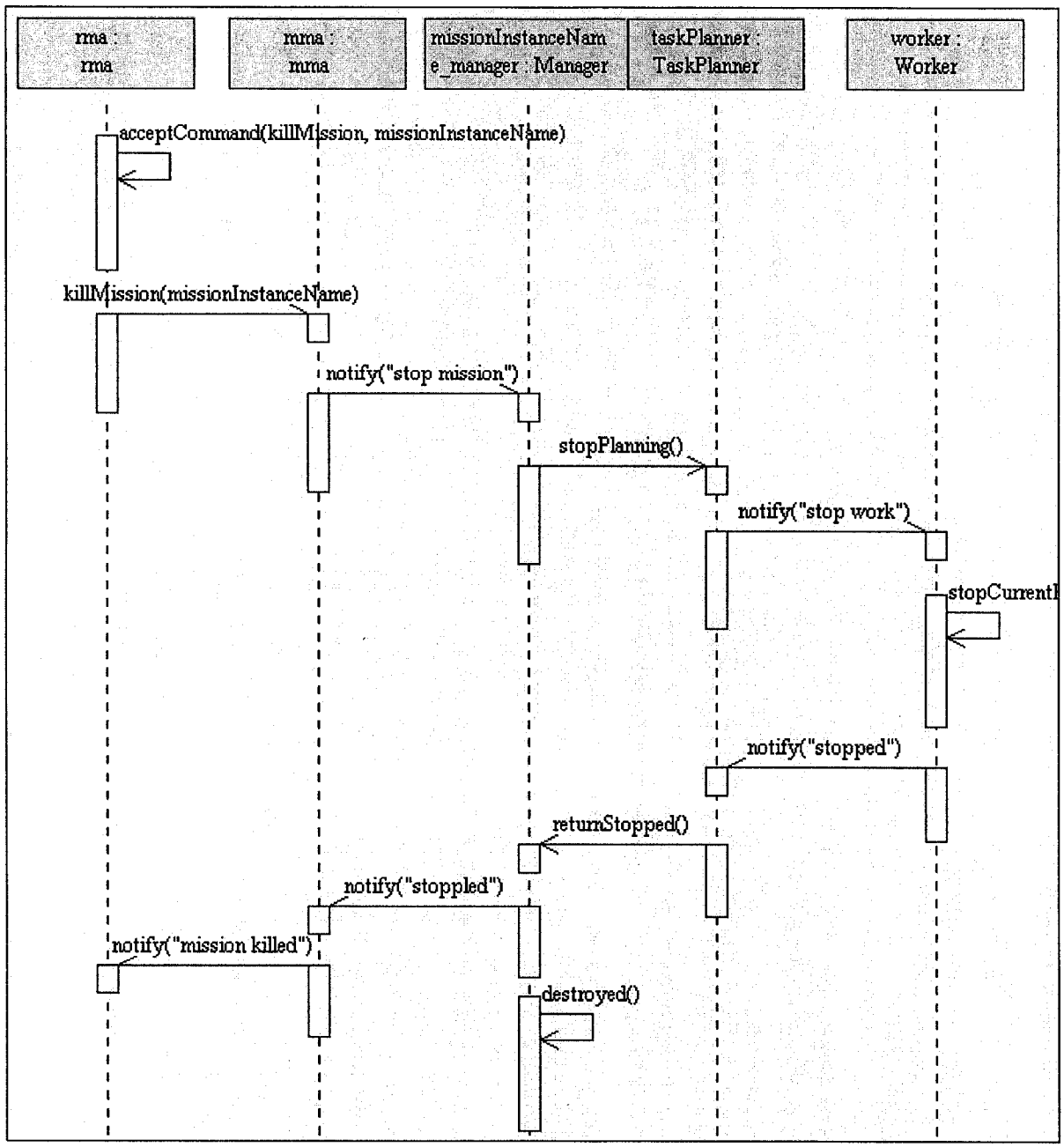

Figure 3-12 Kill a mission

3.3 Services from the Developer's Perspective

In this section, we summarize the services that the framework provides to developers and the developer's responsibility (what the developer has to implement) in the development of an application using this framework.

SPAF supports	Developer's responsibility
1. A mission skeleton formed by a set of API. 2. General application agent (API). 3. A role skeleton (API). 4. Internal scheduler interface (API). 5. Task planer interface (API). 6. Default role scheduling algorithm. 7. Default task planning algorithm. 8. Mission admission control and time consistency checking. 9. Planning exception handling. 10. Resource monitoring. 11. Agent management service (AMS) and message transport service (MTS).	1. A concrete mission class, involving mission definition, phase sequence definition and alternative solutions for each phase. 2. A set of application agents. Each solution includes some of them. 3. If necessary, customized task planner and internal scheduler.

Table 3-4 Services provided to developers

The services provided by our framework belong to two categories: *customizable services* and *fixed services*. Developers can implement new services by extending customizable services that involve mission skeleton, agent and role skeleton, task planning and role scheduling. On the other hand, the fixed services (mission admission control, planning exception handling, resource monitoring and AMS and MTS) are integrated into our framework to support the execution of the application. These services

cannot be customized by developers, but they are general enough to support different kinds of applications.

The mission skeleton (Figure 3-14) is formed by a set of classes: *Mission*, *Phase*, *Solution*, *ResourceNeeds* and *InvocationParameters*. It provides support for developers to create a concrete mission. The example of how to use this skeleton can be found in Chapter 4. In addition, a general application agent skeleton and a role skeleton (Figure 3-8) are provided, which can be used by the developers to implement application agents. Moreover, the implementation interfaces of the internal scheduler and task planner (Figure 3-15) are reserved for developers in case a customized task planner or internal scheduler is needed. Also, the default internal scheduler and task planner provides complete support for executing the customized mission and minimizing the application development effort.

All the fixed services are provided for the development/execution of soft real-time multi-agent systems based on the principle of quality-time tradeoff. Mission admission control only accepts a new mission when its resource requirement can be met by the current resource status of the system. It minimizes the resource conflict among the currently running missions and allows these missions to possibly meet their deadlines. Planning exception handling makes the execution of a mission more robust. Through the re-planning mechanism, this service gives a potentially failing mission another opportunity to finish the phase and eventually meet the deadline. Resource monitoring is very useful for load balancing and resource conflict resolution. It is almost transparent to

the developers and greatly simplifies the development of the application. Last but not the least, the AMS and MTS provide agent lifecycle management service and message delivery service, two basic services for the development of a multi-agent system.

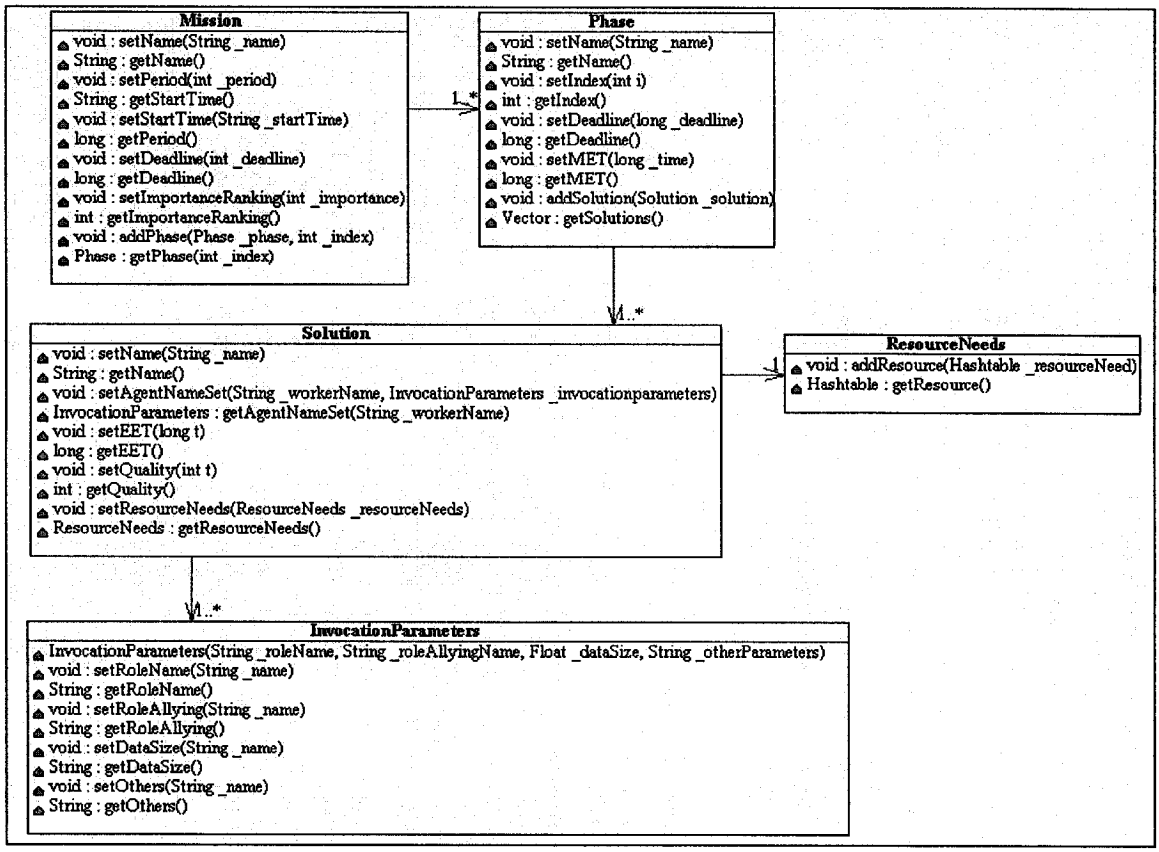

Figure 3-13 Class diagram of mission skeleton

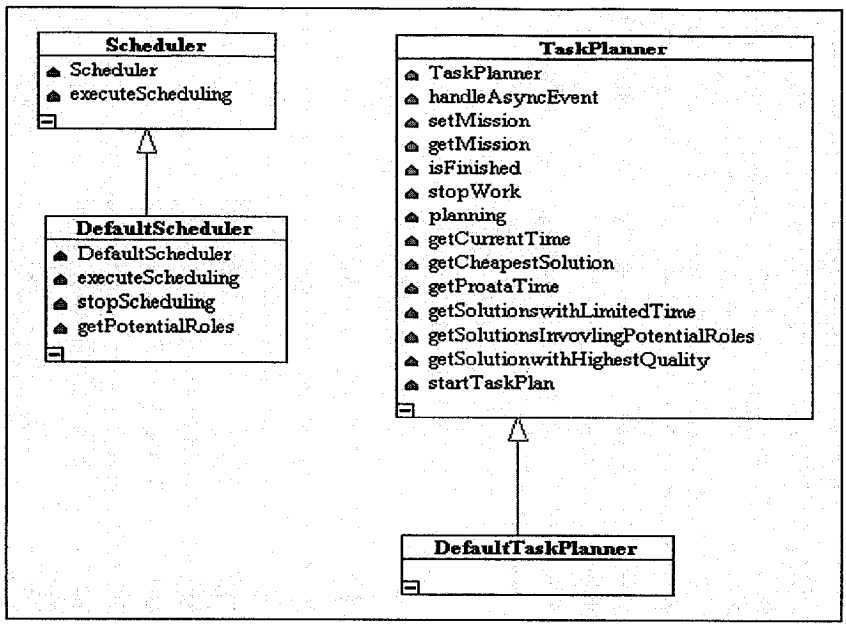

Figure 3-14 Class diagram of scheduler & task planner

Chapter 4

Case Study: Intelligent Security Monitoring System

In this section, we illustrate the use of SPAF by showing how the sample application — Intelligent Security Monitoring System (ISMS) — can fit into our model and can be developed using the classes of SPMM mission and application agent skeleton. The ISMS is a software tool with intelligent surveillance functions that enable effective monitoring of a building. The objective of this case study is to show how SPAF facilitates the design of a soft real-time application involving quality-time tradeoff and the agent programming through the pre-defined application API involving a role-based agent model.

4.1 Intelligent Security Monitoring System (ISMS)

Generally, the ISMS resides over a set of hosts connected via a LAN. In other words, the network delay is somewhat predictable. In the system, there are various monitoring missions running concurrently, each of which has soft real-time constraint. For example, there are 10 periodic missions that monitor 10 different local sites such as lobbies and hallways, and there are sporadic fingerprint verification missions invoked at 10 entrances and face verification missions invoked at another 5 entrances. Each mission is complex and involves multiple steps to be completed. A fingerprint mission, for example, may involve four phases: image sampling, image refinement, image recognition, and result

reporting. Different missions may have different importance measures. For example, the fingerprint verification mission may be more important than the lobby-monitoring mission because someone is waiting to enter. Also the importance of fingerprint verification missions may depend on the priorities of different entrances. Some missions do not have predictable arrival rates or patterns. Typically missions are independent and do not interact directly among themselves. Within a phase of a mission, there may be several alternative solutions to finish this phase. For example, in the image-refining phase of a fingerprint mission, different data granularity can be used. The larger granularity a solution uses, the less time it takes. Also, in fingerprint recognition phase, different recognition algorithms can be used, which may differ in terms of quality and computing time.

4.2 The Development of ISMS

Developing ISMS on SPAF involves the following steps:

(i) Defining each mission.

(ii) Defining each solution.

(iii) Implementing the application agents.

(iv) Implementing each role.

4.2.1 Defining Each Mission

From the perspective of the whole application, the developers have to identify and define the missions in terms of SPAF. In this security monitoring application, there are 25

missions. These missions vary in terms of deadlines, periods and importance rankings. For example, the fingerprint verification mission at one entrance has an importance ranking of 5 out of 10 and has to be finished within 10 seconds. Because the mission is sporadic, its period is 0. The whole mission involves four phases: image sampling, image refinement, image recognition, and result reporting. Image information on the fingerprint is gathered from the image device by a data sampling agent and piped through a series of filter agents for refinement. The refined image data is then passed to the image recognition agents for fingerprint matching. Based on the matching result, the report agents perform certain actions, such as approving, denying, alarming or recording.

Within each phase, there are several alternative solutions. For the image sampling phase, one algorithm can have different data sampling rates. This phase should be finished within 2 seconds. The minimum execution time of this phase for the simplest solution is 1 second. Similarly, the image refinement phase has one refinement algorithm and different data granularities, such as 80%, 60%, or 50% of the entire image data. Its deadline is 3 seconds and minimum execution time is 2 seconds. Within the image recognition phase, 3 different image recognition algorithms can be used. The deadline for this phase is 3 seconds and the minimum execution time for the simplest algorithm is 1.5 seconds. The last phase — result reporting — has to be finished within 2 seconds. An action is performed based on the image matching result and a report is generated. The level of detail in the report depends on the amount of time left. All the above detail can be specified by inheriting a sub-class from the base class *Mission*, and setting its

attributes. Similarly, the phases of a mission are specified by inheriting a sub-class from the base classes *Phase* and setting their attributes, as shown below.

```
// Defining Fingerprint Verification Mission (FVMission)
FVMission extend Mission {
    ...
    setImportanceRanking(5);        //define important ranking as 5
    setDeadline(10);                //define deadline as 10 seconds
    setPeriod(0);                   // no period is defined
    addPhase(SamplingPhase, 1);     // add phases one by one with index
    addPhase(RefiningPhase, 2);
    addPhase(RecognitionPhase, 3);
    addPhase(ReportingPhase, 4);
    ... }
```

Script 4-1 Defining fingerprint verification mission

```
// Defining Refining Phase
RefiningPhase extend Phase {
    ...
    setDeadline(3);                 //set the deadline as 3 seconds
    setMET(2);                      //set the minimum execution time as 2 seconds
    addSolution(Solution_80);       //add the solution whose data granularity is 80%
    addSolution(Solution_60);       //add the solution whose data granularity is 60%
    addSolution(Solution_50);       //add the solution whose data granularity is 50%
    ...
} ...
// Defining Recognition Phase
RecognitionPhase extend Phase {
    ...
    setDeadline(3);                 //set the deadline as 3 seconds
    setMET(1.5);                    //set the minimum execution time as 1.5 seconds
    addSolution(Solution_A1);       //add the solution with the recognition algorithm 1
    addSolution(Solution_A2);       //add the solution with the recognition algorithm 2
    addSolution(Solution_A3);       //add the solution with the recognition algorithm 3
    ...
} ...
```

Script 4-2 Defining phases

4.2.2 Defining Each Solution

Before defining each solution, the following work should be done: the definition of roles and their relationship, and the definition of application agents and role assignment. As Table 4-1 illustrates, four application agents (*Worker1, Worker2, Worker3,* and *Worker4*) are involved in the fingerprint verification mission. *Worker1* works as a *sampler* in Phase1, a *master-refiner* in Phase2 and different *image-extractors* in Phase3. *Worker2* acts as a *slave-refiner* in Phase2. *Worker3* works as a *slave-refiner* in Phase2, different *image-matchers* in Phase3 and *action-performer* in Phase4. *Worker4* acts as a *slave-refiner* in Phase2 and *report-generator* in Phase4. An arrow indicates the potential roles that an agent can play during the switching of phases. Moreover, *image-extractor* and *image-matcher* will work together to complete an image recognition algorithm, and *action-performer* and *report-generator* will work together to fulfill the task in Phase4.

Agent	**Phase1 (roles)**	**Phase2 (roles)**	**Phase3 (roles)**	**Phase4 (roles)**
Worker1	Sampler	Master-refiner	1) Image-extractor1 of algorithm1 2) Image-extractor2 of algorithm2 3) Image-extractor3 of algorithm3	
Worker2		Slave-refiner		
Worker3		Slave-refiner	1) Image-matcher1 of algorithm1 2) Image-matcher2 of algorithm2 3) Image-matcher3 of algorithm3	Action-performer
Worker4		Slave-refiner		Report-generator

Table 4-1 Application agents and roles

The definition of a solution can be done by inheriting a sub-class from the base class *Solution* and setting its attributes, as shown below.

```
//Defining the solution with 80% data granularity in Phase2
Solution_80 extend Solution {
    ...
    //set the quality ranking as 10, the highest within the phase
    setQuality(10)
    //set the estimated execution time as 2.7 seconds
    setEET(2.7)
    // set the resource needs, cpu, memory or bandwith
    setResrouceNeeds( new ResoruceNeeds());

    //define worker agents and their invocation parameters in the phase
    setAgentNameSet ("Worker1", new InvocationParameters("master-refiner", " slave1
    = Worker2, slave2 = Worker3, slave3 = Worker4", "0.8", ""));
    setAgentNameSet ("Worker2", new InvocationParameters("slave-refiner", "master =
    Worker1","0.8",""));
    setAgentNameSet ("Worker3, new InvocationParameters("slave-refiner", "master =
    Worker1", "0.8", ""));
    setAgentNameSet ("Worker4, new InvocationParameters("slave-refiner", "master =
    Worker1","0.8", ""));
    ... }
```

Script 4-3 Defining solution - 1

```
//Defining the solution with image recognition algorithm1 in Phase3
Solution_A1 extend Solution {
    ...
    //set the quality ranking as 10, the highest within the phase
    setQuality(10)
    //set the estimated execution time as 2.9 seconds
    setEET(2.9)
    // set the resource needs, cpu, memory or bandwith
    setResrouceNeeds( new ResoruceNeeds());
    //define worker agents and their invocation parameters in the phase
    setAgentNameSet ("Worker1", new InvocationParameters("Image-extractor1",
    "matcher = Worker3", "","",""));
    setAgentNameSet ("Worker3", new InvocationParameters("Image-matcher1",
    "extractor= Worker1","","",""));
    ... }
```

Script 4-4 Defining solution - 2

4.2.3 Implementing the application agents

The implementation of an application agent involves the role assignment and the definition of role dependencies. As shown in Table 4-1, the application agent *Worker1* takes the role *sampler* in the image sampling phase, plays the role *master-refiner* in the image refinement phase and then it performs one of three mutually exclusive roles: *image-extractor1, image-extractor2* and *image-extractor3,* which correspond to three alternative algorithms. This is specified by adding the five roles: *sampler, master-refiner, image-extractor1, image-extractor2* and *image-extractor3* using the *addRole()* method of *Worker1* sub-class of *Worker agent* base class, and setting the dependencies among them through *setDepedencyRole()*, as shown below.

```
//Implement application agent Worker1
Worker1 extend Worker{
    ...
    //add roles
    addRole(sampler);
    addRole(master-refiner);
    addRole(image-extractor1);
    addRole(image-extractor2);
    addRole(image-extractor3);

    //define role dependency
    setDepedencyRole("sampler","master-refiner");
    setDepedencyRole("master-refiner", "image-extractor1, image-extractor2,
    image-extractor3")
    ...}
```

Script 4-5 Implementing application agent

Note that this dependency reflects the developer's choice of using this agent as a

surviving agent during the switching from Phase1 to Phase2 and from Phase2 to Phase3. The choice of the role played by this agent is decided by the planner at run time. Meanwhile, part of the mission state is retained by this agent until the next assignment by the planner.

4.2.4 Implement each role

The implementation of each role is based on the work partners of an agent that are defined in its invocation parameter. For instance, in *Solution_80* of the image refinement phase, *Worker1* (playing the *main-refiner* role) works with *Worker2, Worker3* and *Worker4* (playing the *slave-Refiner* roles) to perform data-parallel image refinement. Also, *main-refiner* communicates with each *slave-Refiner*, but *slave-refiners* do not talk to each other. The related role parameter for *main-refiner* is *"slave1=Worker2, slave2=Worker3, slave3=Worker4"*. It is assigned to *main-refiner* by the task planner at run-time. Similarly, the related role parameter for *slave-refiner* is *"master=Worker1"*. The sample code segment for *Worker1* and *Worker2* is shown in Script 4-6 on the next page.

SPAF provides a complete model for developers to better use the principle of quality-time tradeoff in the design of soft real-time applications in open environments. Its self-planned mission framework greatly simplifies the development of such applications. In particular, its functional and data decomposition based solution model provides a powerful tool to developers for designing alternative solutions. Also, the availability of

the self-planner facilitates the dynamics of adjusting the mission quality based on run-time resource status, which strengthens the mission's ability to tolerate deadline missing failure and achieve higher average mission quality compared with the mission without planner.

```
//Implement Image –Extractor1
Image-Extractor1 extend Role {
    ...
    //implement this method that perform the feature extracting work
    public void startBehaviors(){
        FeatureData fd = featureExtract(data);// perform feature extracting
        sendFeatureData(image-matcher1, fd);//send the feature data to
                                            //image-matcher1
    }
    ...
}
//Implement Image – Matcher1
Image - Matcher1 extend Role {
    ...
    //implement this method that perform the image matching work
    public void startBehaviors(){
        FeatureData fd = waitingforFeatureData(image-extractor1);
        matchingResult mr = ImageMatching(fd);
        deposit("image-Matcher1", mr);    //deposit matching result that will be used in
                                          //the last phase
    }
    ... }
```

Script 4-6 Implementing application roles – 1

```
//Implement main-refiner role
Main-Refiner extend Role{
    ..
    //implement this method that perform the image refinement work
    public void startBehaviors(){
        Data d1,d2,d3,d4;
        sendWork(slave1, 1, image);        //send the first 1/4 work to slave1
        sendWork(slave2,2,image);          //send the second 1/4 work to slave2
        sendWork(slave3,3,image);          //send the third 1/4 work to slave3
        //perform image refinement on the fourth 1/4 work
        d4 = performRefinement(4, image);
        while (d1 == null || d2 == null || d3 == null || d4 == null){
            Data t = getWork()
            If(t.getSource() == slave1)
                d1 = t;
            else(t.getSource() == slave2)
                d2 = t;
            else(t.getSource() == slave3)
                d3 = t;
        }
        Data total = combineWork(d1,d2,d3,d4);          //combine the works
        deposit("main-refiner", total);          //deposit the combined data that
                                                 //will be used in the image recognition phase
    }
    ...
}
//Implement Slave-Refiner
Slave-Refiner extend Role {
    ...
    //implement this method that perform the image refinement work
    public void startBehaviors(){
        Image im = waitingForWork(master);  //waiting for work from master refiner

        //perform image refinement on the recievied work
        Data d = refine(im);
        sendBackWork(master, d);          //send the refined data back to master refiner
    }
    ... }
```

Script 4-7 Implementing application role - 2

Chapter 5

Performance Evaluation

In Chapter 4 we illustrated the use of our self-planned mission model in the design and implementation of soft real-time multi-agent based applications. So far, a working framework on real-time JVM has been implemented. In this chapter, a set of simulation experiments of our developed prototype Intelligent Security Monitoring System built on our framework is introduced and a performance study of the self-planner is conducted. The objectives of this evaluation are to measure the overhead of the planner, and to test the impact of the planner on both the quantified quality level of the results and the frequency of deadline misses. The experiment was performed on a LAN with 4 hosts each of which ran the TimeSys Linux GPL platform with Real-Time JVM implemented by TimeSys Corporation. A single JVM was used to support an agent container on each host.

5.1 Overhead of the Planner

Compared with an unplanned mission, the overhead of a generic planner for a planned mission comes from the following: message passing latency; planning algorithm; agent switching time. To evaluate this overhead, a mission involving 5 application agents (and one planner agent) was developed. The structure of the mission shown in Figure 5-1 is similar to the abovementioned fingerprint verification mission except that one more

worker agent is added to play the role of slave-refiner.

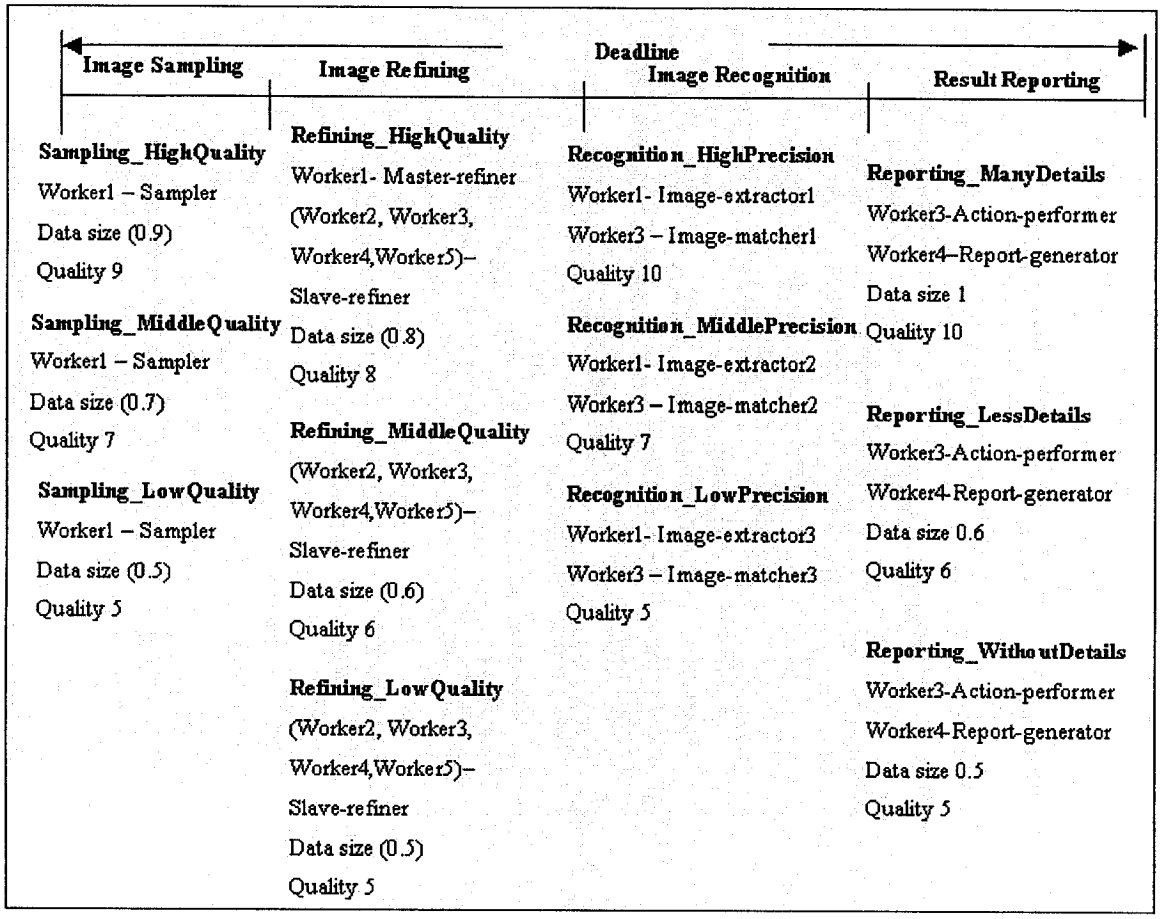

Figure 5-1 Structure of sample mission

These 5 application agents were manually distributed among four hosts: one host ran two agents while each of the remaining hosts ran a single agent. In order to achieve the same execution time during different runs, the deadline was relaxed so that the best solution was selected in every phase leading to the result with the best quality level. The execution time was measured for four different situations: mission without planner, mission with planner (no planning), mission with planner (planning without checking resource), and mission with planner (planning with resource checking). The unplanned mission always

uses the highest mission quality. The deadline of a phase can be adjusted during the runtime and is not used to calculate the utilization. So the utilization achieved by unplanned mission is not worse if some phase(s) did not complete within the deadline as long as the deadline of the mission is met. The graphs are averaged over several runs. The result is shown in Figure 5-2.

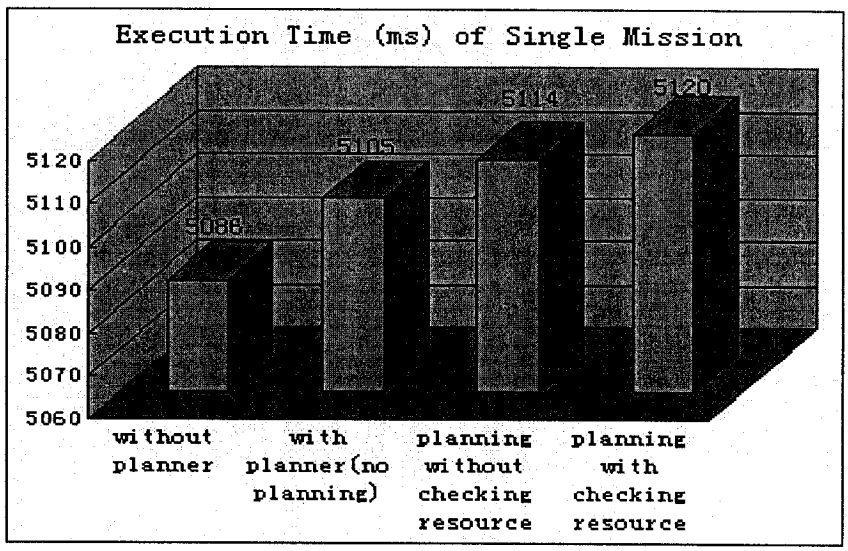

Figure 5-2 Execution Time of Single Mission in Different Situations

Figure 5-2 shows that the overhead of the planner is rather insignificant (less than 1% of the mission execution time — 34 ms for a mission that takes 5086 ms without self-planning). Such a low overhead is mainly due to the planner/application coordination model and the efficient communication layer of the platform.

5.2 Impact of the Planner on Average Quality and Deadline Missing Rate

In order to evaluate the effectiveness of the self-planner on multiple missions, 10 missions each with 4 phases were developed. The structure of these missions is the same

as the one in Figure 5-1. Each mission involved 5 application agents and one planner agent. So the total number of agents in the system was 60. This has provided a reasonable system load (rather than overloading) to the actual runtime environment without saturating the system and creating resource bottlenecks. We also chose reasonable deadline expectations so that under a light load situation, a mission could always be completed without violating the deadline.

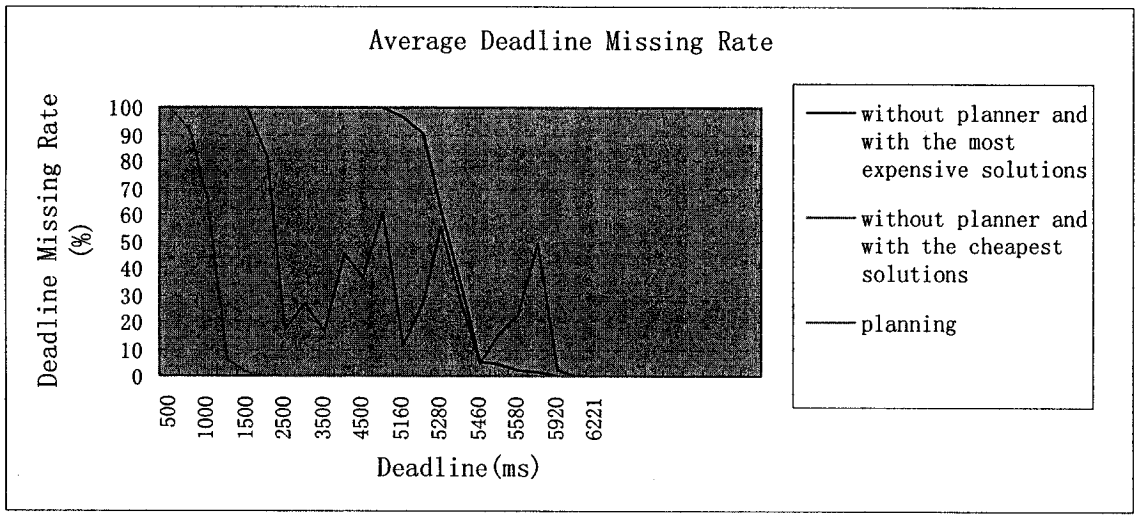

Figure 5-3 Average Deadline Missing Rate

The average execution time of these 10 missions without the planner is about 5331 ms. Figure 5-3 illustrates that the deadline missing rate improves when the deadline is tight (that is, less than the average mission execution time). When the deadline is tight, the planner likely chooses faster solutions, thereby improving the chance to complete before the deadline of the phase, leading to lower deadline misses. Figure 5-4 indicates that the quality of the results also improves when the deadline is tight. This is because self-planning increases the number of successful missions (each probably with lower

quality), which leads to higher overall quality.

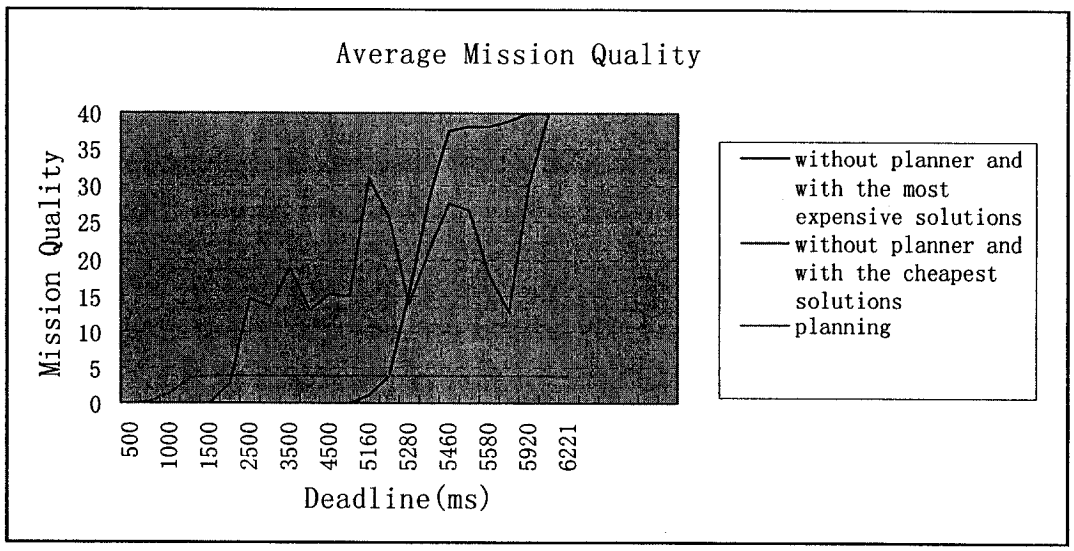

Figure 5-4 Average Mission Quality

Compared with the mission without planner and with the most expensive solutions, however, the test results also show that when the deadline is more relaxed (that is, greater than the average mission execution time), the missions with self-planning sometimes exhibit higher deadline missing rate and poorer result quality. So, here we discuss the conditions under which the tradeoff strategy may work. We assume that T_{min} is the sum of the execution times of all the fastest solutions of a mission; T_{max} is the sum of the execution times of all the slowest solutions of a mission; T_d is the deadline of the mission; T_p is the average time taken for each task planning; N is the number of phases of the mission. When $T_d > T_{min}+N*T_p$, our strategy may work. For Figures 5-3 and 5-4, the execution time of an unplanned mission is T_{max} because the planner always chooses the slowest solutions. Moreover, the max execution time of a planned mission with all the

slowest solutions is $T_{max}+N*T_p$. So when $T_{max} < T_d < T_{max}+N*T_p$, the planned mission has more chance to fail than the unplanned mission if there are no alternative simpler solutions that can be chosen to compensate for the planning time $N*T_p$ during the runtime. From Figures 5-3 and 5-4, we can see that T_{max} is about 5300 ms and $T_{max}+N*T_p$ is about 5600 ms. When 5300ms < T_d < 5600ms, the planned mission has higher average deadline miss rate and lower average mission quality. In addition, if the estimated execution times of all the solutions within each phase are very close, the planned mission will have higher probability to miss its deadline than the unplanned mission.

These preliminary results confirm that the self-planning model and the associated framework are meaningful and practicable tools for soft real-time agent applications. However, its effectiveness may depend on careful tradeoff between the planning overhead and the tightness of deadline. In fact, it may be best to adaptively trigger re-planning, depending on the remaining time before mission deadline.

Chapter 6

Conclusion and Future Work

We have developed a self-planned agent model that makes use of quality-time tradeoff in different phases of a real-time mission. The model involves integration of a few novel concepts including decomposition of mission tasks into concurrent activities using either functional (multiple role) decomposition or data (single role) decomposition, admission control for responsiveness to committed missions, and dynamic optimization using alternate solvers or alternate data granularity in the selected solver. Viability and design issues of this model are further studied by implementing a corresponding agent framework. This framework has to address realization issues such as application interface and continuity of transitioning from one mission phase to the next. The solution we have adopted involves application-selected agents that survive from one phase to another and in so doing will carry forward with them the necessary mission state. Preliminary experimental studies revealed that the planner overhead is rather insignificant, and it can potentially improve quality and deadline missing rate.

Future efforts should include more extensive application studies and extending the system to become node-crash tolerant. Fault-tolerance is an important asset for most real-time applications.

Bibliography

[1] M. Wooldridge, *An Introduction to Multi-Agent Systems*, John Wiley & Sons, UK. 2002.

[2] W. Lee, B. Sabata, "Admission Control and QoS Negotiations for Soft-Real Time Applications", In Proc. of Intl. Conf. on Multimedia Computing and Systems, Centroaffari, Florence, Italy, June 1999.

[3] Tarek F. Abdelzaher, Ella M. Atkins, Kang G. Shin, "QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control", IEEE Transactions on Computers, Volume 49, Issue 11, November 2000, pp. 1170–1183.

[4] John A. Stankovic, T. He, T. Abdelzaher, M. Marley, G, Tao, S. Son, C. Y. Lu, "Feedback Control Scheduling in Distributed Real-Time Systems", 22nd IEEE Real-Time Systems Symposium, London, England, December 2001.

[5] Ing-Ray Chen, "On Applying Imprecise Computation to Real-Time AI Systems", The Computer Journal, Vol. 38, No. 6, 1995.

[6] Jai-Hoon Kim, Kihyun Song, Kyunghee Choi, Gihyun Jung, SeunHun Jung, "Performance evaluation of on-line scheduling algorithms for imprecise computation", Real-Time Computing Systems and Applications, Proceedings, Fifth International Conference, 27-29 Oct. 1998, pp. 217–222.

[7] Wei-Kuan Shih, Che-Rung Lee, Ching-Hui Tang, "A fast algorithm for scheduling imprecise computations with timing constraints to minimize weighted error", Real-Time Systems Symposium, 2000, Proceedings, The 21st IEEE, 27-30 Nov. 2000, pp. 305–310.

[8] A. Garvey, V. Lesser, "A Survey of Research in Deliberative Real-Time Artificial Intelligence", University of Massachusetts Computer Science, Technical Report 93–84, November, 1993.

[9] D.J. Musliner, J.A. Hendler, A.K. Agrawala, E.H. Durfee, J.K. Strosnider, C.J. Paul, "The challenges of real-time AI", Computer, Volume 28, Issue 1, Jan. 1995, pp. 58–66.

[10] M. Boddy, T. Dean, "Deliberation Scheduling for Problem Solving in Time-Constrained Environments", Artificial Intelligence, Volume 67, Issue 2, June 1994, pp. 245–285.

[11] S. Zilberstein and S. J. Russell, "Constructing Utility-Driven Real-Time Systems

Using anytime Algorithms", In Proceedings of the IEEE Workshop on Imprecise and Approximate Computation, pp. 6–10, Phoenix, AZ, December 1992.

[12] Michael C. Horsch and David Poole, "An Anytime Algorithm for Decision Making under Uncertainty", In Proc. 14th Conference on Uncertainty in Artificial Intelligence (UAI-98), Madison, Wisconsin, USA, July 1998, pp. 246-255.

[13] V. Lesser, J. Pavlin, and E. Durfee, "Approximate Processing in Real-Time Problem Solving", AI Magazine, 9(1) pp. 49–61, Spring 1988.

[14] Keith S. Decker, Alan J. Garvey, Marty A. Humphrey, and Victor R. Lesser, "A real-time control architecture for an approximate processing blackboard system", International Journal of Pattern Recognition and Artificial Intelligence, 7(2) pp. 265–284, 1993.

[15] B. Horling, V. Lesser, R. Vincent and T. Wagner, "The Soft Real-Time Agent Control Architecture", Proceedings of the AAAI/KDD/UAI-2002 Joint Workshop on Real-Time Decision Support and Diagnosis Systems, July 2002.

[16] V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. NagendraPrasad, A. Raja, R. Vincent, P. Xuan, X.Q. Zhang, "Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework", Autonomous Agents and Multi-Agent Systems, Vol: 9, Num: 1, pp. 87–143, July 2004.

[17] E. Hodys, "A Scheduling Algorithm For A Real-Time Multi-Agent System", University of Rhode Island, 2000.

[18] E.A. Kendall, "Agent software engineering with role modeling", in: P. Ciancarini, M. Wooldridge, (Eds.), Proc. the First International Workshop (AOSE-2000), Springer-Verlag, Berlin, Germany, Jan. 2000, pp. 163-170.

[19] E.A. Kendall, "Role Models — Patterns of Agent System Analysis and Design", BT Technology Journal, Vol.17, No. 4, October 1999.

[20] N. Carriero, D. Gelerner, *How to write parallel programs: A first course*, MIT Press, Cambridge, MA, USA 1990.

[21] Real-Time Java Specification 1.0, https://rtsj.dev.java.net/

[22] FIPA Agent Management Specification, Foundation for Intelligent Physical Agents, http://www.fipa.org/specs/fipa00023/, 2000.

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