Specification and Selection of Network Management Agents

Ichiro Satoh[*]

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

Abstract. This paper proposes a framework for building a reusable mobile agent from two kinds of components: an itinerary component and an application-specific logic component. Both components are implemented as mobile agents. The former is a carrier of the latter over particular networks and the latter defines management tasks performed at each host independently of any network. This framework also provides a mechanism for matchmaking the two mobile agent-based components. Since the mechanism is formulated based on a process algebra approach, it can strictly select an itinerary component that is suitable to perform management tasks at hosts that the tasks want to visit over networks. A prototype implementation of this framework and its application were constructed on a Java-based mobile agent system.

1 Introduction

Network management for telecommunication systems is is a typical application of mobile agent technology. Adopting mobile agent technology eliminates the need for the administrator to constantly monitor many network management activities, e.g., installation and upgrading of software and periodic auditing of the network. There have been several attempts to apply this technology to network management tasks.

There has been a serious problem associated with the development of mobile agent-based network management systems in addition to security problems. Such systems are required to efficiently migrate their agents among all specified multiple hosts because the itineraries of agents seriously affect the achievement and efficiency of network management tasks. Network management systems on the other hand must often handle incomplete networks that may have various malfunctions and disconnections and whose exact topology may not be known. It is almost impossible to dynamically generate an efficient itinerary among multiple hosts. As a result, many existing mobile agent-based network management systems explicitly and implicitly assume that their mobile agents have been statically designed for particular itineraries over their target networks. However, such an agent that has been optimized for particular networks cannot be reused in other networks.

To solve this, we constructed a framework for building and operating mobile agents for network management without losing their reusability or efficiency. The framework

[*] E-mail: ichiro@nii.ac.jp

separates the application-specific tasks and itineraries of mobile agents. The former defines network management tasks independently of any networks and the latter can be optimized for particular networks. The framework also offers a mechanism for matchmaking between the two. Since the mechanism is formulated based on an extended process algebra for reasoning about the itineraries of mobile agents, it can select an appropriate itinerary that can satisfy the requirements of a network management task. The current implementation of the framework is built on a Java-based mobile agent system, called MobileSpaces [9].

This paper is organized as follows: Section 2 presents the basic ideas behind this framework and Section 3 defines the process algebra for specifying mobile agents. Section 4 describes a prototype implementation of the framework and Section 5 presents some applications. Section 6 surveys related work and Section 7 sums up with concluding remarks.

2 Approach

The goal of this paper is to propose a framework for building and operating mobile agents, which can autonomously travel among hosts on multiple sub-networks to perform their management tasks at each of the hosts they visit.

2.1 Two-Layered Mobile Agents

The framework divides a mobile agent for network management into two layered mobile agents as follows:

Navigator Agent is independent of any application-specific tasks. Instead, it has its own itinerary on a sub-network and carries task agents among their multiple destinations on the sub-network. It is reused with arbitrary network management tasks.

Task Agent is an application-specific agent that performs its management task at each of the hosts it visits. It can travel from sub-network to sub-network, but may be unfamiliar with the sub-networks it visits. It can be reused in other sub-networks.

Most mobile agents for network management, which itinerate among multiple hosts, often perform the same code, such as monitoring and controlling various equipments, at each of the hosts they visit. Therefore, task agents do not always have to change their tasks according to its visiting hosts.

2.2 Mobile Agent Matchmaking Mechanism

This framework also provides a mechanism for matchmaking between tasks agents and navigator agents. The mechanism, called Agent Pool, stores idle agents in a manner similar to that in a bus-terminal or a taxi stand (Fig. 1). Each sub-network has more than one agent pool for storing navigator agents with various itineraries. Each task agent is responsible for traveling among the agent pools of its destination sub-networks, where each navigator agent is responsible for navigating its inner agents among the hosts in its sub-network, and has been designed to return to its place soon after achieving its

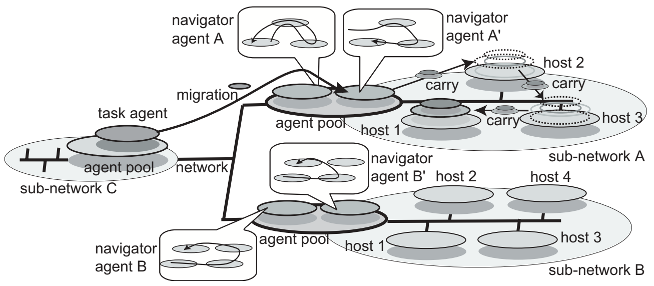

Fig. 1. Agent pools, navigator agents, and task agent

navigation task to wait for the next task. Therefore, to travel among some of the hosts on a sub-network, a task agent migrates to an agent pool at the sub-network and selects a navigator agent stored in the pool to carry it among the hosts. Also, each agent pool should assign a task agent to idle navigator agents, which are staying at agent pools, since moving agents are busy in achieving their current tasks.

Since mobile agents are written in general-purpose programming languages, such as Java, it is difficult to extract only the itineraries of mobile agents from their programs. We therefore defined a process algebra-based specification language for the itineraries of mobile agents to select mobile agents according to their itineraries. This framework assumes that each task agent specifies its required itinerary as a term of the language and each navigator agent specifies its own possible itinerary as a term of a subset of the language. The selection of navigator agents is formulated based on an algebraic order relation over the terms of the language.

3 Mobile Agent Selection

A typical mobile agent for network management must monitor and control some equipment at multiple hosts over a network whose exact topology may not be known and which may have various malfunctions and disconnections. Such an agent often has its own itinerary to statically solve problems in its target network. When a task agent is carried by a navigator agent, the performance and achievement of the task agent is dependent on the itinerary of the navigator. If a mobile agent gathers information from a host and reflects the information on other hosts, its order of movement among these hosts may affect their states. Therefore, such an agent must migrate among hosts according to a specified itinerary. However, if an agent can travel among hosts to aggregate interesting information from them without writing on them, the order of movement may be independent of its achievement. Moreover, an agent's itinerary is often dependent on the results of the agent's network management task. For example, such an agent can determine its destinations based on information, such as routing tables, it has acquired from the hosts that it has visited so far.

Definition 3.1 The set $\mathcal{E}$ of expressions of the language, ranged over by $E, E_1, E_2, \ldots$ is defined recursively by the following abstract syntax:

$$E ::= 0 \;\mid\; \ell \;\mid\; E_1 \,;\, E_2 \;\mid\; E_1 + E_2 \;\mid\; E_1 \,\#\, E_2 \;\mid\; E_1 \,\%\, E_2 \;\mid\; E_1 \,\&\, E_2 \;\mid\; E^{\star}$$

where $\mathcal{L}$ is the set of location names, ranged over by $\ell, \ell_1, \ell_2, \ldots$. We often omit $\mathtt{0}$. We describe a subset language of $\mathcal{E}$ as $\mathcal{S}$, when eliminating $E_1 \mathbin{\#} E_2$, $E_1 \mathbin{\%} E_2$, $E_1 \mathbin{\&} E_2$, and $E^{\star}$ from $\mathcal{E}$. Let $S, S_1, S_2, \ldots$ be elements of $\mathcal{S}$. □

This framework assumes that each agent has its own itinerary written in $\mathcal{S}$. Since each agent has an interpreter for the terms of $\mathcal{S}$, it can dynamically evaluate its itinerary and migrate among hosts along the itinerary. Intuitively, the meanings of the construction are as follows: $\mathtt{0}$ represents a terminated itinerary; ℓ represents agent migration to a host whose name or network address is ℓ; $E_1 \,;\, E_2$ denotes the sequential composition of two itineraries E_1 and E_2. If the migration of E_1 terminates, then the migration of E_2 follows that of E_1; $E_1 + E_2$ denotes that an agent moves according to either E_1 or E_2 where selection can be explicitly done by processing the agent; $E_1 \# E_2$ means that an agent can select either E_1 or E_2 under its control regardless of its processing; $E_1 \% E_2$ means that an agent can follow either E_1 before E_2 or E_2 before E_1 as its itinerary; $E_1 \& E_2$ means that two itineraries E_1 and E_2 can be performed asynchronously[1]; $E^{\star}$ is the transitive closure of E and means that an agent can move along E an arbitrary number of times. The formal semantics of the language is defined as the following labeled transition rules:

Definition 3.2 The language is a labeled transition system $\langle\, \mathcal{E},\, \mathcal{L} \cup \{\tau\}\, \{ \xrightarrow{\alpha} \subseteq \mathcal{E} \times \mathcal{E} \mid \alpha \in \mathcal{E} \cup \{\tau\}\, \}\, \rangle$ defined as induction rules as given below:

$$\frac{-}{\ell \xrightarrow{\ell} \mathtt{0}} \quad \frac{E_1 \xrightarrow{\ell} E_1'}{E_1 \,;\, E_2 \xrightarrow{\ell} E_1' \,;\, E_2} \quad \frac{E_1 \xrightarrow{\ell} E_1'}{E_1 + E_2 \xrightarrow{\ell} E_1'} \quad \frac{E_2 \xrightarrow{\ell} E_2'}{E_1 + E_2 \xrightarrow{\ell} E_2'}$$

$$\frac{E_1 \xrightarrow{\ell} E_1'}{E_1 \mathbin{\&} E_2 \xrightarrow{\ell} E_1' \mathbin{\&} E_2} \quad \frac{E_2 \xrightarrow{\ell} E_2'}{E_1 \mathbin{\&} E_2 \xrightarrow{\ell} E_1 \mathbin{\&} E_2'}$$

$$\frac{-}{E_1 \mathbin{\#} E_2 \xrightarrow{\tau} E_1} \quad \frac{-}{E_1 \mathbin{\#} E_2 \xrightarrow{\tau} E_2} \quad \frac{-}{E_1 \mathbin{\%} E_2 \xrightarrow{\tau} E_1 \,;\, E_2} \quad \frac{-}{E_1 \mathbin{\%} E_2 \xrightarrow{\tau} E_2 \,;\, E_1}$$

$$\frac{E_1 \xrightarrow{\tau} E_1'}{E_1 \,;\, E_2 \xrightarrow{\tau} E_1' \,;\, E_2} \quad \frac{E_1 \xrightarrow{\tau} E_1'}{E_1 + E_2 \xrightarrow{\tau} E_1'} \quad \frac{E_2 \xrightarrow{\tau} E_2'}{E_1 + E_2 \xrightarrow{\tau} E_2'}$$

$$\frac{E_1 \xrightarrow{\tau} E_1'}{E_1 \mathbin{\&} E_2 \xrightarrow{\tau} E_1' \mathbin{\&} E_2} \quad \frac{E_2 \xrightarrow{\tau} E_2'}{E_1 \mathbin{\&} E_2 \xrightarrow{\tau} E_1 \mathbin{\&} E_2'}$$

where $\mathtt{0} \,;\, E$ is treated to be syntactically equal to E and $E^{\star}$ is recursively defined as $\mathtt{0} \mathbin{\#} (E \,;\, E^{\star})$. We often abbreviate $E_0 \xrightarrow{\tau} E_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} E_{n-1} \xrightarrow{\tau} E_n$ to $E_0 (\xrightarrow{\tau})^n E_n$. □

In Definition 3.2, the ℓ-transition defines the semantics of an agent's mobility. For example $E \xrightarrow{\ell} E'$ means that the agent moves to a host named ℓ and then behaves as E'. Also, if there are two possible transitions $E \xrightarrow{\ell_1} E_1$ and $E \xrightarrow{\ell_2} E_2$ in an agent, the agent being processed chooses one of destinations ℓ_1 and ℓ_2. However, the τ-transition corresponds to a non-deterministic choice in an agent's itinerary. Let us describe three agent migration patterns studied in [1]. To simplify our discussion hereafter, we introduce three macros, corresponding to the patterns, e.g., *Travel*, *Star*, and *Turn*. These

[1] In CCS-like process algebras, $\&$ is an operator for specifying parallel executions. The operational semantics of the language is an interleaving model in the literature of process algebras, and each agent migration is an atomic action.

macros do not extend the language because they are mapped into $\mathcal{E}$. We will describe the list of host names as $[\ell_1, \ell_2, \ldots, \ell_n]$, where $\ell_1, \ldots, \ell_n \in \mathcal{L}$. Let [] be an empty list, $car(X)$ be the top element of list X, i.e., let ℓ_1 and $cdr(X)$ be the remaining list of X except for the top element, i.e., $[\ell_2, \ldots, \ell_n]$.

$$\begin{aligned}
Travel(\$(\mathrm{X})) &\stackrel{\mathrm{def}}{=} car(\$(\mathrm{X}))\ ;\ Travel(cdr(\$(\mathrm{X}))) \\
Travel([]) &\stackrel{\mathrm{def}}{=} \mathsf{0} \\
Star(\$(\mathrm{X})|h) &\stackrel{\mathrm{def}}{=} (car(\$(\mathrm{X}))\ ;\ h)\ ;\ Star(cdr(\$(\mathrm{X}))|h) \\
Star([]|h) &\stackrel{\mathrm{def}}{=} \mathsf{0}
\end{aligned}$$

Let h be an element of $\mathcal{L}$ and X be a list of host names in $\mathcal{L}$. To illustrate the transition defined in Definition 3.2, we shows a partial transition of $Star([a, b, c]|h)$ as follows:

$$\begin{aligned}
Star([a,b,c]|h) &\stackrel{\mathrm{def}}{=} (a\ ;\ h)\ ;\ Star([b,c]|h) \\
&\xrightarrow{a} h\ ;\ Star([b,c]|h) \\
&\xrightarrow{h} Star([b,c]|h) \\
&\stackrel{\mathrm{def}}{=} (b\ ;\ h)\ ;\ Star([c]|h) \\
&\xrightarrow{b} h\ ;\ Star([c]|h) \\
&\xrightarrow{h} Star([c]|h)
\end{aligned}$$

We next formulate an algebraic order relation based on the concept of bisimulation [7]. The relation is suitable for selecting one of the navigator agents whose itineraries can satisfy the requirement of a task agent.

Definition 3.3 A binary relation $\mathcal{R}^n$ ($\mathcal{R} \subseteq (\mathcal{E} \times \mathcal{S}) \times \mathcal{N}$) is an *n-itinerary* prebisimulation, where $\mathcal{N}$ is the set of natural numbers. If whenever $(E, S) \in \mathcal{R}^n$ where $n \geq 0$, then the following hold for all $\ell \in \mathcal{L}$ or τ.

(i) if $E \xrightarrow{\ell} E'$ then there is an S' such that $S \xrightarrow{\ell} S'$ and $(E', S') \in \mathcal{R}^{n-1}$
(ii) $E\,(\xrightarrow{\tau})^*\,E'$ and $(E', S) \in \mathcal{R}^n$
(iii) if $S \xrightarrow{\ell} S'$ then there exist E', E'' such that $E\,(\xrightarrow{\tau})^* E' \xrightarrow{\ell} E''$ and $(E', S') \in \mathcal{R}^{n-1}$

where $E \sqsupseteq_n S$ if there exist some n-itinerary prebisimulations such that $(E, S) \in \mathcal{R}^n$. We call $\sqsupseteq_n$ *n-itinerary* order. □

The informal meaning of $E \sqsupseteq_n S$ is that S is included in one of the permissible itineraries specified in E and n corresponds to the number of movements of the agent that can satisfy E. There are some basic examples below.

- $(a \mathbin{\%} b \mathbin{\%} c)\ ;\ h \sqsupseteq_4 c\ ;\ a\ ;\ b\ ;\ h$
  where the right side requires an agent to migrate among three hosts a, b, and c in indefinite order and then return to host h and the right side migrates among three hosts c, a, and b sequentially. When the left side is changed to $a\ ;\ b\ ;\ c\ ;\ h$, the relation is still preserved, but when the left side becomes $a\ ;\ h\ ;\ b\ ;\ h\ ;\ c\ ;\ h$ or $a\ ;\ b\ ;\ h$, the relation is not preserved.
- $((a\ ;\ b\ ;\ c)\ \&\ h^*)\ ;\ h \sqsupseteq_6 a\ ;\ h\ ;\ b\ ;\ h\ ;\ c\ ;\ h$
  where the left side allows an agent to drop in at host h at arbitrary times during the itinerary $a\ ;\ b\ ;\ c$ and then finish its movement at host h. The right is a star-shaped route between three destinations, a, b, c and host h can satisfy the left side.

4 Design and Implementation

This section presents a prototype implementation of our framework. We tried to keep the implementation within the framework as much as possible.

4.1 Hierarchical Mobile Agents

Before describing the framework presented in this paper, let us briefly review the MobileSpaces mobile agent system that has provided the infrastructure for this framework. [2] Mobile agents in MobileSpaces are programmable entities like other mobile agents. They are capable of conserving their state while on the move and their itineraries can include multiple hosts. Furthermore, MobileSpaces provides each mobile agent with two novel concepts: *agent hierarchy* and *group migration*. The former means that another mobile agent can be contained within a mobile agent. The latter means that each mobile agent can migrate to another mobile agent or computer along with all its inner agents, as long as the destination accepts them. Therefore, an agent can contain other mobile agents inside it. Each agent has direct control of all its inner agents and can thus instruct them to move to other locations and destroy them. In contrast, each agent has no direct control over its container agents. Instead, each agent can have a set of service methods, which can be accessed by its containers. Each agent has a globally unique name and can have more than one active thread under the control of the runtime system.

4.2 Navigator Agent

Each navigator agent is a container of one or more task agents and is responsible for carrying them to hosts in the network it covers. It travels with its inner agents in accordance with its itinerary written in $\mathcal{S}$ and invokes the callback methods of its inner task agents at certain times, such as arrival and departure. Each navigator agent is designed to go back to its agent pool and then register its itinerary at the pool soon after achieving its navigation to wait for the next task. This framework provides abstract classes in the Java language and navigator agents can be defined by extending these classes.

```
public class NavigatorAgent extends MobileAgent {
  void setRoute(Route r) throws ... { ... }
  void moveTo(Host h) throws IllegalHostException,
    NoSuchHostException ... { ... }
  void moveToNext() throws MultiplePossibleHostsException,
    NoSuchHostException ... { ... }
  Host[] getPossibleHosts() ... { ... }
  void arrivedAt(Host here);
  void depaturingFor(Host dst);
  ...
}
```

Each navigator agent has its own itinerary as a term of $\mathcal{S}$ and registers the term with itself and an agent pool, in which it is stored, by invoking the `setRoute()` method as follows:

[2] Details on the MobileSpaces mobile agent system can be found in our previous paper [9].

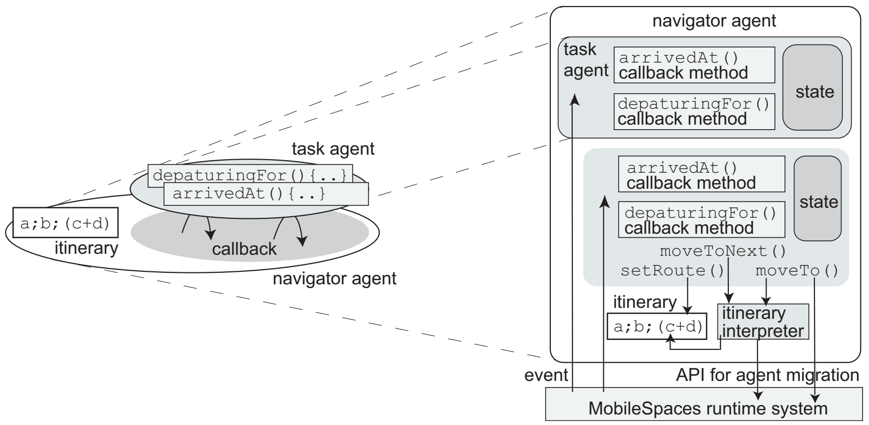

Fig. 2. Structure of navigator agent

```
setRoute(new Route("a;b;(c+d)"));
```

where `a;b;(c+d)` is an itinerary attached to the navigator agent and means that the agent migrates to host `a` and then to host `b`. The agent can then select either host `c` or `d` according to its own processing results. Each agent can migrate itself over a network by using the following two approaches.

The first approach allows each agent to move along the itinerary it has registered with itself. Each agent has a lightweight interpreter for the language in $\mathcal{S}$. When the agent invokes the `moveToNext()` method, the interpreter evaluates the agent's next destination from the itinerary and automatically moves the agent to the destination. However, if the itinerary contains one or more candidate destinations combined by selective operator `+`, then the invocation of the method throws an exception, named `MultiplePossibleHostsException`. The agent can get all the destinations that it can move to at the next hop by invoking the `getPossibleHosts()` method and it moves to one of these by invoking the `moveTo(dst)` method with the selected destination specified as `dst`. For example, suppose that an agent registers `a;b;(c+d)` as its own itinerary. As we can see Fig. 3, it performs method `moveToNext()` two times for two hops; from the current host to `a` and then from host `a` to `b`. It can then select either `c` or `d` and then perform the `moveTo(dst)` method with the name of the selected destination as the method's argument.

The second approach corresponds to the common approach used in existing mobile agent systems. That is, an agent explicitly specifies its destination whenever it migrates itself over a network. The `moveTo()` of the `NavigatorAgent` class causes the agent to move from host `b` to the destination specified as its argument. For example, an agent whose itinerary is `a;b;(c+d)` can invoke the `moveTo()` method with `a` and then `b` to move to host `a` and then to `b`. It can then invoke the same method with either `c` or `d`.

This framework restricts navigator agents from straying from the itinerary they registered with themselves. In both the above approaches, when the movement of a mobile agent deviates from the itinerary registered by invoking the `setRoute()` method, it is constrained and `IllegalHostException` is thrown to the agent. Each navigator agent can explicitly limit the length of the execution period for its incoming task agents after arriving at each destination. When the time limit of a task agent inside it expires, it automatically terminates the task agent. Each agent can dynamically register its itinerary by invoking the `setRoute()` method while it is moving, but the new itinerary only becomes available after it returns to a certain agent pool.

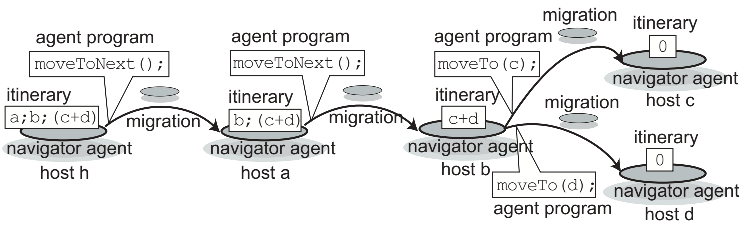

Fig. 3. Following-itinerary movement of a mobile agent whose itinerary is specified as `a;b;(c+d)`.

4.3 Task Agent

Each task agent is a mobile agent that defines its management tasks at each of the hosts in accordance with its management criteria. Although it may be able to travel among the agent pools of its target sub-networks, it is unfamiliar within each of the sub-networks. This framework provides a Java-based abstract class that allows us to easily define advanced task agents by extending the the `TaskAgent` class.

```
public class TaskAgent extends MobileAgent {
  void setRoute(Route r)
    throws IllegalSyntaxException ... { ... }
  void arrivedAt(Host here);
  void depaturingFor(Host dst);
  void finished(Route r);
  ...
}
```

Each agent defines its task in the `arrivedAt()` method. When arriving at an agent pool, a task agent gives the pool the required itinerary along which a navigator agent needs to carry itself by performing the `setRoute()` method with a term of $\mathcal{E}$ corresponding to that itinerary. The agent pool selects a suitable navigator agent and then migrates the task agent into the selected agent. Having arrived at a host, the navigator agent invokes the `arrivedAt()` method of its task agent to instruct it to do something during a given time period at the host. After receiving a certain event from all the task agents or after the period has elapsed, the navigator agent invokes the `depaturingFor()` method with the address of the next host and then moves itself and its task agents to the destination according to its itinerary. For reasons of security, all agents must be authenticated by the agent pool of a sub-network on behalf of the sub-network. This is helpful in network management systems whose hosts may have limited CPU power and memory. Since a sub-network may explicitly prohibit any task agent from visiting its hosts, task agents must be carried by a navigator agent managed by the agent pool of the sub-network. Therefore, a task agent alone cannot migrate to all the hosts, even if it knows the addresses of its target hosts in the sub-network.

4.4 Agent Pool

Each agent pool is a stationary agent, which can contain more than one navigator agent as shown in Fig. 4. It is also responsible for receiving the requirements of a visiting task agent and selecting a suitable navigator agent to carry the task agent among hosts on its sub-network. It maintains a repository database about the itineraries of idle navigator agents waiting for a chance to guide task agents. When an agent pool receives a task agent, it extracts the required itinerary from it and selects a navigator agent whose itinerary can satisfy the required itinerary from the idle navigator agents stored inside it. The selection mechanism is just a direct implementation of the algebraic relation presented in Definition 3.3. That is, the agent pool compares the required itinerary written in $\mathcal{E}$ with each of the possible itineraries of the agents written in $\mathcal{S}$ by directly using the order relation $\sqsupseteq_n \subseteq \mathcal{E} \times \mathcal{S}$ in Definition 3.3. It then assigns the task agent to the navigator agent whose itinerary can satisfy the itinerary required. If more than one navigator agent satisfies the required itinerary, it selects an agent with the least number of agent migrations over a network, which is n of $\sqsupseteq_n$ in Definition 3.3.

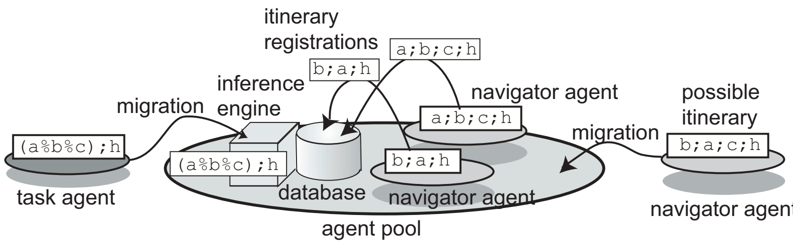

Fig. 4. Agent pool

4.5 Current Status

The cost of selecting navigator agents is dependent on the number of agents and the length of itineraries. Although the current implementation was not optimized for performance, it can handle each of all the itineraries presented in this paper within a few milliseconds. Also, the per-hop latency of migrating a simple task agent using a navigator agent whose itinerary is static is 43 ms. This is where the per-hop latency of agent migration using a non-hierarchical minimal-size agent is 35 ms in a MobileSpaces runtime system running on computers (Pentium III-800 MHz with Windows2000 and JDK 1.4) connected through a 100-Mbps Ethernet.

5 Application

We developed a network management system for a cluster computing environment consisting of two sub-networks to evaluate the effectiveness of this framework. Each of the sub-networks had from four to eight processor elements distributed geographically. [3]

[3] We presented the GRID environment in our previous paper [11] It was small scale because it was implemented as a testbed for developing middleware and applications for GRID computing rather than a computational infrastructure.

The purpose of this system was to monitor various network and computational resources at the hosts.

The system deploys an agent pool at one host of each sub-network and offers several task agents and navigator agents. For example, the task agent that monitors network traffic loads has been designed to perform its task at each host that it visits. Although the system itself is independent of any network management protocols, we constructed a task agent that can access SNMP data from a small stationary agent situated at its visiting host. The stationary agent allows that visiting task agent to access the MIB of its host via interagent communication. Since the task agent can contain codes to perform both information retrieval and filtering, it can only carry relevant information. Also, the system has three other task agents for monitoring computational resources at the processor hosts. They have been designed to collect information on the use of CPU, memory, and disks by incorporating performance monitoring systems at the hosts. Also, the system offers several navigator agents with different itineraries. However, due to word limitations, this section only explains two navigator agents optimized for one of the sub-networks defined by `NaviAgent1` and `NaviAgent2` classes.

```
 1:  public class NaviAgent1 extends NavigatorAgent {
 2:    public NaviAgent1() {
 3:      // registering its possible itinerary
 4:      setRoute(new Route("h;a;b;c;d;h"));
 5:    }
 6:    // invoked at the completion of the task agent's
 7:    // processing at the current host
 8:    public void done() throws
 9:      MultiplePossibleHostsException .. {
10:      moveToNext();
11:    }
12:    ...
13:  }
```

`NaviAgent1` can travel along a sequential route. `NaviAgent2` is defined as the same class except for the fourth line as follows:

```
 4:      setRoute(new Route("h;a;h;b;h;c;h;d;h"));
```

The above itinerary defines a star-shaped route among four hosts,`a`, `b`, `c`, and `d`. Next, suppose a task agent gathers local information from SNMP agent running on each of the hosts that it visits. The agent has its required itinerary specified as follows:

```
 1:  public class SimpleTaskAgent extends TaskAgent {
 2:    public SimpleTaskAgent() {
 3:      setRoute(new Route("h;([SNMP-AGENT]&h^*);h"));
 4:    ...
 5:    }
 6:    ...
 7:  }
```

where `[SNMP-AGENT]` is a list of the hosts that perform SNMP agents. We assumed that the list could be transformed into an itinerary (`a % b % c % d`), which means that the agent must visit the four hosts specified in `a`, `b`, `c`, and `d` in any order of movement and can visit the home host `h` more than zero times on the way. The two navigator agents can satisfy the required itinerary of the task agent. Since `NaviAgent1` has fewer agent migrations than that for `NaviAgent2`, the agent pool selects the former navigator agent and moves the task agent into it. After receiving the task agent, the `NaviAgent1` navigator agent carries it from host to host according to its own itinerary. Whenever it arrives at one of the destinations, it issues certain events to invoke the `arrived()` method of the task. The task agent performs its application-specific task, such as accessing and filtering from the SNMP agent of its visiting host, defined in the `arrived()` method.

6 Related Work

Many mobile agent systems have been developed over the last few years. There have been several attempts to develop mobile agent-based network management, for example see [2, 4]. Existing work on mobile agents has focused on the development of agent infrastructures, applications, and functions that can be used by agents, but not on approaches to selecting mobile agents. Of these, Plangent [8] is a mobile agent system, which can dynamically generate a plan to let itself acquire the knowledge that users need and then perform its application-specific actions and movements according to the plan. However, Plangent's planning functionality does not target the mobility of agents and cannot always generate valid plans. Our approach on the other hand offers a theoretical foundation for the selection of mobile agents and allows us to determine whether or not the movements of agents can satisfy the requirements of network management tasks.

Moreover, there have been various theoretical models for specifying mobile agents, e.g., Mobile UNITY [6] and Ambient calculus [3]. The former is an extension of UNITY and was designed for specifying control flows, variables, and conditional assignment statements at programs; it cannot merely extract or analyze only the itineraries of mobile components. The latter is just a theoretical framework for formalizing the whole computation of mobile agents. As far as the author knows, no existing calculi can provide any preorder relations for the selection of mobile agents according to their itineraries.

We should next compare the framework with our previous works. We presented an approach to building mobile agents for network management in our previous paper [11]. This approach separated a mobile agent into two parts: its mobility control part and its application-specific part, like the framework presented in this paper. However, the previous paper did not provide any approaches to matchmaking two parts, unlike this paper. We presented active network protocols for building and managing several agent migration protocols in our another previous paper [10], but this was just an infrastructure for configurable protocols for agent migration.

7 Conclusion

This paper presented a framework for building and operating mobile agents for network management. The framework makes two contributions to mobile and multi-agent technologies. The first is to propose an approach for building a reusable mobile agent from two subcomponents: a navigator agent and a task agent. The second contribution is to formulate a specification language and an algebraic order relation between the terms of the language as a theoretical foundation for the selection of mobile agents. It provides a matchmaking mechanism for navigator and task subcomponents. We believe that the framework itself is general-purpose so that it is available for other mobile agent-based applications as well as network management.

Finally, we would like to mention some future research directions. We plan to establish an axiomatic system based on the order relation to improve the performance of agent selection. This paper does not discuss any coordination among multiple mobile agents, but we are interested at developing a mechanism for assigning a task to one or more mobile agents. We are interesting in applying the framework to an infrastructure for ambient intelligence in ubiquitous computing environments presented in our previous paper [12] and to mobile agent-based managements for sensor networks presented in another previous paper [13].

References

1. Y. Aridor, and D.B. Lange, Agent Design Patterns: Elements of Agent Application Design, Proceedings of Second International Conference on Autonomous Agents (Agents'98), pp. 108-115, ACM Press, 1998.
2. A. Bieszczad, B. Pagurek, and T. White: Mobile Agents for Network Management, IEEE Communications Surveys, vol. 1, no. 1, 1998.
3. L. Cardelli and A. D. Gordon: Mobile Ambients, Proceedings of Foundations of Software Science and Computational Structures, LNCS, vol. 1378, pp. 140–155, 1998.
4. A. Karmouch, Mobile Software Agents for Telecommunications, IEEE Communication Magazine, vol. 36 no. 7, 1998.
5. B. D. Lange and M. Oshima: Programming and Deploying Java Mobile Agents with Aglets, Addison-Wesley, 1998.
6. P.J. McCann, and G.-C. Roman, Compositional Programming Abstractions for Mobile Computing, IEEE Transaction on Software Engineering, vol. 24, no.2, 1998.
7. R. Milner, Communication and Concurrency, Prentice Hall, 1989.
8. A. Ohsuga, Y. Nagai, Y. Irie, M. Hattori, and S. Honiden, PLANGENT: An Approach to Making Mobile Agents Intelligent, IEEE Internet Computing, vol.1, no.4, pp.55-57, 1997.
9. I. Satoh, MobileSpaces: A Framework for Building Adaptive Distributed Applications Using a Hierarchical Mobile Agent System, Proceedings of International Conference on Distributed Computing Systems (ICDCS'2000), pp.161-168, IEEE Computer Society, April, 2000.
10. I. Satoh, Network Processing of Mobile Agents, by Mobile Agents, for Mobile Agents, Proceedings of Workshop on Mobile Agents for Telecommunication Applications (MATA'2001), LNCS, vol.2146, pp.81-92, Springer, 2001.
11. I. Satoh, A Framework for Building Reusable Mobile Agents for Network Management, Proceedings of Network Operations and Managements Symposium (NOMS'2002), pp.51-64, IEEE Communication Society, April, 2002. (A long version will appear in IEEE Transaction on Systems, Man and Cybernetics, vol.33, no.3, 2003)
12. I. Satoh, Physical Mobility and Logical Mobility in Ubiquitous Computing Environments, Proceedings of Conference on Mobile Agents (MA'02), LNCS, Vol. 2535, pp.186-202, Springer, 2002.
13. T. Umezawa, I. Satoh, Y. Anzai, A Mobile Agent-based Framework for Configurable Sensor Networks, Proceedings of International Workshop on Mobile Agents for Telecommunication Applications (MATA'2002), LNCS, Vol. 2521, pp.128-140, Springer, 2002.