Building and Selecting Mobile Agents for Network Management

Ichiro Satoh

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
Tel: +81-3-4212-2546 Fax: +81-3-3556-1916
E-mail: ichiro@nii.ac.jp

Abstract

This paper presents a framework for reusable mobile agents for network management in the sense that they are independent of either particular networks or applications. The framework enables a mobile agent to be composed from two layered components, which are mobile agents. The former is a carrier of the latter over particular networks independent of any management tasks and the latter defines management tasks performed at each host independently of any networks. The framework also offers a mechanism for matchmaking the two components. Since the mechanism is formulated on a process algebra approach, it can accurately select the appropriate itinerary component to perform management tasks at hosts that the tasks want to visit over networks. The framework provides a methodology for easily developing and operating mobile agents for traveling among multiple sub-networks to perform their management tasks at all of the nodes that they visit. This paper also describes the framework, its prototype implementation, and a practical application.

Keyword: network management, mobile agent, process algebra, separation of concerns, formal specification

Short version of title: Building and Selecting Mobile Agents

1 INTRODUCTION

Mobile agent technology can play an important role in network management. Mobile agents are autonomous programs that travel between network nodes under their own control. They are not linked to the system where they start execution. After being created at a node, each mobile agent can carry its state and code to another node, where its execution can be restarted or continued. By interacting with a node after migrating to it, an agent can perform complex operations on data without transferring them, directly control equipment at the visited node, and dynamically deploy software at the nodes. This is because the agent can deploy the application logic to where it is needed and only carry relevant data rather than the entire data observed in nodes. This emerging technology is useful for distributed managements. Several researchers have attempted to apply the technology to network management [3, 4, 9, 11].

However, there have been serious problems associated with the development of mobile agent-based management systems in addition to security problems. Such systems are required to migrate their agents among all specified nodes using an efficient itinerary to perform their management tasks at all of the nodes visited, because the itineraries of agents seriously affect the achievement and efficiency of their tasks. A real network, on the other hand, often consists of numerous of sub-networks through which nodes are connected, and various of these may have some malfunctions and disconnections. Also, their topology may not be entirely known. Consequently, mobile agents for network management must be able to handle such complex and incomplete networks. However, it is almost impossible to dynamically generate an efficient itinerary among multiple nodes. 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. Such an agent optimized for particular networks cannot, however, be reused in other networks.

To solve this problem, we construct a framework for building and operating mobile agents for network management without losing reusability or efficiency. The framework separates application-specific tasks and the itineraries of mobile agents. The former defines network management tasks independent of any network 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 accurately select the agent whose itinerary can satisfy

the requirements of a network management task. The process algebra provides a theoretical foundation for specifying and reasoning about the routing of programmable packets in active networking technologies. The current implementation of the framework is built on a Java-based mobile agent system, called MobileSpaces [16]. Since our process algebra is small and executable and can specify the itineraries of mobile agents rather than application-specific tasks, it is useful for specifying and evaluating the routing of active packets.

The framework provides a contribution to active network technologies [23], in particular active packets (or so-called programmable packets). Mobile agents can be considered as a special case in mobile code technology, which is the basis for existing active network technologies. Our process algebra was designed to enable the itinerary of a mobile agent to be expressive and compact. Therefore, it is useful as a language to specify the routing of active packets, which are often required to be as small as possible to reduce network traffic. The algebraic-order relation defined in this paper can be directly used as a selection mechanism to select active packets according to their routing.

This paper is organized as follows: Section 2 presents the basic ideas behind this framework and Section 3 defines a 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 has concluding remarks.

2 APPROACH

The goal of this paper is to provide a framework for building and operating reusable mobile agents capable of autonomously traveling among nodes on multiple sub-networks to perform their management tasks at all nodes they visit on distributed or networked systems.

2.1 Mobile Agents for Network Management

Mobile agents are often treated as software agents but they are not always required to offer intelligent capabilities, for example reactive, pro-active, and social behaviors, which are features of existing software agent technologies. This is because these capabilities tend to be large both in scale and processing, while computational resources, which agents can use at the nodes they visit, such as processors, memory, files, and networks, are limited. That is, an intelligent and general-purpose agent is not appropriate in network man-

agement systems because no mobile agent should consume many computational resources at its destination. Also, all mobile agents must be as small as possible because the size of a moving agent seriously affects the cost of migrating it over a network. Therefore, mobile agent-based management systems should offer various small agents specialized to support particular tasks, rather than a few general-purpose agents to support various tasks. They should also select suitable agents to perform these tasks. For these reasons, mobile agents should be statically optimized for their target networks because both the cost of dynamically discovering an efficient itinerary and establishing the size of the program tend to be large. However, an agent optimized for particular networks or tasks cannot be reused in other networks or tasks. This inevitably results in a trade-off between the performance and reusability of a mobile agent.

2.2 Two-layered Mobile Agents

Our framework introduces two types of mobile agents: *task* agents and *navigator* agents,to solve these problem (see Figure 1).

- The *Navigator* agent has no application-specific tasks. Instead, it carries task agents and can be optimized for a particular sub-network.
- The *Task* agent is an application-specific agent that performs its management task at all nodes it visits. It can travel from sub-network to sub-network, but may not know about the sub-networks it visits.

When a task agent arrives at an unknown sub-network, it enters an idle navigator agent that knows the current network well. Then, the selected navigator agent carries the visiting task agent to the nodes that the task agent wants to visit. Each navigator agent is defined and managed by its network and can explicitly limit the nodes to which it can carry task agents.

This framework also provides a mechanism for allowing a task agent to select a navigator agent suitable for the current network. The mechanism, called Agent Pool, stores idle agents in a manner similar to that in a bus-terminal or a taxi stand (Figure 2). Each sub-network has multiple agent places for storing navigator agents specific to it. Each navigator agent is designed to return to its specified agent pool to wait for the next task soon after achieving its navigation task, because tracking moving agents and forwarding messages to them tends to be heavy or unreliable. 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 nodes in its sub-network. Therefore, for a task agent to travel among some nodes on a sub-network, it migrates to the agent pool at the sub-network and asks a navigator agent stored in the pool to carry it among the nodes. Both kinds of agents are implemented as hierarchical mobile agents in the MobileSpaces system [16].

2.3 Mobile Agent Matchmaking Mechanism

Mobile agents should generally be selected according to two criteria: their application-specific behaviors and their itineraries. Although existing task assignment or agent selection mechanisms for non-mobile software agents (see [7, 21]) may be able to deal with the former criterion, they cannot support the latter. The focus of current research on mobile agents, however, is on developing of execution platforms and applications for mobile agents. The task of selecting mobile agents has received little attention so far. Therefore, this paper proposes an approach to selecting mobile agents according to the latter criterion. The approach matchmakes between task agents and navigator agents by comparing the itineraries required by the task agents and the possible itineraries of the navigator agents. Since mobile agent programs are written in general-purpose programming languages, such as Java, it is almost impossible to only extract the mobile agents' itineraries from their programs. Therefore, our approach provides a specification language for the itineraries of mobile agents and assumes that each mobile agent explicitly specifies its own itinerary as a term of the language.

The language is formulated as an extended process algebra with the expressiveness of agent movement. Our mobile agent selection is formulated on an order relation over the terms of the language. The relation is defined based on the notion of bisimulation [14] and can compare the possible itinerary of each mobile agent and the itinerary required by a task request. It allows us to accurately determine the former itinerary can satisfy the latter itinerary or not. We implemented the relation in more than one agent pool allocated to each sub-network. When it received a task agent, it compared the itinerary of each of its stored navigator agents with the itinerary required by the request of the task agent by using this relation to select at most one suitable mobile agent to accomplish the request.

Remarks

We need to explain why our hierarchical agent model is required in the development of network management systems. The distribution of knowledge in a sub-network must be limited only to it for reasons of security

and no node should not have the capability of authenticating its visiting arbitrary agents. To visit nodes on a sub-network, in this framework, task agents must be contained and carried by navigator agents that are provided and authorized by the sub-network. All agent pools can authenticate their visiting task agents on behalf of their sub-network. As a result, each node can thus only accept pre-authorized navigator agents instead of its visiting arbitrary agents. Moreover, the knowledge of the topology of the sub-network is kept inside the navigator agents and no task agents should have such knowledge.

Our framework introduces such a knowledge component as a navigator agent, which is a container comprised of more than one task agent. To visit nodes on a sub-network, task agents must be carried by a navigator agent that is authorized by the sub-network and that has its own itinerary. Therefore, each task agent does not have to be modified and can remain autonomous and self-contained, even while it is contained in a navigator agent traveling over the sub-network. Since knowledge on the topology of the sub-network is kept inside the navigator agent, the task agent does not have access to such knowledge, unlike the two previously discussed approaches. When a network management task consists of multiple mobile agents, a navigator agent can carry these agents together as a whole. Moreover, no node has the capability of authenticating their visiting arbitrary agents. Since each agent pool can authenticate its visiting task agents on behalf of its sub-network before they are contained and carried by a navigator agent, each node can thus only accept authorized navigator agents.

Furthermore, the reader may wonder why agent itineraries should be specified in a formal approach. This is because the requirements of a task agent may often be varied or vague and the itineraries of navigator agents may be complex or large. Therefore, it is not easy to select suitable navigator agents whose itineraries can satisfy the itineraries required by task agents. Also, the reader may think agent itineraries should be passed to navigator agents as parameters written in simple conventional or executable languages, such as Lisp and Prolog. However, it is difficult to verify whether or not itineraries written in such languages are valid. Consequently, we need to construct a mechanism for selecting mobile agents based on a theoretical foundation.

3 MOBILE AGENT SELECTION

A typical mobile agent for network management systems must monitor and control various equipments at multiple nodes over a network whose topology may not be exactly known and which may suffer from malfunctions and disconnections. Such an agent often has its own static itinerary to solve problems in its target network. When a task agent is carried by a navigator agent, the performance and achievement of the task agent are dependent on the itinerary of the navigator. If a mobile agent gathers information from a node and reflects this information in other nodes, the order of its movement among nodes may affect the states of the nodes. Therefore, such an agent must migrate among the nodes according to a specified itinerary. However, if an agent can travel among nodes to aggregate interesting information from them without any writing on any of them, the order of its movement may be independent of its achievements. Moreover, an agent's itinerary is often dependent on the results of the agent's network management task. A given request may permit an agent to migrate along a traverse of all the specified nodes irrespective of the order of arrival, or along a loose route, where a loose route means that some nodes may have been omitted or visited any number of times. The language specifies such vagueness and allows agent discretion by extending itself with non-deterministic operators.

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:

$$\begin{array}{rcl} E & ::= & \mathtt{0} \mid \ell \mid E_1 \,\mathtt{;}\, E_2 \mid E_1 \,\mathtt{+}\, E_2 \\ & \mid & E_1 \,\mathtt{\#}\, E_2 \mid E_1 \,\mathtt{\%}\, E_2 \mid E_1 \,\mathtt{\&}\, E_2 \mid E\mathtt{*} \end{array}$$

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 $E_1 \,\mathtt{\#}\, E_2$, $E_1 \,\mathtt{\%}\, E_2$, $E_1 \,\mathtt{\&}\, E_2$, and $E\mathtt{*}$ 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 terms of $\mathcal{S}$, it can dynamically evaluate its itinerary and migrate itself among nodes along the itinerary. Intuitively, the meaning of constructions is as follows:

- $\mathtt{0}$ represents a terminated itinerary.
- ℓ represents agent migration to a node 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.
- $E_1 + E_2$ represents an agent moving according to either E_1 or E_2, where the selection can be explicitly performed by the processing of 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 in its itinerary.
- $E_1 \,\&\, E_2$ means that two itineraries E_1 and E_2 can be performed asynchronously.[1]
- E^* is a transitive closure of E and means that an agent can move along E for an arbitrary number of times.

To accurately express such itineraries, we need to define a specification language based on a process algebra approach such as CCS [14]. The semantics of the language is defined by the following labeled transition rules:

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

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

$$\frac{E_1 \xrightarrow{\ell} E_1'}{E_1 \,\&\, E_2 \xrightarrow{\ell} E_1' \,\&\, E_2} \qquad \frac{E_2 \xrightarrow{\ell} E_2'}{E_1 \,\&\, E_2 \xrightarrow{\ell} E_1 \,\&\, E_2'}$$

$$\frac{E_1 \xrightarrow{\tau} E_1'}{E_1 \,;\, E_2 \xrightarrow{\tau} E_1' \,;\, E_2} \qquad \frac{-}{E_1 \# E_2 \xrightarrow{\tau} E_1} \qquad \frac{-}{E_1 \# E_2 \xrightarrow{\tau} E_2} \qquad \frac{-}{E_1 \,\%\, E_2 \xrightarrow{\tau} E_1 \,;\, E_2}$$

$$\frac{-}{E_1 \,\%\, E_2 \xrightarrow{\tau} E_2 \,;\, E_1} \qquad \frac{E_1 \xrightarrow{\tau} E_1'}{E_1 + E_2 \xrightarrow{\tau} E_1' + E_2} \qquad \frac{E_2 \xrightarrow{\tau} E_2'}{E_1 + E_2 \xrightarrow{\tau} E_1 + E_2'}$$

$$\frac{E_1 \xrightarrow{\tau} E_1'}{E_1 \,\&\, E_2 \xrightarrow{\tau} E_1' \,\&\, E_2} \qquad \frac{E_2 \xrightarrow{\tau} E_2'}{E_1 \,\&\, E_2 \xrightarrow{\tau} E_1 \,\&\, E_2'}$$

where $0 \,;\, E$ is treated as being syntactically equal to E. E^* is recursively defined as $0 \# (E \,;\, E^*)$. We often abbreviate $E_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} E_n$ 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 node named ℓ and then behaves as E'. Also, if there are two possible transitions

[1] In 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.

$E \xrightarrow{\ell_1} E_1$ and $E \xrightarrow{\ell_2} E_2$ in an agent, the processing by the agent chooses one of the destinations, ℓ_1 and ℓ_2. The τ-transition, on the other hand, corresponds to a non-deterministic choice in an agent's itinerary.

Next, let us formulate an algebraic order relation based on the concept of bisimulation [14]. The relation is suitable for selecting a navigator agent whose itinerary can satisfy the requirements 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. Let us look at some basic examples.

- $(a \,\%\, b \,\%\, c)\,;\, h \sqsupseteq_4 c\,;\, a\,;\, b\,;\, h$

  where the right-side requires an agent to migrate among three nodes a, b, and c in an indefinite order and then return to node h, he right-side migrate among three nodes 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.

- $((a\,;\, b\,;\, c)\,\&\, h^{\star})\,;\, h \sqsupseteq_6 a\,;\, h\,;\, b\,;\, h\,;\, c\,;\, h$

  where the left-side allows an agent to drop in at node h an arbitrary number of times on the itinerary $a\,;\, b\,;\, c$ and then finish its movement at node h. The right-hand is a star-shaped route between three destinations, a, b, c and node h can satisfy the left-side.

4 MOBILE AGENT SYSTEM

Before describing the framework presented in this paper, let us briefly review the MobileSpaces mobile agent system that provides the infrastructure for this framework.[2]

4.1 Hierarchical Mobile Agents

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 nodes. 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 one 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 it. Therefore, an agent can contain other mobile agents inside it and carry the agents to another computer or agent together as a whole. Each agent has a globally unique name and can have more than one active thread under the control of the runtime system.

4.2 Mobile Agent Runtime System

Each MobileSpaces runtime system is a platform for executing and migrating agents. It is built on a Java virtual machine, and mobile agents are Java objects. Each runtime system can subordinate all the agents inside it, and the system maintains the life-cycle state of the agents. When the life-cycle state of an agent is changed, for example, at creation, termination, or migration, the core system issues certain events to invoke specified methods in the agent and the agents it contains. The runtime system provides a mechanism for marshaling and unmarshaling agents.[3] When an agent is marshaled, the runtime system propagates certain events to the agent and its inner agents that are still running to instruct them to stop. It can also automatically stop and serialize them after a given time. The runtime system can transfer agents to the destination computer over a TCP/IP connection.

[2]Details of the MobileSpaces mobile agent system can be found in our previous paper [16].

[3]The current implementation of the system uses the Java object serialization package provided by JDK to marshal and unmarshal agents. The package does not support the capturing of stack frames or program counters for threads. Consequently, our system cannot serialize the execution states of any thread objects.

5 DESIGN AND IMPLEMENTATION

This section presents a prototype implementation of our framework. We tried to retain the implementation within the framework as much as possible. Figure 3 shows the structure of a navigator agent containing a task agent.

5.1 Navigator Agent

Each navigator agent is a container of one or more task agents and is responsible for carrying them to the nodes 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 timings, such as arrival and departure. Each navigator agent is designed to return to its agent pool, then register its itinerary at the pool soon after completing its navigation goals, and then 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 {
  // registering an itinerary
  void setRoute(Route r) throws IllegalSyntaxException ... { ... }
  // migrating to the node specified as h
  void moveTo(Host h) throws NoSuchHostException,
    IllegalHostException .. { .. }
  // migrating to the next node specified in its itinerary
  void moveToNext() throws MultiplePossibleHostsException,
      NoSuchHostException  ... { ... }
  // asking the possible destinations in the next migration
  Host[] getPossibleHosts() ... { ... }
  ...
  // callback method invoked after the agent arrives at a destination.
  void arrivedAt(Host here);
  // callback method invoked before the agent leaves from the current node.
  void depaturingFor(Host dst);
  ...
}
```

Each navigator agent has its own itinerary as a term of $\mathcal{S}$ and registers the term with itself and its agent pool by invoking `setRoute()` as follows:

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

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

- The first approach allows each agent to move along the itinerary registered with itself. Each agent has a lightweight interpreter for the language in $\mathcal{S}$. When the agent invokes `moveToNext()`, 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 the selective operator `+`, the invocation of the method throws a `MultiplePossibleHostsException`. The agent obtains all the destinations that it can move to at the next hop by invoking `getPossibleHosts()` and moves to one of them by invoking `moveTo(dst)` 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 from Figure 4, it performs `moveToNext()` twice for two hops; from the current node to `a` and then from node `a` to `b`. Next, it can select either `c` or `d`, after which it performs `moveTo(dst)` 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 node `b` to the destination specified as its argument. For example, as you can see from Figure 5, an agent whose itinerary is `a;b;(c+d)` can invoke `moveTo()` with `a` and then `b` to move to node `a` and then to `b`. Next, it can invoke the same method with either `c` or `d`.

For reasons of security, this framework prevents 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 `setRoute()`, the agent is constrained and an `IllegalHostException` is thrown to the agent. Each navigator agent can explicitly limit the length of the execution period of its incoming task agents after arriving at each destination. When the time limit for a task agent inside it expires, it automatically terminates the agent. Each navigator agent can dynamically register its itinerary by invoking `setRoute()` while it is moving, but the new itinerary becomes available after it returns to a certain agent pool.

5.2 Task Agent

Each task agent is a mobile agent that defines its management tasks at each of the nodes in accordance with its management criteria. Although it may be able to travel among the agent pools of its target sub-networks, it is not familiar with these sub-networks. This framework provides a Java-based abstract class that allows us to easily define advanced task agents by extending the `TaskAgent` class.

```
public class TaskAgent extends MobileAgent {
  // registering its requiring itinerary
  void setRoute(Route r) throws IllegalSyntaxException ... { ... }
  // callback method invoked after the agent arrives at one of its destinations.
  void arrivedAt(Host here);
  // callback method invoked before the agent leaves from the current node.
  void depaturingFor(Host dst);
  // callback method invoked after the agent visits all the nodes in
  // its itinerary
  void finished(Route r);
  ...
}
```

The interaction between a navigator agent and the task agents inside it is based on event-based communication introduced in the Abstract Window Toolkit of JDK 1.1. A navigator agent invokes certain methods for its task agents, whenever it arrives at one of the destinations. For example, each task agent defines its task in `arrivedAt()`. When arriving at an agent pool, the task agent provides the pool with the required itinerary along which a navigator agent is required to carry itself by performing `setRoute()` with an itinerary specified in $\mathcal{E}$. The agent pool selects a suitable navigator agent and then migrates the task agent into the selected agent. When arriving at a node, the navigator agent invokes `arrivedAt()` of its task agent to instruct it to do something for a given period of time at the node. After receiving a certain event from all the task agents or after the period has elapsed, the navigator agent invokes `depaturingFor()` with the address of the next node and then moves itself and its task agents to the next destination on its itinerary. After it has traveled among all the required nodes, the navigator agent invokes `finished()`. 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 nodes may have limited CPU resources and memory. Since a sub-network may explicitly prohibit a task agent from visiting its nodes, task agents must be carried by a navigator agent managed by the agent pool of the sub-network. Therefore, a task agent cannot migrate to all the nodes by itself, even if it knows the addresses of the target nodes in its sub-network.

5.3 Agent Pool

Each agent pool is a stationary agent that can contain more than one navigator agent (see Figure 6). It is also responsible for receiving the requirements of visiting task agents and selecting a suitable navigator agent to carry a task agent around the nodes on its sub-network. Here, let us explain the selection algorithm for the current implementation, which we tried to make as faithful to Definition 3.3 as possible. Each agent pool maintains a repository database containing the possible itineraries of its idle navigator agents waiting for the chance to guide task agents. To reduce the cost of the selection algorithm, the possible itineraries written in $\mathcal{E}$ are transformed into tree structures, which are called `transition trees` or `derivation trees` in the literature on process algebra [14], before they are stored in the database. Each tree is derived from an itinerary in $\mathcal{E}$ according to Definition 3.2 and consists of arcs corresponding to ℓ-transitions or τ-transitions in the itinerary. When an agent pool receives a task agent, it extracts the required itinerary written in $\mathcal{S}$ from the task agent and then transforms the itinerary into a transition tree. Next, it determines whether or not the trees derived from the possible itineraries of its stored navigator agents can satisfy the tree derived from the required itinerary by matching the two trees according to the definition of the order relation ($\sqsupseteq_n \subseteq \mathcal{E} \times \mathcal{S}$) as follows:

(1) If each node in one of the two trees has arcs corresponding to ℓ-transitions, then the corresponding node in the other tree can have the same arcs, and the sub-nodes derived through the two trees' matching arcs can still satisfy either (1) or (2).

(2) If each node in the tree derived from the required itinerary has one or more arcs corresponding to τ-transitions, then at least one of the nodes derived through the arcs and the corresponding node in the tree derived from the agent's itinerary can still satisfy (1) or (2).

(3) If neither (1) nor (2) is satisfied, the agent pool backtracks from the current nodes in the two trees and tries to apply (1) or (2) to their two backtracked nodes.

The agent pool assigns the task agent to the navigator agent whose itinerary can satisfy the above conditions. If more than one navigator agent satisfies the required itinerary, it selects the agent with the least number of agent migrations over a network, which is n of $\sqsubseteq_n$ in Definition 3.3. The current algorithm for agent selection in agent pools was not optimized for performance. Although the cost of selecting navigator agents

is dependent on the number of agents and the length of itineraries, it can handle each of the itineraries presented in this paper within a few milliseconds.

6 APPLICATIONS

To explain the utility of the framework, let us describe an application of the framework. The application is a network management system for a cluster computing environment consisting of three sub-networks and each of the sub-networks has from four to eight processor elements distributed geographically.[4] The management system deploys agent pools at one cluster of each sub-network and offers several task agents and navigator agents. Since each task agent can contain codes to perform both information retrieval and filtering, it can only carry relevant information. We implemented various task agents, which collected information on the use of CPU and memory and traffic of network by incorporating performance monitoring systems at the clusters. Although the system itself is independent of any network management protocols, we constructed a task agent that could access SNMP data from a small stationary agent situated at its visiting cluster. The stationary agent allows a visiting task agent to access the MIB of its cluster via interagent communication. For example, a task agent that monitors network traffic loads is designed to perform its task at each cluster that it visits. The system also provides more than twenty navigator agents with different itineraries. The agents are statically optimized for the topology of their target sub-networks so that they can efficiently travel among the clusters in the sub-networks.

The system deploys an agent pool at one host of each sub-network and offers several task agents and navigator agents we can see from Figure 7. For example, a task agent that monitors network traffic load is designed to perform its task at each cluster it visits. Although the system itself is independent of any network management protocols, we constructed a task agent that could access SNMP data from a small stationary agent located at its visiting cluster. The stationary agent allows a visiting task agent to access the MIB of its cluster via interagent communication. Since the task agent can contain code to perform both information retrieval and filtering, it carries only relevant information. In addition, the system has three other task agents to monitor computational resources at clusters. They are designed to collect information on the use of CPU, memory, and disks by incorporating performance monitoring systems at the clusters. The system also offers

[4]The environment was small in scale because it is implemented as a testbed for developing middleware and applications for Grid or cluster computing rather than as a computational infrastructure.

several navigator agents with different itineraries. However, due to a lack of space, this section only illustrates two navigator agents optimized for one of the sub-networks defined by `NaviAgent1` and `NaviAgent2` classes. `NaviAgent1` can travel along a sequential route, `h;a;b;c;d;h`.

```
public class NaviAgent1 extends NavigatorAgent {
  public NaviAgent1() {
    // registering its possible itinerary
    setRoute(new Route("h;a;b;c;d;h"));
  }
  // invoked at the completion of the task
  // agent's processing at the current cluster
  public void done() throws MultiplePossibleHostsException .. {
    moveToNext();
  }
  ...
}
```

`NaviAgent2` can move along a star-shaped route, `h;a;h;b;h;c; h;d;h`.

```
public class NaviAgent2 extends NavigatorAgent {
  public NaviAgent2() {
    setRoute(new Route("h;a;h;b;h;c;h;d;h"));
  }
  public void done() throws MultiplePossibleHostsException .. {
    moveToNext();
  }
  ...
}
```

Next, let us consider a task agent, which gathers local information from the SNMP agent running on each of the clusters that it visits. The agent has its required itinerary specified as

```
h;Tour($(SNMP-AGENT)&h^*);h
```

where `h^*` denotes h^* in the language $\mathcal{E}$ and `SNMP-AGENT` specifies $[a, b, c, d]$ as a list of the clusters that offers snmp agents on the sub-network. When an agent pool receives the task agent, it selects a suitable idle navigator agent whose possible itinerary can satisfy the required itinerary of the task agent according to the algorithm presented in Section 5. In this example, the two navigator agents can satisfy the required itinerary of the task agent. Since the number of agent migrations for `NaviAgent1` is less 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 cluster to cluster according to its own itinerary. Whenever it arrives at one of the destinations, it issues certain events to invoke `arrived()`

for the task. The task agent performs its application-specific task, such as accessing and filtering from the SNMP agent of its visiting cluster, as defined in `arrived()`.

We have obtained a preliminarily measured the cost of migrating a navigator agent over a sub-network of the cluster system. Note that the system is just a prototype implementation; hence it has not been optimized for efficient agent migration. Actually, the total size of the navigator agent containing one of the task agents is about 8 KB (zip-compressed) and this was only 20 percent more than that for a self-contained task agent that controlled its own itinerary. This is only a small increase if we take into account the amount of data such agents can collect from clusters. The cost of detecting a navigator agent in an agent pool is less than 10 msec, although the current algorithm for agent selection in agent pools was not optimized for performance.

The total cost of management depends on application-specific tasks performed at clusters rather than agent migration. After receiving a task agent at the agent pool of the sub-network, the navigator agent travels directly around four clusters and then returns to the agent pool of the sub-network, where the clusters and the pool are Pentium III, 800-MHz computers connected using a 100-Mbps Ethernet. The itinerary of the navigator agent is statically defined and corresponds to five hops. The round-trip time of the agent is about 480 msec where the per-hop latency of agent migration for the task agent using the navigator agent is at most 25 percent greater than the per-hop latency of a self-contained task agent.

Although it is difficult to quantitatively compare between this approach and other approaches, the former has several advantages, which the latter does not have. Our early experience with this system suggests that the framework presented in this paper enables each task agent to be built independently of any sub-network and to move efficiently among multiple nodes by using navigator agents. By dynamically changing to a navigator agent suitable for the current sub-network, a task agent can efficiently migrate among nodes in various sub-networks to perform its task, without modifying its own program. However, existing mobile agent-based approaches, tend to depend on their target networks, because their agents are often designed for these. Our system also enables both navigator and task agents to be small and simple, because navigator agents do not have to offer adaptive mechanisms for handling various networks and task agents do not contain specific knowledge about sub-networks; they only have to know the location of the agent pools of their destinations. Moreover, the framework can accurately select one of the most suitable navigator agents, since it provides a theoretical and practical mechanism for comparing the itineraries of the navigator agents. Our experience

tells us that our navigator agents are useful in managing the resources of networked systems, including cluster computing environments, because they can provide a decentralized mechanism for deploying computational tasks at remote nodes. As a result, the performance of our framework is scalable in the number of nodes, whereas existing approaches have performance bottlenecks in their centralized control servers. That is, it is natural to expect that the system will still be scalable even when applying it to larger networks.

7 RELATED WORK

There have been several research projects to develop mobile agent-based network management [3, 4, 9, 11]. Current works on mobile agents, however, focused on the development of agent infrastructure, applications, and functions that can be used by agents, but not on approaches to selecting mobile agents. In fact, most existing systems have been constructed in an ad-hoc manner or are dependent on their target networks. Several researches provides policy-based mechanisms for controlling mobile agents. Nevertheless, the tasks of building and operating mobile agents have received little attention so far, although creating and operating such agents can be tedious and susceptible to errors. In this researches, ADK [10] is notable because it can separate the travel itinerary of an agent from its behavior as our approach does. Aglets [12] introduces the notion of an itinerary pattern, which is similar to design patterns in software engineering, to shift the responsibility for navigation from an application-specific agent to a framework library described in [1]. Both approaches allow us to design an application-specific itinerary for an agent independent of the agent's logical behavior, but the itinerary components must be statically and manually embedded in the agent. Consequently, this agent, unlike ours, cannot dynamically change its itinerary and cannot travel beyond its networks familiar to it. Mobile agents can be considered as a special case in mobile code technology, which is the basis of existing active network technologies [23]. There have been numerous attempts to apply mobile agent technology to the development of active networking, in particular active node techniques [2, 5]. However, most of these existing attempts to have targeted active node techniques in active networking instead of active packet techniques.

Several papers have explored theoretical models for reasoning about mobile agents, for example, Mobile UNITY [13], Join calculus [8], Ambient calculus [6], Distributed π-calculus [15], and Nomadic π-calculus [22]. Mobile UNITY is an extension of UNITY, which is an existing formal model for specifying distributed

systems, with the expressiveness of moving components, including mobile computers and mobile software. Since it has been designed for specifying variable and conditional assignment statements in programs by incorporating with UNITY, it cannot extract or reason about the itineraries of mobile components. Most existing formal models for mobile agents are based on process calculi (called process algebras), like ours. Ambient calculus [6] allows mobile agents (called ambients in the calculus) to contain other agents and to move with all inner ambients. The calculus must always model the mobility of agents as a navigation along a hierarchy of agents, whereas the itineraries of real mobile agents may be more complicated. Join-calculus [8] also introduces the notion of named locations that form a tree and the mobility of an agent is modeled as a transformation of subtrees from one part of the tree to another. Distributed π-calculus and Nomadic π-calculus are extensions of π-calculus with the notion of locations. Existing process calculus-based models are just theoretical frameworks for reasoning about the whole computation of mobile agents. As far as this author knows, no existing calculi have provided any preorder relations for mobile agents.

We should now compare the approach presented in this paper with our previous work. We presented a framework for building reusable mobile agents for network management in other papers [17, 18]. The framework separated a mobile agent into two: mobility control and application-specific parts and they were presented as two layered agents in this paper. However, it did not provide any specification language nor any mechanism for selecting the proper itineraries of mobile agents, unlike the framework presented in this paper. We also presented an approach for managing cluster or grid computing in another paper [19]. The approach was just an extension of the previous framework [17, 18] and aimed at updating software in computing clusters, unlike the framework presented in this paper. Lastly, we should describe an approach to building configurable protocols for agent migration in another paper [20]. While that approach customizes network processing for agent migration embedded in a mobile agent runtime system, the approach presented in this paper enables application-specific agents to dynamically select itineraries from multiple clusters according to the topology of the current network and the requirement of application-specific tasks.

8 CONCLUSION

This paper presented a methodology for building and operating reusable mobile agents for distributed network management. The methodology has two key ideas. The first is to compose a mobile agent from two

layered components, where the lower layer components carry upper layered components between hosts following their own itineraries optimized for their target sub-networks and the upper layer components define a set of management tasks to be performed at each of the nodes to be visited. The second idea is to provide a matchmaking mechanism between the two layer components. The mechanism is formulated based on a process algebra-based language and an algebraic order relation between the terms of the language. The language can specify the possible itineraries of lower layer components and the requiring itineraries of upper layer components. The relation can strictly decide whether or not the possible itinerary of each lower layer component can satisfy the itinerary required by an upper layer component or given request. When an upper layer component arrives at a sub-network, the approach can strictly and automatically select a suitable lower layer component according to the requirement of the visiting upper layer component. A prototype implementation system based on methodology has been constructed on a Java-based mobile agent system and applied to our experimental cluster computing system to demonstrate the effectiveness of the methodology. We believe that the system is practical in deploying and upgrading software at nodes, including routers and gateways, as well as monitoring nodes and networks. The language is useful to specify the routings of active packets, because it is small and designed for specifying only the itinerary of mobile software, including mobile agents and active packets. The algebraic order relation provides a selection mechanism for active packets according to their routings.

Finally, we would like to mention some future research directions. This paper does not discuss any coordination among multiple mobile agents, but we are interested in developing a mechanism for assigning a task to one or more navigator agents. Also, we plan to establish an axiomatic system based on the order relation, which could improve the performance of the agent selection. The performance of the current implementation is not yet satisfactory, so further measurements and optimizations are needed. We plan to design another scheme to perform security and access control specified to mobile agent-based network management.

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), ACM Press, pp. 108-115, 1998

[2] C. Baumer, and T. Magedanz, The Grasshopper Mobile Agent Platform Enabling Short-Term Active Broadband Intelligent Network Implementation, Proceedings of International Working Conference on Active Networks, pp.109-116, LNCS, Vol.1653, Springer, 1999.

[3] A. Bieszczad, B. Pagurek, and T. White, Mobile Agents for Network Management, IEEE Communications Surveys, Vol. 1, No. 1, 1998.

[4] C. Bohoris, G. Pavlou, and H. Cruickshank, Using Mobile Agents for Network Performance Management, in Proceedings of IEEE/IFIP Network Operations and Management Symposium, pp. 637-652. April 2000.

[5] I. Busse, S. Covaci, and A. Leichsenring, Autonomy and Decentralization in Active Networks: A Case Study for Mobile Agents, Proceedings of Working Conference on Active Networks, pp.165-179, LNCS, Vol.1653, Springer, 1999.

[6] L. Cardelli and A. D. Gordon, Mobile Ambients, Proceedings of Foundations of Software Science and Computational Structures, LNCS, Vol. 1378, pp. 140-155, 1998.

[7] T. Finin, Y. Labrou, and J. Mayfield, KQML as An Agent Communication Language, in Software Agents, MIT Press, 1997.

[8] C. Fournet, G. Gonthier, J. Levy, L. Marnaget, and D. Remy, A Calculus of Mobile Agents, Proceedings of CONCUR'96, LNCS, Vol. 1119, pp.406-421, Springer, 1996.

[9] D. Gavalas, D. Greenwood, M. Ghanbari, and M. O'Mahony, An Infrastructure for Distributed and Dynamic Network Management based on Mobile Agent Technology, Proceedings of Conference on Communications, pp. 1362-1366, 1999.

[10] T. Gschwind, M. Feridun, and S. Pleisch, ADK: Building Mobile Agents for Network and System Management from Reusable Components, Proceedings of Symposium on Agent Systems and Applications / Symposium on Mobile Agents (ASA/MA'99), pp.13-21, IEEE Computer Society, 1999.

[11] A. Karmouch, Mobile Software Agents for Telecommunications, IEEE Communication Magazine, Vol. 36 No. 7, 1998.

[12] B. D. Lange and M. Oshima: Programming and Deploying Java Mobile Agents with Aglets, Addison-Wesley, 1998.

[13] P.J. McCann, and G.-C. Roman, Compositional Programming Abstractions for Mobile Computing, IEEE Transaction on Software Engineering, Vol. 24, No.2, 1998.

[14] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[15] J. Riely and M. Hennessy, Distributed Processes and Location Failures, ICALP'97, LNCS, Vol. 1256, pp.471-481, Springer, 1997.

[16] 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.

[17] 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.

[18] I. Satoh, Building Reusable Mobile Agents for Network Management, to appear in IEEE Transactions on Systems, Man and Cybernetics, Vol.33, No. 3 (Accepted), October 2003.

[19] I. Satoh, Reusable Mobile Agents for Cluster Computing, to appear in Proceedings of IEEE International Conference on Cluster Computing (Cluster'2003), IEEE Computer Society, December 2003.

[20] I. Satoh, Configurable Network Processing for Mobile Agents on the Internet, to appear in Cluster Computing, Vol. 7, No. 1 (Accepted), Kluwer, January 2004.

[21] R. G. Smith, The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver, IEEE Transactions on Computers, pp.1104-1113, 1980.

[22] P. Swell, P. T. Wojciechowski, and B. C. Pierce, Location-Independent Communication for Mobile Agents: A Two-Level Architecture, Workshop on Internet Programming Languages, LNCS, Vol. 1686, Springer, 1998.

[23] D. L. Tennenhouse et al., A Survey of Active Network Research, IEEE Communication Magazine, Vol. 35, No. 1, 1997.

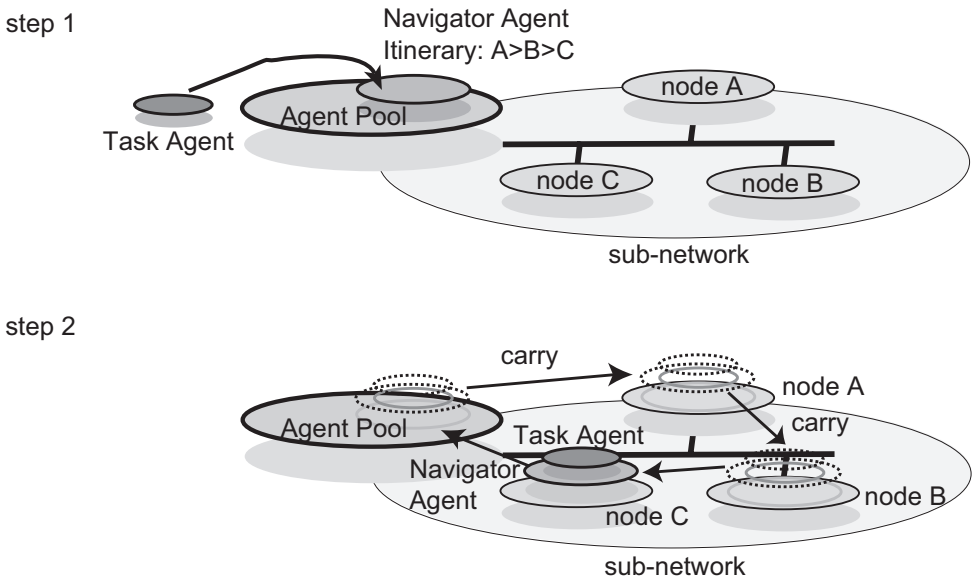

Figure 1: Navigator agents and task agents.

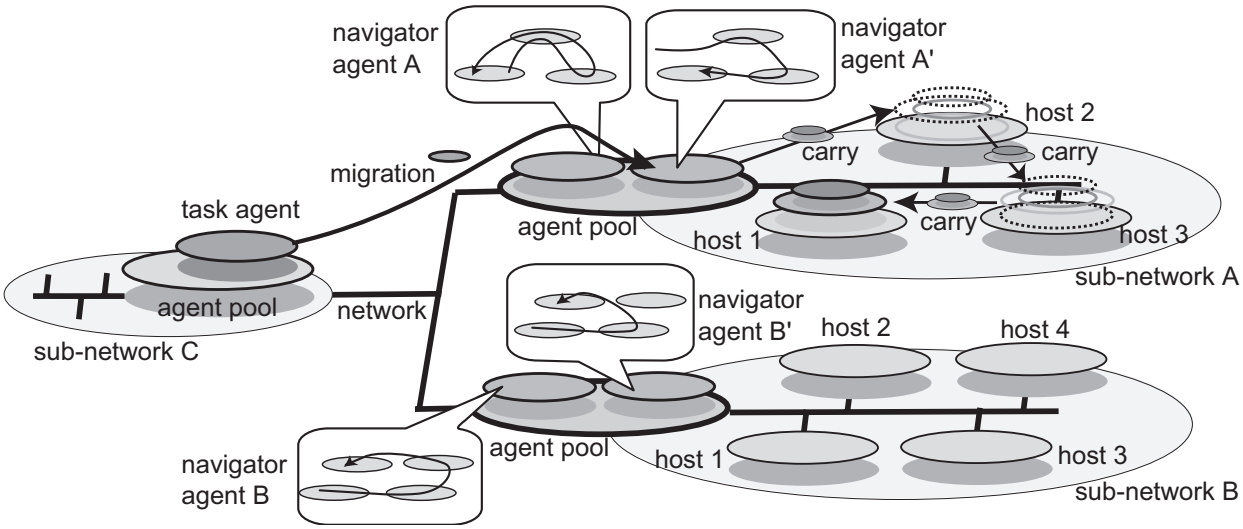

Figure 2: Agent Pools.

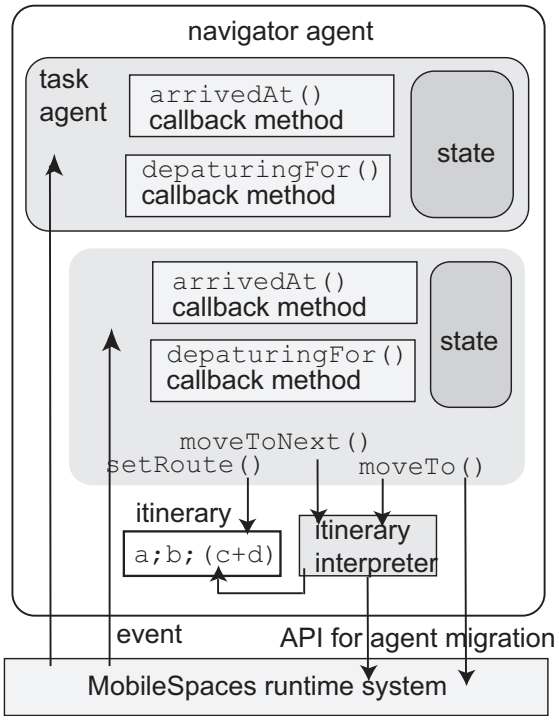

Figure 3: Structure of navigator agent.

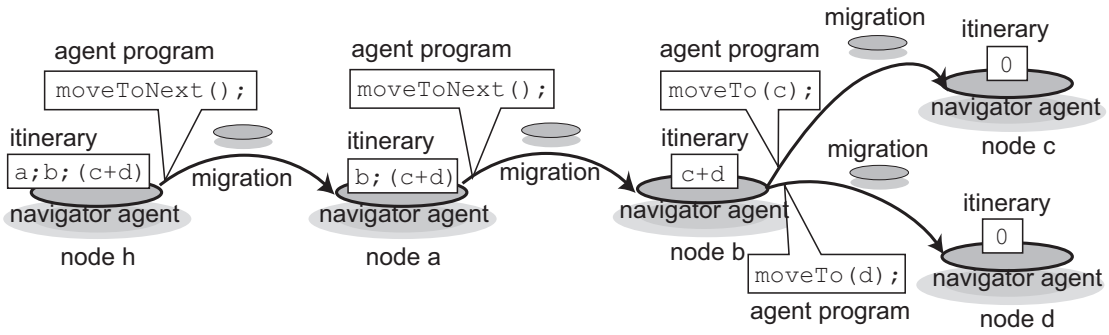

Figure 4: Following-itinerary movement of mobile agent with itinerary specified as `a;b;(c+d)`.

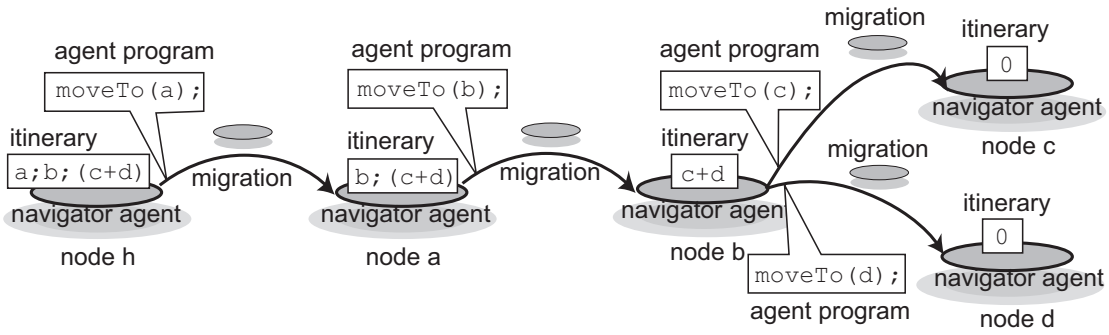

Figure 5: Following-itinerary movement of mobile agent with itinerary specified as `a;b;(c+d)`.

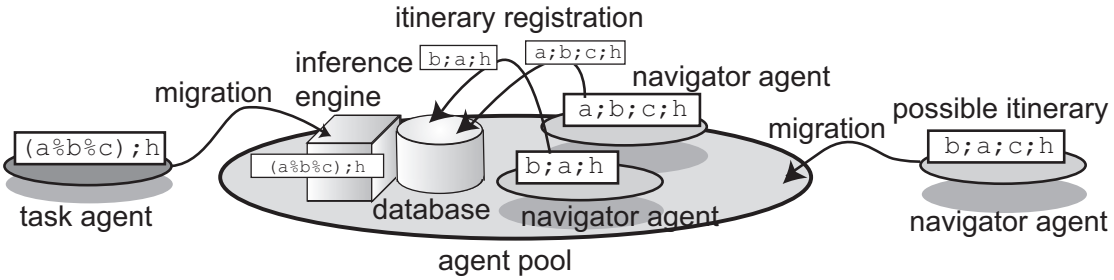

Figure 6: Agent pool

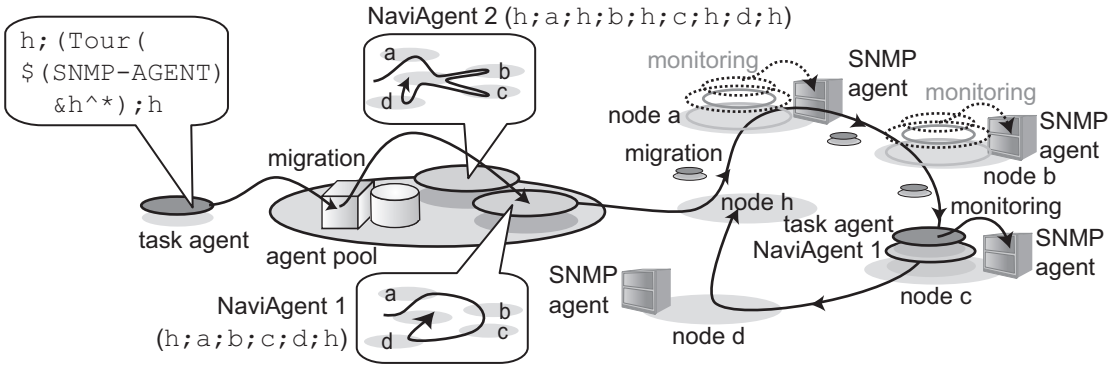

Figure 7: Mobile agent-based management system