Reusable Mobile Agents for Cluster Computing

Ichiro Satoh
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
E-mail: ichiro@nii.ac.jp

Abstract

Adopting mobile agent technology can eliminate the need for the administrator to manage clusters, e.g., installation and upgrading of software, and auditing of clusters and networks. However, creating mobile agent-based network management systems is still in an ad-hoc manner but not on methodologies for building mobile agents for cluster and Grid computing. This paper presents a framework for reusable mobile agents for managing clusters in the sense that they are independent of either particular cluster systems or applications. The framework enables a mobile agent to be composed from two layered components enables, which are mobile agents. The former is a carrier of the latter over particular networks independently 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 based on a process algebra approach, it can strictly select an itinerary component suitable to perform management tasks at the 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 clusters to perform their management tasks at each of the clusters that they visit.

1. Introduction

Mobile agent technology can play an important role in the management of cluster computing environments. Mobile agents are autonomous programs that travel from cluster to cluster under their own control. They are not linked to the system where they start their execution. After being created at an execution cluster, each mobile agent can carry its state and code to another cluster, where its execution can be restarted or continued. By interacting with a cluster after migrating to it, an agent can perform complex operations on data without transferring them, directly control equipments of the visited cluster, and dynamically deploy software at the clusters, because the agent can deploy the application logic to where it is needed and carry only relevant data rather than the whole data observed in clusters. This emerging technology is useful for managing cluster computing systems. Several researchers have attempted to apply the technology to the management of cluster and Grid computing systems.

However, there has been a serious problem associated with the development of mobile agent-based management systems for cluster computing in addition to security problems. Such systems are required to migrate their agents among all specified clusters along an efficient itinerary suitable to perform their management tasks at each of the visited clusters, because the itineraries of agents seriously affect the achievement and efficiency of their tasks. On the other hand, the network of a cluster computing system often consists of a lot of sub-networks through which computers are connected, and some of the sub-networks may have some malfunctions and disconnections. Also their topology may not be exactly known. That is, management agents for clusters must be able to handle such complicated and incomplete networks. However, it is almost impossible to dynamically generate an efficient itinerary among multiple clusters. As a result, many existing mobile agent-based network management systems for cluster computing systems or other networked systems explicitly and implicitly assume that their mobile agents are statically designed for particular itineraries over their target networks. However, such an agent optimized for particular networks cannot be reused in other networks.

To solve the above problem, we construct a framework for building and operating mobile agents for network management without losing their reusability and efficiency in cluster computing environment. The framework separates the application-specific tasks and itineraries of mobile agents. The former parts define network management tasks independently of any networks and the latter parts can be optimized for particular networks. The framework also offers a mechanism for matchmaking between the two parts. Since the mechanism is formulated based on an extended process algebra for reasoning about the itineraries of mobile agents, it can strictly select an appropriate itinerary that can satisfy the requirement of a network management task. The current implementation of the framework is built on a Java-based mobile agent system, called MobileSpaces [11].

This paper is organized as follows: Section 2 presents the basic ideas of 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 makes some 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 clusters on multiple sub-networks to perform their management tasks at each cluster they visit on cluster computing systems.

2.1. Mobile Agent-based Management for Cluster Computing

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 in scale and processing, while computational resources, which agents can use at its visiting clusters, such as processors, memory, files, and networks are limited. That is, an intelligent and general-purpose agent is not appropriate in the management of cluster computing systems because each mobile agent should not consume many computational resources at its destinations. Also, each mobile agent must be made 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 for supporting their particular tasks, rather than a few general-purpose agents for supporting various tasks, and they should select suitable agents to perform the tasks. For the same reason, as mentioned previously, mobile agents should be statically optimized for their target networks because both the cost of dynamically discovering an efficient itinerary and the size of its program tend to be large. However, an agent optimized for particular networks or tasks cannot be reused in other networks or tasks. This results in an inevitable trade-off between the performance and reusability of a mobile agent.

2.2. Two-layered Mobile Agents

To solve the above mentioned problem, the framework introduces two types of mobile agents: *task* agents and *navigator* agents, as shown in Figure 1.

- The *Navigator* agent does not have any 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 each of the clusters it visits. It can travel from sub-network to sub-network, but may not know the sub-networks it visits.

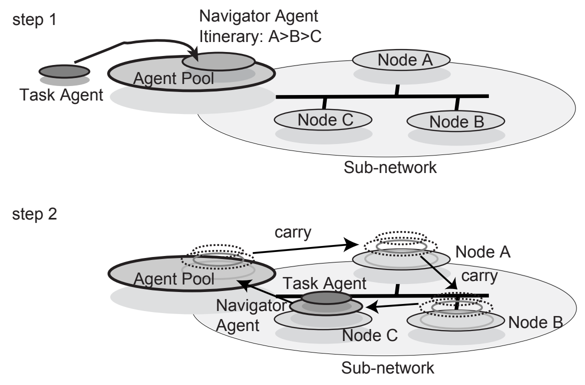

Figure 1. Navigator agents and task agents

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 clusters that the task agent wants to visit. Each navigator agent is defined and managed by its network and can explicitly limit the clusters 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, as shown in Figure 2. Each sub-network has multiple agent places for storing navigator agents specific to the sub-network. 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 tend to be heavyweight or unreliable functions. 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 clusters in its sub-network. Therefore, to travel among some of the clusters on a sub-network, a task agent migrates to the agent pool at the sub-network and asks a navigator agent stored in the pool to carry it among the clusters. Both kinds of agent are implemented as hierarchical mobile agents in the MobileSpaces system [11].

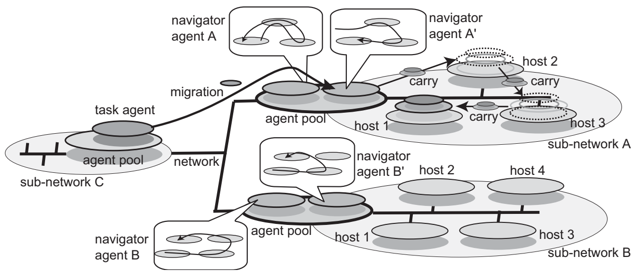

Figure 2. Agent Pools

2.3. Mobile Agent Matchmaking Mechanism

Mobile agents should be generally selected according to two criteria: their application-specific behaviors and their itineraries. Existing task assignment or agent selection mechanisms for non-mobile software agents (for example, see [4, 15]) may be able to deal with the former criterion but cannot support the latter. The focus of current researches on mobile agents, however, is on the development 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 for 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 agents' programs are written in general-purpose programming languages, such as Java, it is almost impossible to extract only the mobile agents' itinerary 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 based on an order relation over the terms of the language. The relation is defined based on the notion of bisimulation [8] and can compare the possible itinerary of each mobile agent and the itinerary required by a task request. It allows us to strictly judge whether or not the former itinerary can satisfy the latter itinerary. We implemented the relation in more than one agent pool allocated to each sub-network. When it receives a task agent, it compares the itinerary of each of its stored navigator agents with the itinerary required by the request of the task agent by using the relation to select at most one suitable mobile agent to accomplish the request.

Remarks We should give some supplemental explanations. We should explain why our hierarchical agent model is needed in the development of network management on cluster computing systems. The distribution of knowledge of the sub-network must be limited to the sub-network for reasons of security and all clusters may not have the capability of authenticating their visiting arbitrary agents. In this framework, to visit clusters on a sub-network, task agents must be contained and carried by navigator agents that are provided and authorized by the sub-network. Each agent pool can authenticate its visiting task agents on behalf of its sub-network. As a result, each cluster can thus accept only 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 any task agents does not access to and cannot have such knowledge.

Furthermore, the reader may wonder why agent itineraries should be specified in a formal approach. This is because the requirement of a task agent may be often various and vague and the itineraries of navigator agents may be complex and 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 management in cluster computing systems must monitor and control some equipments at multiple clusters over a network whose topology may not be exactly known and which may have some malfunctions and disconnections. Such an agent often has its own itinerary statically 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 is dependent on the itinerary of the navigator. If a mobile agent gathers information from a cluster and reflects the information on other clusters, its movement order among clusters may affect the states of the clusters. Therefore, such an agent must migrate among the clusters according to a specified itinerary. On the other hand, if an agent can travel among clusters to aggregate its interesting information from the clusters without any writing on any clusters, the order of its 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. A given request may permit an agent to migrate along a traversal of all the specified clusters irrespective of the arrival order, or along a loose route, where a loose route means that some clusters may be omitted or visited any number of times. The language specifies such vagueness and allows the agents' 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:

$$\begin{array}{lllllllll} E & ::= & 0 & \mid & \ell & \mid & E_1 \mathbin{;} E_2 & \mid & E_1 + E_2 \\ & \mid & E_1 \mathbin{\#} E_2 & \mid & E_1 \mathbin{\%} E_2 & \mid & E_1 \mathbin{\&} E_2 & \mid & E^\star \end{array}$$

where $\mathcal{L}$ is the set of location names, ranged over by $\ell, \ell_1, \ell_2, \ldots$. We often omit 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 terms of $\mathcal{S}$, it can dynamically evaluate its itinerary and migrate itself among clusters along the itinerary. Intuitively, the meaning of constructions are as follows:

- 0 represents a terminated itinerary.
- ℓ represents agent migration to the cluster 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 that an agent moves 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 as its itinerary.
- $E_1 \& E_2$ means that two itineraries E_1 and E_2 can be performed asynchronously.[1]
- $E^\star$ is a transitive closure of E and means that an agent can move along E in an arbitrary number of times.

To strictly express such itineraries, we here define a specification language based on a process algebra approach such as CCS [8]. The semantics of the language is defined as the following labeled transition rules:

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

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

$$\frac{E_1 \xrightarrow{\ell} E_1'}{E_1 \mathbin{\&} E_2 \xrightarrow{\ell} E_1' \mathbin{\&} E_2} \qquad \frac{-}{E_1 \mathbin{\#} E_2 \xrightarrow{\tau} E_1}$$

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

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

where $+$, $\&$, $\#$, and $\%$ are symmentric binary-operators[2] and $0 \mathbin{;} E$ is treated to be syntactically equal to E and $E^\star$ is recursively defined as $0 \mathbin{\#} (E \mathbin{;} E^\star)$. 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 cluster 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 processing of the agent choose one of the destinations ℓ_1 and ℓ_2. On the other hand, the τ-transition corresponds to a non-deterministic choice in an agent's itinerary.

Next, we formulate an algebraic order relation based on the concept of bisimulation [8]. 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 number, 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. We show some basic examples.

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.

2 For example, $E_1 + E_2$ is equal to $E_2 + E_1$.

- $(a \mathbin{\%} b \mathbin{\%} c) \,;\, h \sqsupseteq_4 c \,;\, a \,;\, b \,;\, h$
  where the right side requires an agent to migrate among three clusters a, b, and c in an indefinite order and then return to cluster h and the right side migrate among three clusters 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 cluster h in arbitrary times on the itinerary $a \,;\, b \,;\, c$ and then finish its movement at cluster h. The right is a star-shaped route between three destinations, a, b, c and cluster h can satisfy the left side.

4. Mobile Agent System

Before describing the framework presented in this paper, we briefly review the MobileSpaces mobile agent system that provides the infrastructure for this framework.[3]

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 clusters. 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 carry these agents to another computer or agent 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 certain methods in the agent and the agents it contains. The runtime system provides a mechanism for marshaling and unmarshaling agents.[4] 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 also can automatically stop and serialize them after a given time period. The runtime system can transfer agents to the destination computer over TCP/IP connection.

3 Details of the MobileSpaces mobile agent system can be found in our previous paper [11].

4 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 capturing the stack frames or a program counter of 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 keep the implementation within the framework as much as possible. Fig. 3 shows the structure of a navigator agent containing a task agent.

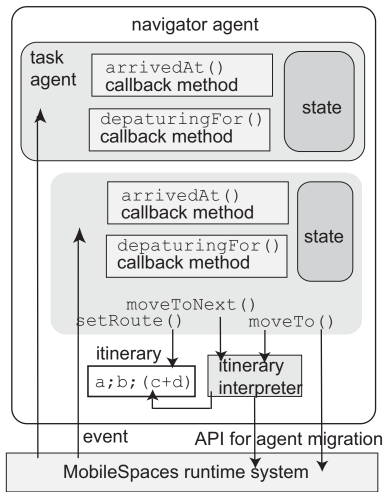
navigator agent
task agent
arrivedAt() callback method
depaturingFor() callback method
state
arrivedAt() callback method
depaturingFor() callback method
state
moveToNext()
setRoute()
moveTo()
itinerary
a;b;(c+d)
itinerary interpreter
event
API for agent migration
MobileSpaces runtime system

Figure 3. Structure of navigator 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 clusters 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 go back to its agent pool and 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 cluster specified as h
  void moveTo(Host h) throws NoSuchHostException,
    IllegalHostException .. { .. }
  // migrating to the next cluster 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 cluster.
  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 the `setRoute()` method as follows:

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

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

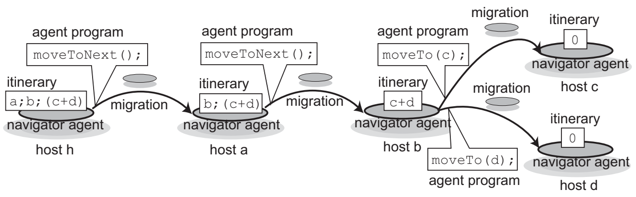

Figure 4. Following-itinerary movement of a mobile agent with itinerary specified as `a;b;(c+d)`.

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 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 the selective operator `+`, the invocation of the method throws a `MultiplePossibleHostsException`. The agent gets all the destinations that it can move to at the next hop by invoking the `getPossibleHosts()` method and moves to one of them 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 shown in Fig. 4, it performs the `moveToNext()` method twice times for two hops; from the current cluster to `a` and then from cluster `a` to `b`. Next, it can select either `c` or `d`, after which it performs 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 cluster `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 cluster `a` and then to `b`. Next, it can invoke the same method with either `c` or `d`.

For the reason of security, this framework prevents navigator agents from straying from the itinerary they registered with themselves. In both of the above approaches, when the movement of a mobile agent deviates from the itinerary registered by invoking the `setRoute()` method, 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 of a task agent inside it expires, it automatically terminates the agent. Each navigator agent can dynamically register its itinerary by invoking the `setRoute()` method 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 clusters in accordance with its management criterion. Although it may be able to travel among the agent pools of its target sub-networks, it is unfamiliar with 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 `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 cluster.
  void depaturingFor(Host dst);
  // callback method invoked after the agent
  // visits all the clusters 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 of its task agents, whenever it arrives at one of the destinations. For example, each task agent defines its task in the `arrivedAt()` method. When arriving at an agent pool, the task agent gives the pool the required itinerary along which a navigator agent is required to carry itself by performing the `setRoute()` method 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. Upon arrival at a cluster, the navigator agent invokes the `arrivedAt()` method of its task agent to instruct it to do something for a given time period at the cluster. 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 cluster and then moves itself and its task agents to the next destination on its itinerary. After it has traveled among all the required clusters, the navigator agent invokes its `finished` method. 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 clusters may have limited CPU power and memory. Since a sub-network may explicitly prohibit any task agent from visiting its clusters, 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 clusters, even if it knows the addresses of its target clusters in the sub-network.

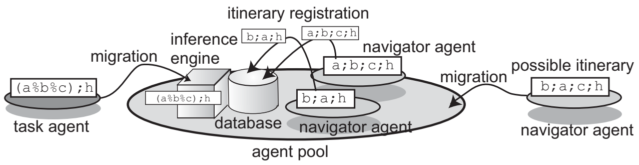

Figure 5. Agent pool

5.3. Agent Pool

Each agent pool is a stationary agent that can contain more than one navigator agent as shown in Fig. 5. It is also responsible for receiving the requirements of the visiting task agents and selecting a suitable navigator agent to carry the task agent around the clusters on its sub-network. Here, we explain the selection algorithm for the current implementation. The algorithm tried to be as faithful to Definition 3.3 as possible. Each agent pool maintains a repository database containing the possible itineraries of its idle navigator agents awaiting 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 of process algebra [8], 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 judges 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 subnodes 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. The cost of selecting navigator agents is dependent on the number of agents and the length of itineraries, but it can handle each of the itineraries presented in this paper within a few milliseconds.

6. Application

To explain the utility of the framework, we 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.[5] 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 carry only relevant information. We implemented some task agents, which collect information on the use of CPU and memory and the 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 can access SNMP data from a small stationary agent situated at its visiting cluster. The stationary agent allows that 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 having 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.

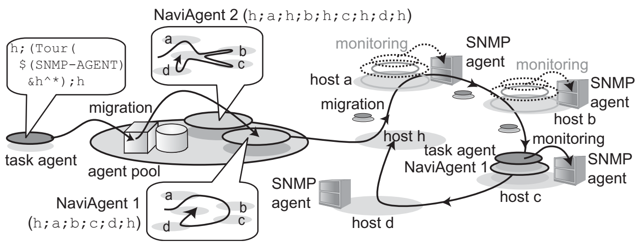

Figure 6. Mobile agent-based management system

The system deploys an agent pool at one host of each sub-network and offers several task agents and navigator agents as shown in Fig. 6. 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 can access SNMP data from a small stationary agent located at its visiting cluster. The stationary agent allows that 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 for monitoring 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 several navigator agents with different itineraries. However, due to a lack of space, this section illustrates only two navigator agents optimized for one of the sub-networks defined by `NaviAgent1` and `NaviAgent2` classes, respectively. `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 $h^\star$ 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 the above 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.

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

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 cluster, as defined in the `arrived()` method.

We have obtained a preliminary measurement of 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 is not 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 it is only 20 percent greater than the size of a self-contained task agent that controls its own itinerary. This is a small increase in size 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 straightly 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.

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 clusters by using navigator agents. By dynamically changing to a navigator agent suitable for the current sub-network, a task agent can efficiently migrate among clusters in various sub-networks to perform its task, without modifying its own program. The system also enables both navigator and task agents to be small and simple, because navigator agents do not have to offer any adaptive mechanisms for handling various networks and task agents contain no specific knowledge about sub-networks; they only have to know the location of the agent pools of their destinations. Moreover, the framework can strictly select one of the most suitable navigator agents, since it provides a theoretical and practical mechanism for comparing itineraries of the navigator agents. Our experience tells us that our navigator agents are useful in resource management of cluster computing environments, because they can provide a decentralized mechanism for deployment of computational tasks at remote clusters. As a result, the performance of our framework is scalable in the number of clusters. That is, we can naturally expect the system to still to be scalable even when applying it to a larger cluster computing environment.

7. Related Work

Mobile agent technology can provide a convenient, efficient, and robust management framework for cluster and grid computing. There have been several attempts to apply mobile agent technology to the management of cluster and grid computing [9, 10]. The focus of current research is, however, on the development mobile agent-based management systems themselves for particular cluster and grid computing environments. In fact, most existing systems have been constructed in ad-hoc manners or dependently on their target cluster computing systems or particular applications. Nevertheless, the tasks of building and operating mobile agents, which are specific for cluster and grid computing, have received little attention so far, although creating and operating such agents can be tedious and susceptible to errors.

Next, we compare our framework with some methodologies for building management mobile agents for distributed systems ADK [5] is notable because it can separate the travel itinerary of an agent from its behavior as our approach does. Aglets [6] 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 parts must be statically and manually embedded in the agent. Consequently, this agent, unlike ours, cannot dynamically change its itinerary and cannot travel beyond its familiar networks.

Also, there have been some theoretical models developed for specifying mobile agents, for example, Mobile UNITY [7] and Ambient calculus [3]. Mobile UNITY can specify control flows, variable, and conditional assignment statements at programs but cannot extract and reason about the itineraries of mobile components. The existing process algebra-based models, including Ambient calculus, are just theoretical frameworks for formalizing the whole computation of mobile agents and, as far as the author knows, they do not support any preorder relations for selecting mobile agents according to their itineraries.

Lastly, we should describe an approach to building configurable protocols for agent migration in another paper [13]. 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 among multiple clusters according to the topology of the current network and the requirement of the application-specific tasks. Our previous papers [12, 14] presented an approach for building a mobile agent from two layer components, like the framework presented in this paper. However, the previous approach aimed at only mobile agent-based network management systems, instead of any cluster computing. The previous papers did not present any matchmaking mechanisms for the two layer components. That is, they provide just a component-based approach for the development of mobile agent-based network management and did provide neither specification languages for agent itineraries nor algebraic relations.

8. Conclusion

This paper presented a methodology for building and operating reusable mobile agents for cluster and Grid computing. 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 clusters 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-cluster, 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 clusters as well as monitoring clusters and networks.

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.

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

[5] 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.

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

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

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

[9] O. F. Rana (eds), Proceedings of 2nd Workshop on Agent Based Cluster and Grid Computing, May 2002.

[10] O. F. Rana and S Graupner (eds), Proceedings of 3rd Workshop on Agent Based Cluster and Grid Computing, May 2003.

[11] 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.

[12] 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.

[13] I. Satoh, Configurable Network Processing for Mobile Agents on the Internet, Cluster Computing, Vol. 6, No.4 (Accepted), Kluwer, October 2003.

[14] 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.

[15] 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.