

# ADK — Building Mobile Agents for Network and Systems Management from Reusable Components

Thomas Gschwind\*  
tom@infosys.tuwien.ac.at

Metin Feridun†  
fer@zurich.ibm.com

Stefan Pleisch†  
spl@zurich.ibm.com

\*Distributed Systems Group  
Technische Universität Wien

†Zurich Research Laboratory  
IBM Research

## Abstract

*Mobile agents, programs that move within a system performing a set of tasks, are an active field of research. The focus of current research, however, is on the development of execution platforms and applications for mobile agents and not on methodologies for building agents. Creating mobile agents can be tedious and susceptible to errors. We propose a framework where the agent is composed using a well-defined set of categories of software components. Building systems from software components has already proven useful in the context of large software systems, increasing the productivity of the development process and the reliability of the resulting system by reusing proven components. We claim that the same holds true for the construction of mobile agents for network and systems management as well as for other domains. We have designed and implemented an agent construction toolkit (the AgentBean Development Kit—ADK) to demonstrate the usability and flexibility of this approach.*

## 1. Introduction

Globally distributed, network-based services such as e-commerce depend on the effective management of the infrastructure consisting of networks, servers and end-devices that support these services. The geographic reach of the components providing the service and the scale of the elements of the infrastructure ranging from servers to cellular phones call for models that differ from the widespread centralized management paradigm. One model is distributed management where management tasks are spread across the managed infrastructure and are carried out at or near managed resources. The goal is to minimize the network traffic that relates to management and to speed up management tasks by distributing operations across resources.

Our approach to distributed network and systems man-

agement is to use mobile code and mobile agents with a distributed infrastructure [4]. There is a class of tasks where mobile agents are advantageous over other approaches such as the client/server paradigm: management actions to be performed are relatively simple, the itinerary of an agent is well defined (explicitly or algorithmically) and the global operation and side-effects of the algorithm are well understood. One such example is a task for *monitoring* the system, where an agent observes a set of components and reacts locally to certain behavioral patterns. An agent may monitor the network traffic load on a router and switch to a higher bandwidth (and more expensive) connection when there is congestion. Another example is a task for *collection* of distributed data as in topology discovery, where the span of the operation and therefore the itinerary of the agent is determined based on the collected information. A discussion of the suitability of mobile agents in network management can be found in [2].

Current work on mobile agents focuses on the creation of an infrastructure, that among other tasks, provides functions and services that can be used by agents and a secure environment for both the mobile agents and their local execution environment [9, 10, 11, 13]. However, the task of constructing an agent as a software application has received little attention. The creation of a mobile agent, however simple, can be a tedious and error-prone task. To make the mobile agent design process easy and accessible, we propose a framework where the agent is composed of a set of categories of components: *navigational components* that are responsible for the agent's travel itinerary, *performers* that are responsible for executing one or more management tasks on each node, and *reporters* that determine how the results are collected and reported back.

In Section 2 we explain our categorization of software components for mobile agents, their interaction and a simple construction model for mobile agents. Section 3 outlines the overall architecture of our system. We present the infrastructure to execute agents constructed with the Agent-

Bean Development Kit and the architecture of the ADK itself. Section 4 details our implementation and Section 5 shows the usability of our toolkit based on a real world example: a mobile agent that discovers the network topology. Section 6 compares our approach with related work and Section 7 discusses issues we plan to attack in future versions of the ADK. Finally, we present our conclusions in Section 8.

## 2. Agent Model

Before we describe our agent model we provide an overview of our terminology. Based on this model, we will then explain how we envisage the construction process of a mobile agent for network and systems management.

### 2.1. Basic Agent Definitions

Throughout this paper we will adopt the definitions of the OMG's Mobile Agent System Interoperability Framework (MASIF) [12]. An *agent* is defined as a computer program that acts autonomously on behalf of a person or an organization.

An *agent system* is a platform that can create, interpret, execute, transfer and terminate a mobile agent. An agent system is associated with an authority for whom the agent system acts.

A *place* is a context within the agent system that provides a uniform environment in which an agent can execute. It is associated with a particular agent system. A place provides the means for managing mobile agents, enforcing security policies and accessing local resources.

*Mobile agents* are agents that move from place to place to perform given tasks. Usually, the itinerary of the mobile agent is determined by the mobile agent itself. Since this paper focuses on mobile agents we will refer to *mobile agents* simply as *agents*.

In our model we only consider *weak mobility* [5], which defines the ability to transfer code and initialization data, but not the execution state. The agent, however, may explicitly decide to store its execution state within its attributes.

### 2.2. Component-based Agent Model

The approach we have taken is to define a component model that helps to simplify the design of mobile agents. For many network and systems management tasks, we can identify categories of components and patterns for interconnecting them in order to perform the task. We propose three categories of interrelated components as follows:

**Navigator** The components in this category are responsible for determining and managing the itinerary of the

agent. The itinerary may be static, i.e., a list of nodes to visit determined at the time the agent was designed or instantiated, or dynamically based on the agent's previous computations and the current environment.

**Performer** The components of the performer category carry out the management tasks that should be executed at the host of the currently visited place. An agent may contain one or more components linked together to perform a task.

**Reporter** The components of this category manage the delivery of the agent's results to the designated destinations. A reporter component may for instance send a message to some display tool or aggregate the results and deliver them upon its return to the agent authority.

Note that these categories can also be applied to the design of mobile agents for handling tasks in other disciplines and are not restricted to network and systems management.

The concept of components allows a modular definition, facilitates creation of agents and encourages further reuse. Instead of creating an agent from scratch, the creator of an agent can glue the required components together. This is compliant with the fact that the creator frequently wants to focus on the business logic of the agent and not to bother with the technical details of the agent creation.

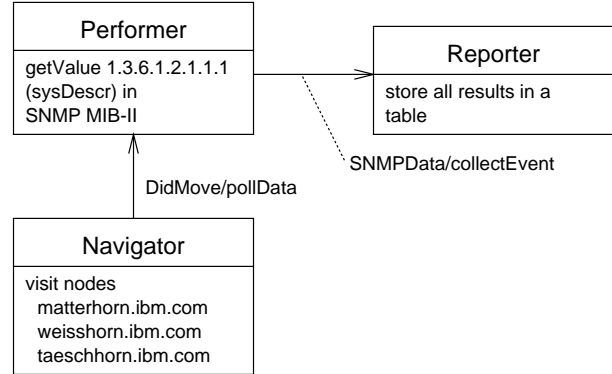


Figure 1. Example illustrating the three component categories

The interaction between components is based on an event/action-based communication. An event generated by one component may trigger an action by another component. Figure 1 shows an example of the model, a simple mobile agent for collecting configuration information. The navigator component provides a list of nodes to be visited sequentially. Upon arrival on a place the navigator generates a *DidMove* event which triggers the performer's polling action. The performer reads the value of

the system description from a local Management Information Base (MIB) [17] and generates an `SNMPData` event that is recorded by the reporter. After the events have been processed by all the components, the navigator continues with its itinerary.

### 2.3. Construction Process

Based on our component model we envisage a construction process as outlined in Figure 2. A visual design tool allows the selection of components from a list. The agent creator uses these components to compose the agent and specifies their interaction. Finally, the tool generates the source code for the agent.

Our visual design tool is based on Sun's `BeanBox` [18] model, but adds support for the construction of mobile agents. One major advantage to this approach is that users who have previously used one of the many `BeanBox` implementations will be immediately familiar with the ADK. Creating an agent thus becomes straightforward and happens on a business logic level. A system administrator who wants to create an agent does not have to be concerned about writing the code himself. Once he knows how to deal with the ADK he is able to build agents visually and only needs to focus on the management task of the agent.

Visual composition might be difficult if a large number of components needs to be modeled. However, we do not believe that this problem has a major impact on our approach. Management agents are preferably small in size for reasons of performance and maintenance, which makes agents with many components unlikely. Visual composition is usually targeted at component-based software systems with a small or medium number of components [19]. Tools such as wizards and catalogs can be used to simplify the location and selection of components during the design. As a fall back strategy, however, the components are general enough to support agent development using the components on a source code level.

## 3. Architecture

Our system consists of two modules. One module provides the agent construction toolkit (the `AgentBean Development Kit`) and the other module is a plug-in abstraction layer for existing mobile agent platforms to allow the execution of mobile agents created with the ADK.

### 3.1. AgentBean Development Kit

In Section 2 we have categorized the tasks that will be executed by most mobile agents dedicated to network and systems management. For our system we decided that each of these tasks should be implemented in a different software

component to allow for flexible reuse. For the modeling of the components we have decided to use Java Beans.

In [7], a Java Bean is defined as a reusable software component that can be visually manipulated in a builder tool (a so called *BeanBox*). A screenshot of the `BeanBox` contained in the ADK that allows the construction of mobile agents is shown in Figure 3. The most important features of Java Beans are the exposed set of properties, the set of methods they provide and the support for a set of events. Properties and events are either exposed implicitly by naming conventions and reflection or by a *BeanInfo* class that provides this kind of meta-information.

**Properties** are named attributes associated with a Java Bean that can be read and written using the appropriate methods. When using a Java Bean in a builder tool, properties can be edited and changed visually. In Figure 3, the properties of the *SNMPRequester* bean are shown on the right hand side.

**Methods** are like any other Java methods which can be called from other components.

**Events** provide a way for a component (the event source) to notify other components (the event listeners) that an event has happened. Whenever the event happens a method of the event listener will be called.

The advantage of the Java Beans architecture is that all the configuration classes such as property editors and `BeanInfo` classes will only be used at design time, but not after the code for the resulting agent has been generated. This makes the agent nearly as small as if it had been coded manually.

The Java Beans model perfectly matches our concept of a component-based agent and the interactions between the components. We therefore implement each component as a Java Bean, and base the communication on the Java Bean model. This approach has three key advantages:

- The widespread familiarity with the Java Beans concepts and tools means that it is likely that the `AgentBean Development Kit` will be accepted readily by users.
- A user who wants to create an agent need not be concerned about writing code himself. Once he knows how to deal with the ADK he is able to build agents visually and only needs to focus on the task of the agent.
- JavaBeans is the preferred component model for the Java programming language.<sup>1</sup> Our approach allows reuse of components even for mobile agents without having to rewrite those components. In addition, new

---

<sup>1</sup>A list of components available as Java Beans can be found at <http://java.sun.com/beans/marketing.html>

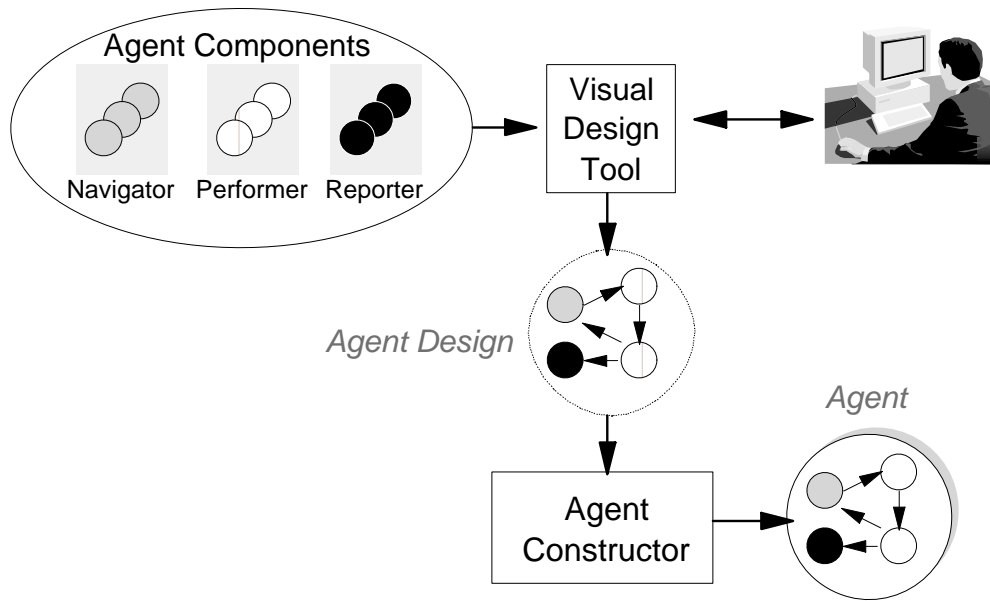


Figure 2. Based on the ADK the user combines components to build the agent

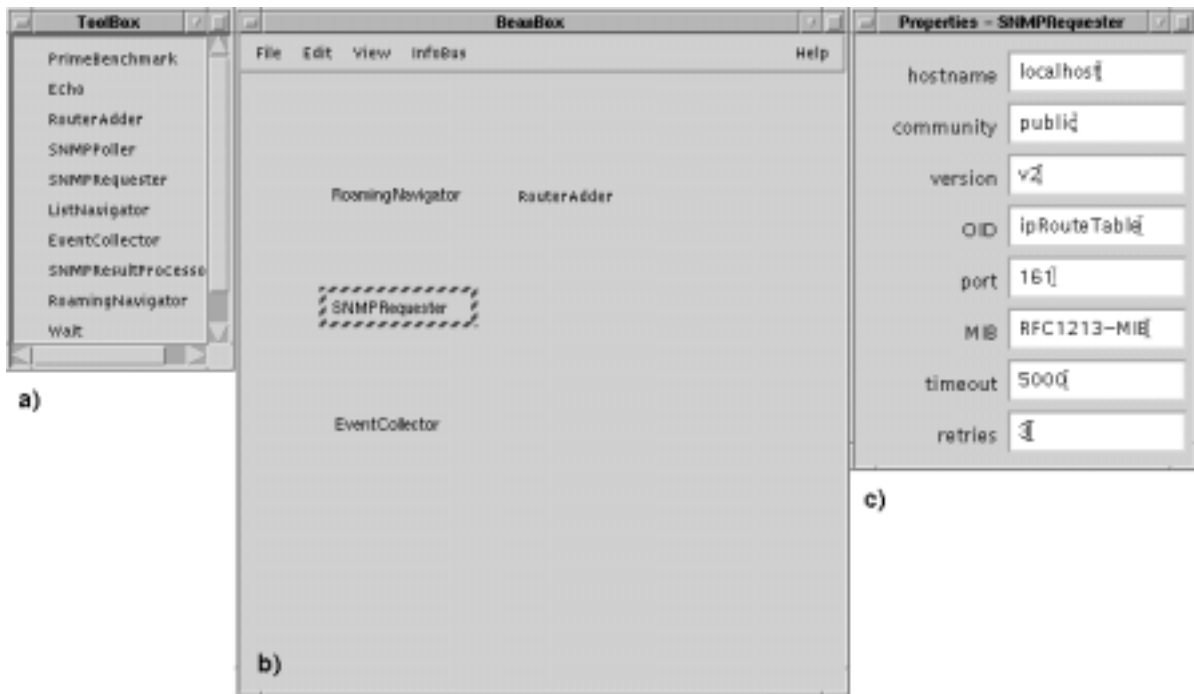
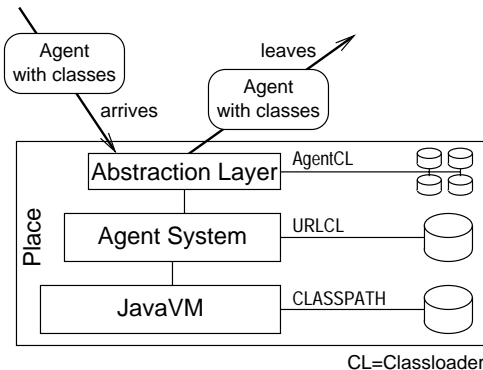


Figure 3. The AgentBean Development Kit with a) the available Java Beans, b) the composition area, and c) the SNMPRequester's property sheet

components can be added to ADK without modifications to the platform.

### 3.2. Abstraction Layer

In order to provide support for our agent model we have defined an architecture as depicted in Figure 4. A place that supports the execution of agents constructed with the ADK consists of three layers. Each layer provides a different kind of service and manages different types of classes. The lower two layers consist of the Java Virtual Machine and an existing agent system (i.e., Aglets [10], Distributed Management Framework - DMF [4], Voyager [13]). On top of these layers we have built an abstraction layer which isolates the diversity of the underlying agent system's transportation mechanisms.



**Figure 4. The layers of a place providing support for agents built with the ADK**

The Java Virtual Machine provides the same abstract execution environment on every operating system and is responsible for loading system classes such as the Java system classes and those used by the agent system. Usually, these classes are located via the CLASSPATH.

The agent system is responsible for protecting the host from malicious agents and for providing interfaces to local services. Classes loaded by the agent system are usually loaded using some sort of a URLClassLoader. The agent system is responsible for loading the abstraction layer and the well known components that are used to construct the agent.

The abstraction layer provides a consistent view of the underlying system to allow agents to migrate between places running different agent systems and provides the means to protect mobile agents from each other. Basically, the abstraction layer maps service requests into corresponding requests to the underlying agent system.

The abstraction layer is not responsible for providing a

common interface to all services provided by the agent system, but only to those that will be used in migration of the agent. For instance, the DMF provides a broad range of services for management tasks, such as a distributed directory. The interfaces to these services will likely be different from those on other agent system platforms. Beans relying on such services either need to implement their own abstraction mechanism or can only be used in combination with that specific agent system. Automating such adaptation is still an open research issue and will be attacked in future versions of the ADK.

Another purpose of the abstraction layer is the migration of agents through firewalls. To allow an agent to cross a firewall the abstraction layer needs to be installed on a router that is already running the DMF platform. Then the agent can cross this firewall because all the classes necessary to move the agent are passed on with the agent and are not being retrieved from a fixed codebase as it is the case in core Voyager [15] or in Gypsy mobile agent platform [11]. The ability to cross firewalls is of special importance for agents that perform network and systems management tasks.

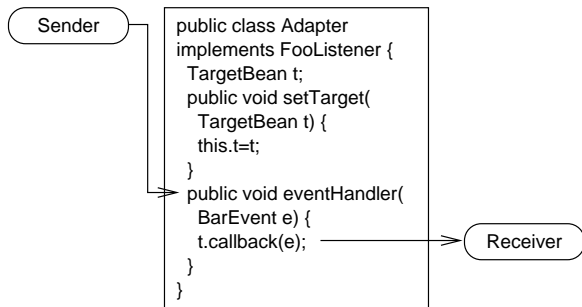
Finally, the abstraction layer protects an agent from other malicious agents. This feature has been included because we identified weaknesses in several agent systems allowing an agent to tamper with another agent by providing a class with the same name as used by the other agent but with a different implementation. For instance, ObjectSpace Voyager uses a static list of resource loaders [14]. Our approach to solve this problem is explained in Section 4.

Currently the abstraction layer is available for the DMF as well as for ObjectSpace's Voyager platform. In the near future, we also plan to port it to the Aglets Workbench [8] and to the Gypsy.

## 4. Implementation

The ADK is an extension of Sun's BeanBox implementation found in the Bean Development Kit (BDK) [18]. As the BeanBox provided by the BDK only allows the construction of applets, we adapted the BDK to generate mobile agents. Since we are reusing the principles provided by Java Beans, little work had to be done for that adaptation. We added a code generation module that works in a way similar to the applet generator, but sets up the components and their adaptors for an agent. Using other BeanBox implementations should be possible if they provide a means to specify custom initialization code.

When the user has finished the construction of the agent, the ADK generates source code to build the agent as specified by the user. The construction occurs in two phases. First, the beans are instantiated and their properties are configured according to the user's specification. Second, the communication channels between the beans are set-up.



**Figure 5. A sample adapter as provided by a typical BeanBox implementation**

Since a bean  $x$  receiving events of type  $foo$  will not always implement the corresponding  $foo$ -listener interface, special adapters have to be set up and configured. The adapter class implements the corresponding listener interface and stores a reference to the target bean. Whenever the event occurs, the event handler for the adapter class will be called, which in turn calls the method on the target bean as specified by the user in the BeanBox. A sample adapter is shown in Figure 5.

The code generated by the ADK consists of the agent's event adapter classes and an agent construction class that instantiates the beans used by the agent, sets the properties, and configures the event adapters. The agent is then started by calling the `start` method of the agent's navigator component. When the agent is moved to its first host, it is serialized starting from the agent's navigator component class and thus discards all the classes used for the agent's construction.

The abstraction layer is responsible for transmitting the agent between different mobile agent systems (e.g., between an Aglet and a Voyager server). For implementing that functionality it provides its own `move` method. The arguments of the `move` method are the agent's destination and the callback method to be called to revive the agent at the target system. The `move` method then checks which classes need to be transferred and stores them along with the callback method and the serialized version of the agent in the `AgentDescriptor` class.

The `AgentDescriptor` class is transferred to the target place using the target system's transfer semantics. In case of Voyager, we implemented our own remote interface. When the target place receives the agent, it is passed on to its abstraction layer. The abstraction layer creates its own class loader for loading the classes specified in the `AgentDescriptor` instance, deserializes the agent using this class loader and calls the agent's callback method for reviving it.

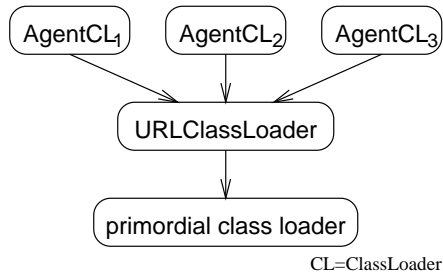
Depending on the configuration of the agent of the agent either all the classes are piggy-backed onto the agent or just the adaptor classes providing the glue between the components. While the first approach provides maximum flexibility since it allows the agent to be executed on every node, the second approach reduces the size of the agent to its minimum by requiring the agent components to be installed locally on the places where the agent will be executed.

Since the agent's class files are included in the `AgentDescriptor`, they are migrated from the place that has been last visited and not from a central place. This allows the agent to cross firewalls which is of special importance for agents that have to pursue network management tasks (e.g., an agent for discovering the network topology as shown in Section 5).

We did not try to protect the mobile agent platform from a malicious mobile agent since this functionality is already provided by the underlying mobile agent system. However, since we have found weaknesses in several mobile agent systems that do not prevent a malicious agent from tampering with other agents, we have implemented our own mechanisms to protect agents from one another. It seems that many mobile agent system implementations rely on general adherence to the Java naming conventions and assume that agent implementations use unique, globally reserved package names (the author's domain name in reverse order). However, this convention is not enforced and thus cannot be relied upon.

Initially we wanted to utilize Voyager's class loading architecture [14]. Voyager's `VoyagerClassLoader` allows a set of resource loaders to be registered. Whenever a class needs to be loaded, Voyager queries these resource loaders one after the other until the class has been found. However, the list of class loaders is static and will be used for all classes loaded by Voyager—be it agent classes as well as Voyager system classes. This allows an agent to tamper with another agent by just providing a class with the same name as that used by another agent and letting Voyager load the malicious replacement. In some cases this approach even allows the replacement of some of Voyager's system classes with a fake class set. For this reason, we implemented our own approach for class loading in the abstraction layer.

We decided to cope with the standard class loading architecture as presented in [6] and make use of the Java language property that a given class can be loaded several times by different class loaders. The classes will be treated as different types and thus cannot be mixed. Whenever a class is instantiated, Java tries to load the new class using the class loader of the instantiating class. This class loader attempts to load the class using its own mechanisms. If that fails, it tries the upstream class loader (the class loader that loaded the class loader's class). This approach will always form



**Figure 6. A simple powerful class loading architecture**

a delegation tree that defines how a class that needs to be loaded is located.

Figure 6 shows the delegation tree relating to Figure 4. The leaves are formed by the `AgentClassLoaders`. There is one `AgentClassLoader` for each agent that is responsible for loading the classes provided by its agent (i.e., the adapter classes). If this class loader cannot locate the requested class (i.e., a Java Bean that is not provided by the agent) it will delegate the request to the `URLClassLoader` of the DMF. If this also fails, the request will be delegated to the primordial class loader (Java’s system class loader).

In comparison to relying on a single class loader for loading all classes, the idea of using a class loader for each agent has another advantage. All the agent’s data are garbage collected when the agent moves on to the next host. Even the agent’s class loader and its adapter class files are garbage collected. With an approach that uses a single class loader this is not possible since a class must not be loaded twice and thus has to be retained until the class loader is garbage collected itself [6].

## 5. Example

In the previous sections we presented the architecture of the ADK and its implementation. As a proof of concept we will discuss an example, the *Network Topology Discovery Agent*, which discovers the topology of a given network. The example illustrates the steps an agent passes through from its creation and configuration to its termination.

Figure 7 shows the four Java Beans we use for this agent and the communication patterns among them. The components have the following tasks:

### Navigator

- The `RoamingNavigator` bean maintains a list of hosts to be visited and provides methods to add and remove elements from this list.

### Performer

- The `RouterAdder` bean receives SNMP events and extracts a list of routers from the SNMP event. These routers are then fed into the roaming navigator as nodes to be visited if they have not already been visited.
- The `SNMPRequester` bean requests SNMP data from a nearby located SNMP MIB and triggers an event containing the data requested from the MIB.

### Reporter

- The `EventCollector` bean collects all events received and provides a method to print all the events that have been collected so far.

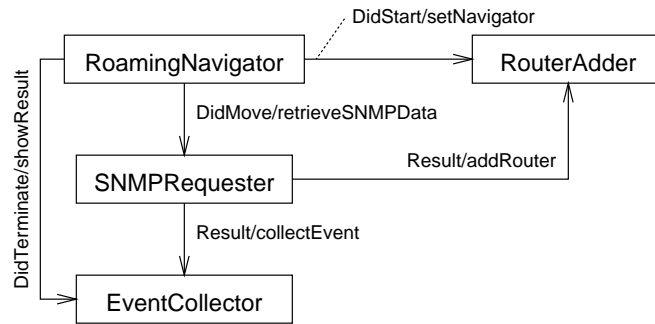
Without loss of generality we consider for the purpose of this discussion only the creation of the agent, the execution of the agent at its current place, its transition to the next place (next router on the list) and the activities upon termination of the agent.

The agent authority builds the agent by adding the corresponding components to the ADK, configuring the bean’s properties and linking them with the appropriate events to specify the interactions among the JavaBeans. For instance the `SNMPRequester` bean is configured with the appropriate MIB object identifier (OID) for polling the routing table of the SNMP agent. Then, the ADK generates the agent that the agent authority can then inject into the network.

When the agent starts, it generates the `didStart` event which triggers the `RouterAdder`’s `setNavigator` method. This allows the `RouterAdder` to obtain a reference to the agent’s navigator as soon as the agent is started.

Whenever the agent moves to a new place a `didMove` event will be generated and subsequently the `SNMPRequester`’s `retrieveSnmpData` method will be called. This informs the `SNMPRequester` to query the local host for its routing table using the preconfigured SNMP query. The `SNMPRequester` itself generates a `result` event when the routing is queried containing the routing table retrieved from the local host. Both the `EventCollector` and the `RouterAdder` are subscribers to this event and start executing the methods `collectEvent` and `addRouters` respectively. The `collectEvent` method simply stores the event and `addRouters` filters out all the routers from the routing table. Since the example implementation runs on the DMF platform, the `RouterAdder` checks which of these routers are running the DMF and feeds the list of routers supporting the DMF into the roaming navigator as nodes that need to be visited (if they have not yet been visited). Now the agent can move on to the next router.

The agent completes its itinerary when the `RoamingNavigator` has no more routers that need to be visited. This is indicated by the `didTerminate` event which



**Figure 7. A sample agent that discovers the network topology**

forces the `EventCollector` to make its results available by dumping them to some predefined location. Finally, the agent terminates its execution.

## 6. Related Work

We have identified two other projects that focus on the construction of mobile agents. The first project is the Jumping Beans project [1, 16]. Compared to our system it uses a completely different approach. While we help the user in the construction of the mobile agent, the Jumping Beans approach is an API which provides a framework for mobilizing applications whose key features is the integration into existing environments. One disadvantage is that applications mobilized with Jumping Beans cannot cross firewalls. We think that the reason for this is that applications using Jumping Beans have to return back to the Jumping Beans Server after each hop.

Another approach is taken by the Iconic Modeling Tool (IMT) [3]. The IMT is a tool that allows the user to specify a step by step workflow that should be followed. [3] suggests that the tasks that can be used to specify the agent's workflow are fixed and that the workflow cannot be computed dynamically while the agent is executing. The tasks that can be specified allows the agent to move between a set of hosts and to acquire various data, which can be reported back in several ways. It seems that the IMT needs to be patched when a new task has to be added. In comparison, adding a new task to agents constructed with the ADK requires only the implementation of a Java Bean.

## 7. Future Work

The current BeanBox implementation we are using has two limitations (they more generally apply to the BeanBox contained in Sun's BDK [18]). The structure of how events are reported has to be specified beforehand. Thus it is not possible to change the event structure dynamically and thus

adapt to changing requirements. One solution to this problem is to extend the event model with conditional events. The condition that has to be met could be included in the event's adapter classes.

The other drawback is that an event can only be processed by a method having one parameter of the type of the event (or the type of one of its parent classes) or no argument at all. This problem might also be solved by extending the adapters. The adapter could translate the event and thus would allow to call methods with other arguments as well.

Currently the abstraction layer is only available for the DMF [4] and ObjectSpace's Voyager [13] platform. In the future we plan to support Aglets [8] and Gypsy [11] too. Additionally, we plan to convert the abstraction layer itself into a mobile agent itself that has the ability to migrate between different agent platforms. This will give us the possibility to move agents constructed with the ADK to places that do not provide the abstraction layer initially.

Providing the infrastructure to allow the migration of an agent between places using different agent systems is only half a solution. Usually an agent will have to use the local infrastructure which differs on every agent system semantically. For instance the interface to initiate a traceroute request will look differently on different systems. One solution is to hardcode every possible interface into the agent. However, this obviously will not scale well. Thus, we will be focusing on the dynamic adaptation of interfaces. Possible solutions are the use of meta-languages or the consultation of an adapter library that could provide a matching adapter.

Another solution that we will be considering in the future versions of the ADK is a distributed debug and trace environment that allows users to discover bugs within the design of an agent or gives the agent authority feedback on the agent's current state.

## 8. Conclusion

We have presented a new approach for building mobile agents. The idea is to build agents from customizable and reusable components. With current mobile agent systems that support the agent's creator only at a basic level (i.e., by only providing a means to move an agent), we will be facing a huge set of monolithic and inflexible agents in the future.

Our approach reduces the construction of an agent to the configuration of well-defined components and the relationships between them, therefore reducing the time needed to develop a mobile agent. Additionally, agents constructed using this approach will be more reliable since they are reusing components that have already proven their reliability in other agents.

We have identified three categories of software components for network and systems management mobile agents: *Navigators*, *Performers*, and *Reporters*. We expect, however, that the same categorization can be applied to other application domains. As a means to model these components we have chosen to use Java Beans. Describing components written in Java as Java Beans is a well-established standard and allows to reuse Java Beans for mobile agents that initially were written for different purpose. We also implemented the AgentBean Development Kit (an extension of Sun's Bean Development Kit) to support the visual construction of mobile agents.

To prove our concepts we have implemented a set of mobile agents for pursuing network management tasks. We have presented one of those agents in Section 5, illustrating the usability of our approach. Based on our experiences we expect that similar support for agent construction will be included in other agent systems.

## Availability

The AgentBean Development Kit is available from <http://www.infosys.tuwien.ac.at/ADK/>.

## References

- [1] Ad Astra Engineering. Jumping Beans White Paper. <http://www.JumpingBeans.com/>, Dec. 1998. Sunnyvale, California.
- [2] A. Bieszczad, B. Pagurek, and T. White. Mobile Agents for Network Management. *IEEE Communications Surveys*, 1(1), Fourth Quarter 1998. <http://www.comsoc.org/pubs/surveys/>.
- [3] B. Falchuk and A. Karmouch. Visual Modeling for Agent-Based Applications. *COMPUTER*, 31(12):31–38, Dec. 1998.
- [4] M. Feridun, W. Kasteleijn, and J. Krause. Distributed Management with Mobile Components. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, pages 857–870, May 1999. Boston, Massachusetts.
- [5] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5), 1998.
- [6] L. Gong. Secure Java Class Loading. *IEEE Internet Computing*, 2(6):56–61, November/December 1998.
- [7] G. Hamilton, editor. *JavaBeans*. Sun Microsystems, <http://java.sun.com/beans/>, July 1997.
- [8] IBM. Aglets Software Development Kit. <http://www.trl.ibm.co.jp/aglets/>, Mar. 1999.
- [9] J. Kiniry and D. Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, pages 21–30, July/August 1997.
- [10] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Computer & Engineering Publishing Group, 1998.
- [11] W. Lugmayr. The Gypsy Project on Mobile Agents. <http://www.infosys.tuwien.ac.at/Gypsy/>, 1998.
- [12] D. Milojevic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF, The OMG Mobile Agent System Interoperability Facility. In K. Rothermel and F. Hohl, editors, *Proceedings of the Mobile Agents'98*. Springer, Sept. 1998.
- [13] ObjectSpace, <http://www.objectspace.com/products-voyager1.htm>. *ObjectSpace Voyager 2.0*, 1998.
- [14] ObjectSpace. Voyager Core Technology 2.0 API Documentation. Included in the Voyager distribution., 1998.
- [15] ObjectSpace. *Voyager Core Technology 2.0 User Guide*, 1998.
- [16] R. Orfali and D. Harkley. *Client/Server Programming with Java and Corba*. Wiley Computer Publishing, 1998.
- [17] W. Stallings. *SNMP, SNMPv2, and CMIP*. Addison-Wesley, 1993.
- [18] Sun Microsystems. The Bean Development Kit. [http://java.sun.com/beans/software/bdk\\_download.html](http://java.sun.com/beans/software/bdk_download.html), July 1998.
- [19] C. Szyperski. *Component Software: Beyond Object Oriented Programming*. Addison-Wesley, 1998.