Mobile Agents with Java: The Aglet API[1]

Danny B. Lange
General Magic Inc.
420 North Mary Avenue
Sunnyvale, CA 94086 U.S.A.
danny@acm.org

Mitsuru Oshima
IBM Tokyo Research Laboratory
1623-14 Shimotsuruma, Yamato-shi
Kanagawa-ken 242, Japan
moshima@trl.ibm.co.jp

Abstract

Java, the language that changed the Web overnight, offers some unique capabilities that are fueling the development of mobile agent systems. In this article we will show what exactly it is that makes Java such a powerful tool for mobile agent development. We will also draw attention to some shortcomings in Java language systems that have implications for the conceptual design and use of Java-based mobile agent systems. Last, but not least, we will introduce the aglet – a Java-based agile agent. We will give an overview of the aglet and, its application programming interface, and present a real-world example of its use in electronic commerce.

1 Introduction

Mobile agents are the basis of an emerging technology that promises to make it very much easier to design, implement, and maintain distributed systems [White 1996]. We have found that mobile agents reduce network traffic, provide an effective means of overcoming network latency, and perhaps most importantly, through their ability to operate asynchronously and autonomously of the process that created them, help us to construct more robust and fault-tolerant systems.

This is not the place to examine the characteristics of the numerous agent systems made available to the public by many research and development labs. But if we looked at all these systems, we would find that a property shared by all agents is that fact that they live in some environment. They have the ability to interact with their execution environment, and to act asynchronously and autonomously upon it. No one is required either to deliver information to the

[1] This article is based on a chapter of a forthcoming book by Lange and Oshima entitled *Programming and Deploying Mobile Agents with Java* (Addison-Wesley) [Lange and Oshima 1998].

To appear in *World Wide Web* (Baltzer Science Publishers, The Netherlands).

agent or to consume any of its output. The agent simply acts continuously in pursuit of its own goals.

We define agents in the following way: An agent is a software object that

- is situated within an execution environment;
- possesses the following mandatory properties:

  Reactive — senses changes in the environment and acts in accordance with those changes;
  Autonomous — has control over its own actions;
  Goal-driven — is pro-active;
  Temporally continuous — executes continuously;

- and may possess one or more of the following orthogonal properties:

  Communicative — can communicate with other agents;
  Mobile — can travel from one host to another;
  Learning — adapts in accordance with previous experience;
  Believable — appears believable to the end-user.

Mobility is an orthogonal property of agents. That is, all agents do not necessarily *have* to be mobile. An agent can just sit there and communicate with its surroundings by conventional means. These means include various forms of remote procedure calling and messaging. We call agents that do not or cannot move *stationary agents*. In contrast, a *mobile* agent is not bound to the system where it begins execution. The mobile agent is free to travel among the hosts in the network. Created in one execution environment, it can transport its state and code with it to another execution environment in the network, where it resumes execution.

By the term "state," we typically understand the agent attribute values that help it determine what to do when it resumes execution at its destination. By the term "code," we understand, in an object-oriented context, the class code necessary for the agent to execute. We define agents in the following way: A mobile agent is not bound to the system where it begins execution. It has the unique ability to transport itself from one system in a network to another. The ability to travel allows a mobile agent to move to a system that contains an object with which the agent wants to interact, and then to take advantage of being in the same host or network as that object.

So does mobile agent technology sound appealing? Our interest in mobile agents should not be motivated by the technology *per se*, but rather by the benefits they provide for the creation of distributed systems. So here are *seven good reasons* to start using mobile agents:

1. **They reduce the network load.** Distributed systems often rely on communication protocols that involve multiple interactions to accomplish a given task. This is especially true when security measures are enabled. The result is a lot of network traffic. Mobile agents allow us to package a conversation and dispatch it to a destination host where the interactions can take place locally. Mobile agents are also useful when it comes to reducing the flow of raw data in the network. When very large volumes of data are stored at remote hosts, these data should be processed in the locality of the data, rather that transferred over the network. The motto is simple: move the computations to the data rather than the data to the computations.

2. **They overcoming network latency.** Critical real-time systems such as robots in manufacturing processes need to respond to changes in their environments in real time. Controlling such systems through a factory network of a substantial size involves significant latencies. For critical real-time systems, such latencies are not acceptable. Mobile agents offer a solution, since they can be dispatched from a central controller to act locally and directly execute the controller's directions.

3. **They encapsulate protocols.** When data are exchanged in a distributed system, each host owns the code that implements the protocols needed to properly code outgoing data and interpret incoming data, respectively. However, as protocols evolve to accommodate new efficiency or security requirements, it is a cumbersome if not impossible task to upgrade protocol code properly. The result is often that protocols become a legacy problem. Mobile agents, on the other hand, are able to move to remote hosts in order to establish "channels" based on proprietary protocols.

4. **They execute asynchronously and autonomously.** Often mobile devices have to rely on expensive or fragile network connections. That is, tasks that require a continuously open connection between a mobile device and a fixed network will most likely not be economically or technically feasible. Tasks can be embedded into mobile agents, which can then be dispatched into the network. After being dispatched, the mobile agents become independent of the creating process and can operate asynchronously and autonomously. The mobile device can reconnect at some later time to collect the agent.

5. **They adapt dynamically.** Mobile agents have the ability to sense their execution environment and react autonomously to changes. Multiple mobile agents possess the unique ability to distribute themselves among the hosts in the network in such a way as to maintain the optimal configuration for solving a particular problem.

6. **They are naturally heterogeneous.** Network computing is fundamentally heterogeneous, often from both hardware and software perspectives. As mobile agents are generally computer- and transport-layer-independent, and dependent only on their execution environment, they provide optimal conditions for seamless system integration.

7. **They are robust and fault-tolerant.** The ability of mobile agents to react dynamically to unfavorable situations and events makes it easier to build robust and fault-tolerant distributed systems. If a host is being shut down, all agents executing on that machine will be warned and given time to dispatch and continue their operation on another host in the network.

It would be an understatement to say that the Java programming language [Arnold and Gosling 1998] has revolutionized the mobile agent field. The remainder of this article is devoted to the application of Java for mobile agents. Most of our presentation is based on the experience gained from creating IBM's Aglets Workbench (`http://www.trl.ibm.co.jp/aglets`)- a system that provides an *applet-like* programming model for mobile agents [Lange and Oshima 1998].

The article is structured as follows: in Section 2 we present the agent characteristics of the Java programming language as well some of its shortcomings; Section 3 and Section 4 introduce the Aglet programming model and the Aglet API; Section 5 reports on an application of aglets in electronic commerce; Section 6 gives a brief overview of contemporary Java-based agent systems; and Section 7 concludes the paper.

2 Agent Characteristics of Java

Java is an object-oriented network-savvy programming language. Some call it a *Better C++* that omits many rarely used, confusing features of C++. Others call it *the* language of the Internet. We think of as *jet fuel* for mobile agents. Let us view some of the properties of Java that make it a good language for mobile agent programming.

Platform-independence. Java is designed to operate in heterogeneous networks. To enable a Java application to execute anywhere on the network, the compiler generates architecture-neutral byte code, as opposed to non-portable native code. For this code to be executed on a given computer, the Java runtime system needs to be present. There are no platform-dependent aspects of the Java language. Primitive data types are rigorously specified and not dependent on the underlying processor or operating system. Even libraries are platform-independent parts of the system. For example, the window library provides a single interface for the GUI that is independent of the underlying operating system. It allows us to create a mobile agent without knowing the types of computers it is going to run on.

Secure execution. Java is intended for use on the Internet and intranets. The demand for security has influenced the design in several ways. For example, Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data. Java simply does not allow illegal type casting or any pointer arithmetic. Programs are no longer able to forge access to private data in objects that they do not have access to. This prevents most activities of viruses. Even if someone tampers with the byte code, the Java runtime system ensures that the code will not be able to violate the basic semantics of Java. The security architecture of Java makes it reasonably safe to host an untrusted agent, because it cannot tamper with the host or access private information.

Dynamic class loading. This mechanism allows the virtual machine to load and define classes at runtime. It provides a protective name space for each agent, thus allowing agents to execute independently and safely from each other. The class-loading mechanism in extensible and enables classes to be loaded via the network.

Multithread programming. Agents are by definition autonomous. That is, an agent executes independently of other agents residing within the same place. Allowing each agent to execute in its own lightweight process, also called a thread of execution, is a way of enabling agents to behave autonomously. Fortunately, Java not only allows multithread programming, but also supports a set of synchronization primitives that are built into the language. These primitives enable agent interaction.

Object serialization. A key feature of mobile agents is that they can be serialized and deserialized. Java conveniently provides a built-in serialization mechanism that can represent the state of an object in a serialized form sufficiently detailed for the object to be reconstructed later. The serialized form of the object must be able to identify the Java class from which the object's state was saved, and to restore the state in a new instance. Objects often refer to other objects. Those other objects must be stored and retrieved at the same time, to maintain the object structure. When an object is stored, all the objects in the graph that are reachable from that object are stored as well.

Reflection. Java code can discover information about the fields, methods, and constructors of loaded classes, and can use reflected fields, methods, and constructors to operate on their underlying counterparts in objects, all within the security restrictions. Reflection accommodates the need for agents to be smart about themselves and other agents.

Although the Java language system is highly suitable for creating mobile agents, we should be aware of some significant shortcomings. Some of these shortcomings can be worked around; others are more serious, and have implications for the overall conceptual design and deployment of Java-based mobile agent systems.

Inadequate support for resource control. The Java language system provides no means for us to control the resources consumed by a Java object. For example, an agent can start looping and waste processor cycles. The agent can also start consuming memory resources. These two examples relate to a specific type of security attack termed *denial of service.* This is the most feared type of attack by mobile agents: Agents swarm into a computer and take over all its resources, making it impossible for the owner to control his or her own computer. Unfortunately, Java provides no ways for the host to limit the processor and memory resources allocated by a given object or thread. A related issue is the ability of the agent to allocate resources external to the program, for example by opening files and sockets, and creating windows. The agent's allocation of these resources can be controlled but once the agent is disposed of or dispatched to another host, these resources must be released. However, it is difficult to do so, because there is no way of binding such resources to a specific object. So what we may see is a mobile agent that “forgets” its GUI and leaves behind an open window on our display when it leaves for another host.

No protection of references. A Java object’s public methods are available to any other object that has a reference to it. Since there is no concept of a protected reference, some objects are allowed access to a larger set of public methods in the Java object’s interface than others. This is important for the agent. There is no way that it directly can monitor and control which other agents are accessing its methods. We have found that a practical and powerful solution to this problem is to insert a *proxy* object between the caller and the callee to control access. This not only provides protection of references, it also offers a solution to the problem mentioned in the next item and provides location transparency in general (explained later).

No object ownership of references. No one owns the references to a given object in Java. For an agent, this means that we can take its thread of execution away from it, but we cannot explicitly void the agent (object) itself. This is a task for the automated garbage collector. At present, the garbage collector will not reclaim any object until all references to the object have been voided. So if some other agent has a reference to our agent, it will unavoidably open a loophole for that agent to keep *our* agent alive against our will. All we can do is to repeat the warning against giving away direct references to agents in Java. The upcoming Java Development Kit (JDK) 1.2 provides support for weak reference, which will solve this problem.

No support for preservation and resumption of the execution state. It is currently impossible in Java to retrieve the full execution state of an object. Information such as the status of the program counter and frame stack is permanently forbidden territory for Java programs. Therefore, for a mobile agent to properly resume a computation on a remote host, it must rely on internal attribute values and external events to direct it. An embedded automaton can keep track of the agent's travels and ensure that computations are properly halted and properly resumed.

3 The Aglet Model

IBM's Aglets Workbench (Aglets), created by the authors of this article, mirrors the applet model in Java. The goal was to bring the flavor of mobility to the applet. The term *aglet* is indeed a portmanteau word combining *agent* and *applet*. We attempted to make Aglets an exercise in "clean design," and it is our hope that applet programmers will appreciate the many ways in which the aglet model reflects the applet model.

3.1 Basic Elements

Now let us take a closer look at the model underlying the Aglet API. This model was designed to benefit from the agent characteristics of Java while overcoming some of the above-mentioned deficiencies in the language system.

Most notably, the model defines a set of abstractions and the behavior needed to leverage mobile agent technology in Internet-like open wide-area networks. The key abstractions are aglet, proxy, context, message, future reply, and identifier:

Aglet. An aglet is a mobile Java object that visits aglet-enabled hosts in a computer network. It is autonomous, since it runs in its own thread of execution after arriving at a host, and reactive, because of its ability to respond to incoming messages.

Proxy. A proxy is a representative of an aglet. It serves as a shield for the aglet that protects the aglet from direct access to its public methods. The proxy also provides location transparency for the aglet; that is, it can hide the aglet’s real location of the aglet.

Context. A context is an aglet's workplace. It is a stationary object that provides a means for maintaining and managing running aglets in a uniform execution environment where the host system is secured against malicious aglets. One node in a computer network may run multiple servers and each server may host multiple contexts. Contexts are named and can thus be located by the combination of their server's address and their name.

Message. A message is an object exchanged between aglets. It allows for synchronous as well as asynchronous message passing between aglets. Message passing can be used by aglets to collaborate and exchange information in a loosely coupled fashion.

Future reply. A future reply is used in asynchronous message-sending as a handler to receive a result later asynchronously.

Identifier. An identifier is bound to each aglet. This identifier is globally unique and immutable throughout the lifetime of the aglet.

Behavior supported by the aglet object model is based on a careful analysis of the "life and death" of mobile agents. There are basically only two ways to bring an aglet to life: either it is instantiated from scratch (creation) or it is copied from an existing aglet (cloning). To control the population of aglets one can of course destroy aglets (disposal). Aglets are mobile in two different ways: active and passive. The first is characterized by an aglet pushing itself from its current host to a remote host (dispatching). A remote host pulling an aglet away from its current host (retracting) characterizes the passive type of aglet mobility. When aglets are well and running they take up resources. To reduce their resource consumption, aglets can go to sleep temporarily, releasing their resources (deactivation), and later be brought back into running mode (activation). Finally, multiple aglets may exchange information to accomplish a given task (messaging).

This is just about the minimal set of operations required to create and manage a distributed mobile agent environment. When we created the Aglet API, one of our goals was to create a lightweight API that would be both easy to learn to use and sufficiently complete and robust for real applications. It is tempting to call the Aglet API the "RISC" of mobile agents.

Below we summarize the aglets' fundamental operations, namely, creation, cloning, dispatching, retraction, deactivation, activation, disposal, and messaging (see also Figure 1):

Creation. The creation of an aglet takes place in a context. The new aglet is assigned an identifier, inserted into the context, and initialized. The aglet starts executing as soon as it has been successfully initialized.

Cloning. The cloning of an aglet produces an almost identical copy of the original aglet in the same context. The only differences are the assigned identifier and the fact that execution restarts in the new aglet. Note that execution threads are not cloned.

Dispatching. Dispatching an aglet from one context to another will remove it from its current context and insert it into the destination context, where it will restart execution (execution threads do not migrate). We say that the aglet has been “pushed” to its new context.

Retraction. The retraction of an aglet will pull (remove) it from its current context and insert it into the context from which the retraction was requested.

Activation and deactivation. The deactivation of an aglet is the ability to temporarily halt its execution and store its state in secondary storage. Activation of an aglet will restore it in a context.

Disposal. The disposal of an aglet will halt its current execution and remove it from its current context.

Messaging. Messaging between aglets involves sending, receiving, and handling messages synchronously as well as asynchronously.

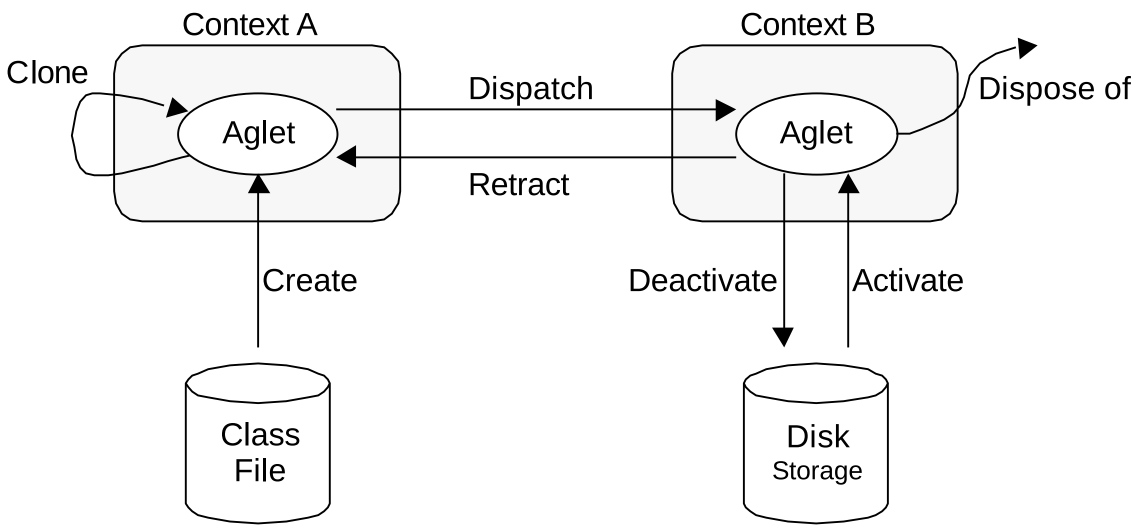


Figure 1 Aglet Life-Cycle Model

3.2 The Aglet Event Model

The Aglet programming model is event-based. The model allows the programmer to "plug in" customized *listeners* into an aglet. Listeners will catch particular events in the lifecycle of an aglet and allow the programmer to take action when the aglet is being dispatched, for instance.

Three kinds of listener exist:

> **Clone Listener.** Listens for cloning events. One can customize this listener to take specific actions when an aglet is about to be cloned, when the clone is actually created, and after the cloning has taken place.
>
> **Mobility Listener.** Listens for mobility events. One can use this listener to take action when an aglet is about to be dispatched to another context, when it is about to be retracted from a context, and when it actually arrives in a new context.
>
> **Persistence Listener.** Listens for persistent events. This listener allows the programmer to take action when an aglet is about to be deactivated and after it has been activated.

The `CloneLimiter` is an example of a reusable listener. It can be plugged into any aglet for which we want to limit the number of clones that can be made from it. The `CloneLimiter` listener implements three methods, `onCloning`, `onClone`, and `onCloned`, which are called in sequence when an application tries to clone the aglet that hosts the listener. Most notably, this implementation of `CloneLimiter` will allow at most five clones of the original aglet, none of which can themselves be cloned.

```
public class CloneLimiter implements CloneListener() {

   final integer MAX = 5;
   boolean original = true;
   int number_of_clones = 0;

   // Called when the aglet is about to be cloned.
   public void onCloning(CloneEvent ev) {
```

```
            if (original == false)
               throw new SecurityException("Clone cannot create a clone");
            if (number_of_clones > MAX)
               throw new SecurityException("Exceeds the limit");
         }

         // Called in the cloned aglet.
         public void onClone(CloneEvent ev) {
            original = false;
         }

         // Called in the original aglet after the cloning.
         public void onCloned(CloneEvent ev) {
            number_of_clone++;
         }
      }
```

3.3 Agent Design Patterns

A very important part of the Aglet programming model is the idea of *agent design patterns*. [Aridor and Lange 1998] During the early work on the Aglet API, we recognized a number of recurrent patterns in the design of aglet applications. Several of these patterns were given intuitive meaningful names such as *Master-Slave*, *Messenger*, and *Notifier*. They were implemented in Java and included in the first release of the Aglets Workbench. These early patterns were found to be highly successfully for jump-starting users who were new to Aglets and the mobile agent paradigm.

This experience tells us that it is important to identify the elements of good and reusable designs for mobile agent applications and to start formalizing people's experience with these designs. This is the role of agent design patterns. The concept originated with software engineers and researchers in the object-oriented community, and has been recognized as one of the most significant innovations in the object-oriented field.

The patterns we have discovered so far can be conceptually divided into three classes: *traveling*, *task*, and *interaction*. This classification scheme makes it easier to understand the domain and application of each pattern, to distinguish different patterns, and to discover new patterns.

4 A Tour of the Aglet API

Now, get ready for a tour of the core classes and interfaces of the aglet API. This is the API that the programmer will use to create and operate aglets. It contains methods for initializing an aglet, message handling, and dispatching, retracting, deactivating/activating, cloning, and disposing of the aglet. The aglet API is simple and yet flexible. Created in the spirit of Java and representing a lightweight pragmatic approach to mobile agents, the aglet API is a Java package (`aglet`) consisting of classes and interfaces, most notably: `Aglet`, `AgletProxy`, `AgletContext`, and `Message`.

4.1 `Aglet` Class

The Aglet class is the key class in the Aglet API. This is the abstract class that the aglet developer uses as a base class when he or she creates customized aglets. The `Aglet` class defines methods for controlling its own life cycle, namely, methods for cloning, dispatching, deactivating, and disposing of itself. It also defines methods that are supposed to be overridden in its subclasses by the aglet programmer, and provides the necessary "hooks" to customize the behavior of the aglet. These methods are systematically invoked by the system when certain events take place in the life cycle of an aglet.

Let us describe some of the methods in the `Aglet` class. The `dispatch` method causes an aglet to move from the local host to the destination given as the argument. The `deactivate` method allows an aglet to be stored in secondary storage, and the `clone` method spawns a new instance of the aglet, which has the state of the original aglet.

The Aglet class is also used to access the attributes associated with an aglet. The `AgletInfo` object, which can be obtained by `getAgletInfo()`, contains an aglet's built-in attributes, such as its creation time and code base, as well as its dynamic attributes, such as its arrival time and the address of its current context.

Now, let us demonstrate how simple it is to create a customized aglet. Start by importing the `aglet` package, which contains all the definitions of the Aglet API. Next, define the `MyFirstAglet` class, which inherits from the `Aglet` class:

```
import aglet.*;
public class MyFirstAglet extends Aglet { ... }
```

For instance, if we want an aglet to perform some specific initialization when it is created, we just need to override its `onCreation` method:

```
public void onCreation(Object init) {
   // Do some initialization here...
}
```

When an aglet has been created or when it arrives in a new context, it is given its own thread of execution through a system invocation of its `run` method. This invocation can be seen as a means of giving the aglet a degree of autonomy. The `run` method is called every time the aglet arrives at or is activated in a new context. One can say that the `run` method becomes the main entry point for the aglet's thread of execution. By overriding this method, it is possible to customize an aglet's autonomous behavior.

```
public void run() {
   // Do something here...
}
```

For example, one can use `run()` to let the aglet dispatch itself to some remote context by calling its dispatch method with the Uniform Resource Locator (URL) of the remote host as the argument. This URL should specify the host and domain names of the destination context, and the protocol (`atp`) to be used for transferring the aglet over the network. The URL can also include the name of the remote context (if more than one context is supported at the remote server). If no context name is specified, the aglet will move into the default context.

```
dispatch(new URL("atp://some.host.com/context"));
```

So what exactly happens to an aglet when `dispatch()` is called? Basically the aglet will disappear from the current host machine and reappear in the same state at the specified destination. First, a special technique called object serialization is used to preserve the state information of the aglet by making a sequential byte representation of the aglet. Next, this representation is passed to the underlying transfer layer that brings the aglet (byte code and state information) safely over the network. Finally, the transferred bytes are de-serialized to recreate the aglet's state:

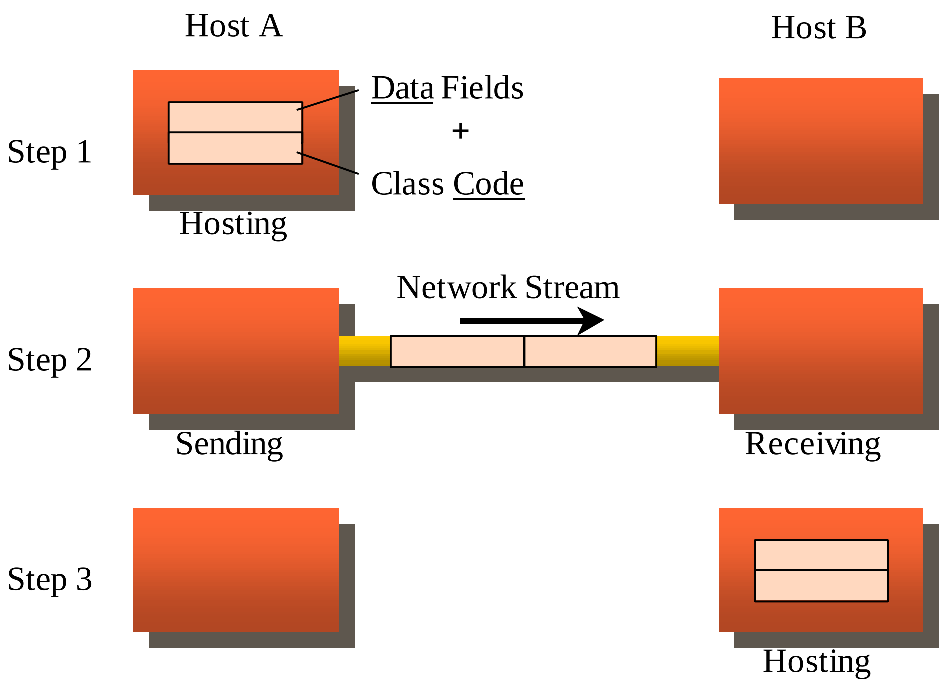


Figure 2 Transfer of an Aglet

Below is a tiny example in which we let the aglet named `DispatchingExample` dispatch itself. `DispatchingExample` extends the `Aglet` class and overrides two methods: `onCreation` and `run`. A *mobility listener* is defined as an inner class. This listener handles mobility-related events. We use a Boolean field, `_theRemote`, to distinguish between the aglet before and after it has been dispatched. When this aglet is created and starts running (`run()`), it creates a URL for its destination. When the aglet has been dispatched, all its threads will be killed. In other words, one should not expect the execution to return from a successful call to the `dispatch` method. When the aglet arrives at a new host, `onArrival` is called and the Boolean field is toggled. The aglet will now remain at this host until it is disposed of.

```
public class DispatchingExample extends Aglet {

   boolean _theRemote = false;

   public void onCreation(Object o) {
      addMobilityListener(
         new MobilityAdapter() {
            public void onDispatching(MobilityEvent e) {
               // Print to the console...
            }
            public void onArrival(MobilityEvent e) {
               _theRemote = true;  //-- Yes, I am the remote aglet.
               // Print to the console...
            }
         }
      );
   }

   public void run() {
      if (!_theRemote) {
         // The original aglet runs here
```

```
            try {
               dispatch(destination);
               // You should never get here!
            } catch (Exception e) {
               System.out.println(e.getMessage());
            }
         } else {
            // The remote aglet runs here...
         }
      }
   }
```

4.2 `AgletProxy` Interface

The `AgletProxy` interface acts as a handle of an aglet and provides a common way of accessing the aglet behind it. Since an aglet class has several public methods that should not be accessed directly from other aglets for security reasons, any aglet that wants to communicate with other aglets has to first obtain the proxy, and then interact through this interface. In other words, the aglet proxy acts as a shield object that protects an aglet from malicious aglets. When invoked, the `AgletProxy` object consults the *SecurityManager* [Karjoth *et al.* 1997] to determine whether the current execution context is permitted to perform the method. Another important role of the `AgletProxy` interface is to provide the aglet with location transparency. If the actual aglet resides at a remote host, the proxy forwards the requests to the remote host and returns the result to the local host.

Creating an aglet is one way to get a proxy. The `AgletContext.createAglet`[2] method will return the proxy of the newly created aglet. Other methods that return proxies include `AgletContext.retractAglet`, `AgletContext.activateAglet`, `AgletProxy.clone`, and `AgletProxy.dispatch`.

Proxies of existing aglets can also be obtained in the following ways:

- Retrieve an enumeration of proxies in a context by calling the `AgletContext.getAgletProxies` method.
- Get an aglet proxy for a given aglet identifier via the `AgletContext.getAgletProxy` method.
- Get an aglet proxy via message passing. An `AgletProxy` object can be put into a `Message` object as an argument, and sent to the aglet locally or remotely.
- Put an `AgletProxy` object into a context property by using the `AgletContext.setProperty` method, and share the proxy object.

4.3 `AgletContext` Interface

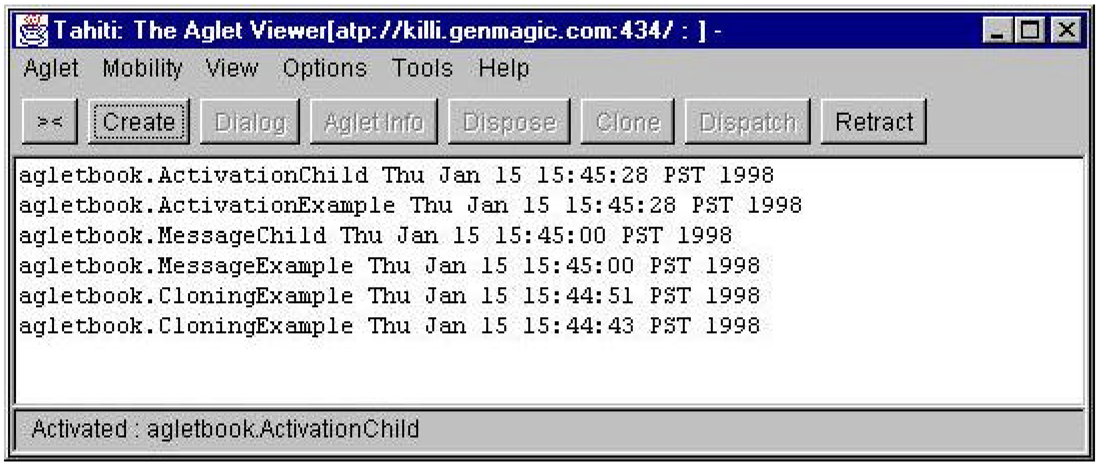
An aglet spends most of its life in an aglet context. It is created in the context, it goes to sleep there, and it dies there. When it travels in a network, it really moves from context to context. In other words, the context is a uniform execution environment for aglets in an otherwise

[2] We use the “dot-notation” (*class.method*) to denote the class that a given method belongs to.

heterogeneous world.

The `AgletContext` interface is used by an aglet to get information about its environment and to send messages to the environment, including other aglets currently active in that environment. It provides means for maintaining and managing running aglets in an environment where the host system is secured against malicious aglets.

IBM's Aglets Workbench comes with a graphical user interface for the context. It is called Tahiti and enables the user to create, clone, deactivate/activate, dispose of, dispatch, and retract aglets. Tahiti allows the user to monitor aglets running in the local context. Tahiti is basically an application built on top of the Aglet API. Developers can build proprietary versions of Tahiti that serve specific purposes.



```
agletbook.ActivationChild Thu Jan 15 15:45:28 PST 1998
agletbook.ActivationExample Thu Jan 15 15:45:28 PST 1998
agletbook.MessageChild Thu Jan 15 15:45:00 PST 1998
agletbook.MessageExample Thu Jan 15 15:45:00 PST 1998
agletbook.CloningExample Thu Jan 15 15:44:51 PST 1998
agletbook.CloningExample Thu Jan 15 15:44:43 PST 1998
```

Activated : agletbook.ActivationChild

Figure 3 A Screen Dump of Tahiti

4.4 `Message` Class

Aglets communicate by exchanging objects of the `Message` class. A string field named "kind" distinguishes messages. This field is set when the message is created. The second parameter of the message constructor is an optional message argument:

```
Message myName = new Message("my name", "Jacob");
Message yourName = new Message("your name?");
```

Having created the message objects, we can send them to an aglet by invoking one of the following methods defined in the `AgletProxy` class:

- `Object sendMessage(Message msg)`
- `FutureReply sendFutureMessage(Message msg)`
- `void sendOnewayMessage(Message msg)`

The Message object is passed as an argument to the aglet's `handleMessage` method. It is now up to this method to handle the incoming messages. It should return true if a given message is handled; otherwise, it should return false. The sender will then know if the aglet actually handled the message. In this example, the aglet will recognize and respond accordingly to `"hello"` messages:

```
public boolean handleMessage(Message msg) {
```

```
         if (msg.sameKind("hello")) {
            doHello();    // Respond to the 'hello' message...
            return true;  // Yes, I handled this message.
         } else
            return false; // No, I did not handle this message.
      }
```

Only from `yourName` do we expect to receive a return value:

```
      proxy.sendMessage(myName);
      String name = (String)proxy.sendMessage(yourName);
```

In the aglet's handleMessage method, we distinguish between the `myName` message and the `yourName` message by testing the "kind" field of incoming messages. From `myName` we extract the name argument, and for `yourName` we set the return value (`sendReply()`):

```
      public boolean handleMessage(Message msg) {
         if (msg.sameKind("my name")) {
            String name = (String)msg.getArg();  // Gets the name...
            return true;  // Yes, I handled this message.
         } else if (msg.sameKind("your name?")) {
            msg.sendReply("Yina");              // Returns its name...
            return true;  // Yes, I handled this message.
         } else
            return false; // No, I did not handle this message.
      }
```

What we have described here is the default implementation of the Message class. The programmer can create specialized message classes that provide more advanced messaging capabilities for the agents. As an example, we could add a KQML message class that provided some of the necessary abstractions required for agents to use KQML [Labrou and Finin 1997] for inter-agent communication.

5 Application of Aglets

Electronic commerce is one the fields that we expect to benefit from mobile agent technology. In this section we will describe an aglet-based framework that has been used to create a commercial service named Tabican.[3] Tabican is an *electronic marketplace* for air tickets and package tours (flight + hotel) that has been designed to host thousands of agents. In the server, shop agents wait for requests from consumer agents. Different shop agents may implement different sales policies, and consumer agents may implement different negotiation strategies. Users visiting this site can leave behind agents that will search for deals for up to 24 hours.

An IBM team has developed an electronic marketplace framework [Nakamura and Yamamoto 1997] on the basis of Aglets. An electronic marketplace is a multi-agent system in which selling and buying agents interact with each other. This architecture redefines the roles of participants, namely, marketplace owners, shop owners, and consumers. The role of marketplace owners is simply to manage system resources such as hardware and database systems. Users of marketplaces — that is, shop owners and consumers — are responsible for maintaining applications that are implemented as software agents. The framework is developed on top of Aglets and is called Aglet Meeting Place Middleware (AMPM). An important construct of AMPM is a type database that stores information on the types of messages between agents. The type database enables unfamiliar agents that were independently developed to interact with each

[3] The URL is `http://www.tabican.ne.jp`. Unfortunately, this service is available only in Japanese.

other. See the figure below for an overview of the electronic marketplace.

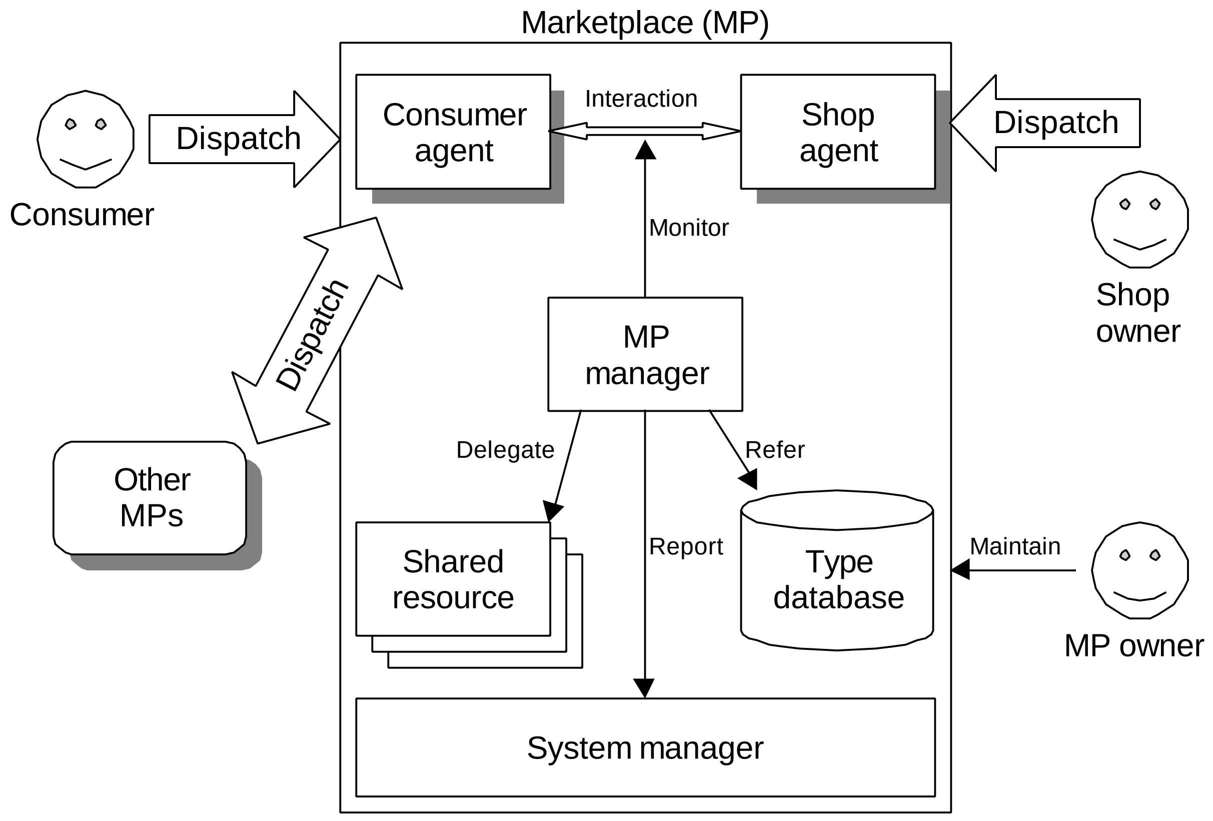


Figure 4 The Aglet Marketplace Architecture

From the perspective of electronic marketplace implementation, the key functions in Aglets are agent mobility and messaging. Mobility makes possible the following agent activities in a marketplace:

- Shop agents go to a market from a shop owner's terminal.
- Customer agents travel around various markets to get more information.
- Market advertisers go to other markets to invite customer agents.

These activities were easily implemented by simply using the built-in mobility in Aglets. In the actual implementation, the aglet programming model was reported to be useful, since the marketplace agents' activities fit nicely into the programming model. The aglet programming model defines events such as creation, cloning, dispatching, retraction, disposal, deactivation, and activation. Then, on the basis of these events, listeners and associated methods, such as, `onCreation()`, `onDispatching()`, `onArrival()` and `onDeactivating()`, are defined.

A typical behavioral pattern of the customer agent is (1) load type information, (2) advertise a request, (3) receive the results, and (4) go to another market. In (1), an agent obtains an interaction protocol in order to talk with unfamiliar agents: this activity is carried out in `onCreation()` and in `onArrival()`. Steps (2) and (3) are ordinary activities within a marketplace. Finally, (4) causes a dispatching event and `onDispatching()` is invoked. Within this function, the customer agent is removed from the list of registered participants in the electronic marketplace.

For the electronic marketplace, the key aglets constructs related to interaction are the `AgletProxy` and asynchronous messaging. An important purpose of the electronic marketplace is to develop agents that are self-serving. Aglet proxies and asynchronous messages directly contribute to the achievement of this purpose. An aglet proxy is an interface object for an aglet: aglets cannot talk to each other directly, but must talk through a proxy. This means that an aglet cannot directly manipulate another aglet, and it provides a way of protecting aglets from each other. Asynchronous messaging is also important, if we assume that unfamiliar agents have to interact, since it minimizes the possibility of two agents hung on account of a protocol mismatch is minimized. In general, the team developed agents so that they can be executed independently.

Since agents can be developed independently by different people, the agent system must be flexible in the sense that adding a new agent or updating an existing agent can easily modify its behavior. For example, one should be able to replace a shop or a customer agent with an updated agent as long as the new agent behaves in accordance with the defined interaction protocol. When developing the electronic marketplace, the team first implemented a single market application, and then transformed it into a multiple market configuration. What they did was as follows: first, they added a protocol for advertising a market; next, they created a new agent, that is, the Market Advertiser; and finally, they updated the customer agent so that it could move to another market in response to an advertiser's message. The actual modification of the customer agent consisted in the addition of the necessary code for going to another market.

On the basis of the team's observations, we see promising signs that the use of agent technology such as Aglets ultimately will lead to a new approach to software development and maintenance. The trend toward improved encapsulation and delegation that was started by object technology will perhaps be taken to new levels by agent technology.

6 Contemporary Mobile Agent Systems

So what kind of mobile agent systems are currently available? Fortunately, Java has generated a flood of experimental mobile agent systems. Numerous systems are currently under development, and most of them are available for evaluation on the Web.

The field is developing so dynamically and so fast that any attempt to map the agent systems will be outdated before this article goes to press. We will, however, mention a few interesting Java-based mobile agent systems: Odyssey, Concordia, and Voyager. Notice that we do not attempt to compare any of these systems.

Odyssey. General Magic Inc. invented the mobile agent and created the first commercial mobile agent system, called Telescript. Being based on a proprietary language and network architecture, Telescript had a short life. In response to the popularity of the Internet and later the steamrollering success of the Java language, General Magic decided to re-implement the mobile agent paradigm in its Java-based Odyssey. This system effectively implements the Telescript concepts in the shape of Java classes. The result is a Java class library that enables developers to create their own mobile agent applications. More information is available at `http://www.genmagic.com/html/agent_overview.html.`

Concordia. Mitsubishi's Concordia is a framework for the development and management of mobile agent applications that extends to any system supporting Java. It consists of multiple components, all written in Java, which are combined together to provide a complete environment for distributed applications. A Concordia system, at its simplest, is made up of a standard Java virtual machine, a server, and a set of agents. Like Aglets, Concordia provides a security manager that allows for secure execution of agents. It also supports a checkpoint mechanism for fault tolerance. Again, like aglets, it provides event-based communication. More information is available at `http://www.meitca.com/HSL/Projects/Concordia`.

Voyager. ObjectSpace's Voyager is a platform for agent-enhanced ORB (Object Request Broker) for Java. Voyager not only provides an extensive set of object messaging capabilities, but also allows objects to move as agents in the network. Voyager combines the properties of a Java-based object request broker with those of a mobile agent system. In this way, it allows Java programmers to create network applications using both traditional and agent-enhanced distributed programming techniques. One drawback of Voyager is that it provides no security management for untrusted agents. More information is available at `http://www.objectspace.com/voyager`.

We would like to note that Java-based mobile agent systems have a lot in common. Besides their programming language, they all rely on standard versions of the Java virtual machine and Java's object serialization mechanism, and a common server-based architecture permeates all the systems. However, their agent transport mechanisms and support for interaction (messaging) vary considerably.

7 Conclusion

We have given a brief introduction to the world of mobile agents. In particular, we have conducted a quick tour of the essentials of the Aglet API and its underlying programming model. Aglets' event-based programming model emphasizes delegation and reuse of components such as listeners that efficiently define agent behavior. Code fragments of aglets have been shown in order to convince the reader that there is nothing mysterious about programming mobile agents in Java.

In addition, we have reported on one of the first real Internet applications based on mobile agent technology, namely, the Tabican electronic marketplace in Japan. This and future projects will demonstrate that Java-powered mobile Internet agents have a role to play outside the research labs. We envision a not-too-distant future when mobile agent technology is an integrated part of the network or perhaps even the operating system. This process will accelerate with the adoption of Java as a universal language system and with standardization efforts such as the Object Management Group's Mobile Agent System Interoperability Facility (MASIF) [OMG 1997].

The Aglets project has been a very successful research project that has exceeded even our wildest imagination. We are confident that the entire field of mobile agents will continue to be an interesting and exciting place on the map in the years to come.

8 Acknowledgements

We wish in particular to thank the past and present members of the *Aglets Team* in the IBM, without whom aglets would not have made it. They include: Günter Karjoth, Kazuya Kosaka, Yoshiaki Mima, Dr. Yuichi Nakamura, Kouichi Ono, Hideaki Tai, Kazuyuki Tsuda, Gaku Yamamoto, Dr. Yariv Aridor, and Dr. Tsutomu Kamimura. We are also grateful to Mike McDonald of IBM Japan for checking the wording of this paper.

9 References

Aridor, Y. and D.B. Lange (1998), "Agent Design Patterns: Elements of Agent Application Design," To appear in *Proceedings of the Second International Conference on Autonomous Agents (Agents '98)*, ACM Press.

Arnold, K. and J. Gosling (1998), *The Java Programming Language*, Second edition, Addison-Wesley, Reading, MA.

Karjoth, G., D.B. Lange and M. Oshima (1997), "A Security Model for Aglets," *IEEE Internet Computing 1*, 4, 68-77.

Labrou, Y. and T. Finin (1997), "A Proposal for a new KQML Specification," Technical Report CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD.

Lange, D. B. and M. Oshima (1998), *Programming and Deploying Mobile Agents with Java*. Forthcoming book. Addison-Wesley, Reading, MA.

Nakamura, Y. and G. Yamamoto (1997), "An Electronic Marketplace Framework Based on Mobile Agents," Research Report, RT0224, IBM Research, Tokyo Research Laboratory, Japan.

The Object Management Group (1997), "The Mobile Agent System Interoperability Facility," OMG TC Document orbos/97-10-05, The Object Management Group, Framingham, MA.

White, J. (1996), "Telescript Technology: Mobile Agents," In *Software Agents,* J. Bradshaw Ed., MIT Press.