*The* UNIVERSITY *of* EDINBURGH
SCHOOL *of* COMPUTER SCIENCE

# CS4/MSc

# Distributed Systems

Jane Hillston

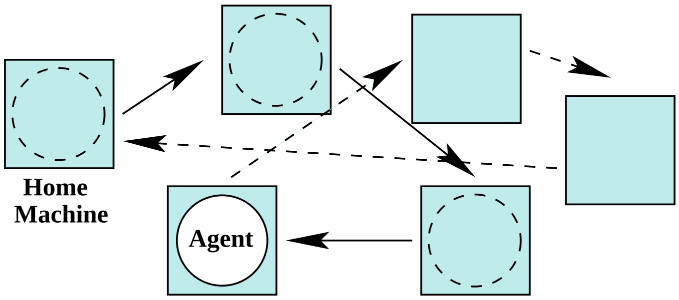
jeh@dcs.ed.ac.uk

Room 1604

(Lecture 17: Mobile Agents, 4th March 2002)

## Mobile agent: basic definition

- An autonomous program that can move from machine to machine in a heterogeneous network under its own control.
- It can suspend its execution at any point, transport itself to a new machine and resume execution on the new machine at the point at which it left off (`GO` instruction).
- On each machine, it interacts with service agents and other resources to accomplish its task, returning to its home site with a final result when that task is finished.




## `GO` Instruction

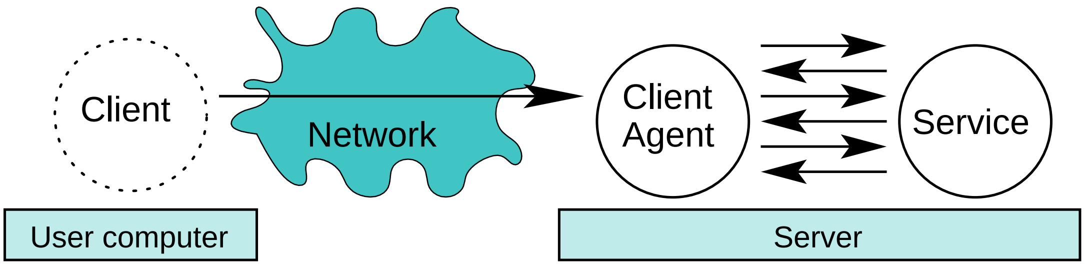
1. Current agent process is suspended or new agent daughter process is created;
2. Suspended or new process, including its state (and context), is collected and processed into a message;
3. Message may be addressed explicitly to final destination or may be routed depending on its content;
4. The server's message sub-system delivers the message to the agent execution environment;
5. The received message is reconstituted into the executable and the process is dispatched;
6. Execution continues at the next instruction in the agent program.

## Conceptual model


An application is implemented as a number of interacting agents, which may be local or remote. Each agent will executes in a "**place**" which is the execution environment on the host computer. The communication infrastructure must handle transport of both agents and messages: both are transmitted as streams of bytes.

## Mobile Agents

Process-to-process communication not only allows one process to call methods on another, but also to supply the methods to be performed. The mobile agent is a computational element that the sending process began but which the receiving process will continue; the data transmitted is the current state.

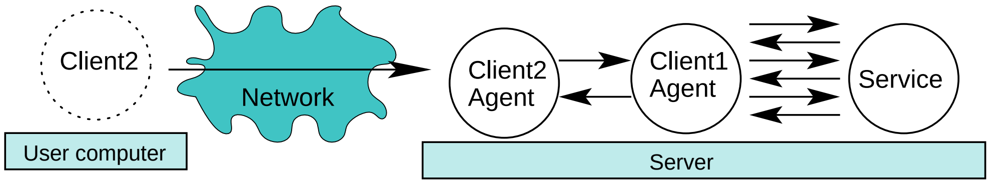



*ongoing interaction does not require ongoing communication*
*(c.f. RPC)*

## Advantages over RPC

- Network connectivity does not need to be maintained throughout interaction: support for disconnected operation.
- Less network traffic which leads to improved performance, especially for networks with low throughput, high latency or high cost, e.g. wireless networks, POTS.
- Secure transactions carried out via RPC have a very high overhead; whilst such transactions still incur an overhead with mobile agents it is generally lower.
- Mobile agents are both practical and appealing to the Internet community since they can provide an effective means for dealing with problems of finding services and information and since they empower the individual user.

## Advantages (cont.)




- Customised services may be built on top of server applications. For example, if the age-dependent file deleting agent could be invoked by other clients the server now appears to offer an enriched service. (*public networks treated as platforms*).
- As well as providing efficient support for existing services, a mobile agent framework also enables new, derivative network services and hence new business opportunities; e.g. "consultant agents".
- Remote real-time control (extreme example: space probes) becomes feasible, and error detection and recovery may be greatly enhanced. Similarly remote interactions are more robust.

## Example Application: Information Foraging

- Mobile agent starts on user's home machine (laptop) where it asks the user for a free-text query.
- The agent then travels to each site in a distributed collection of technical reports, where it interacts with a dedicated information retrieval agent to retrieve relevant documents.
- As it travels the agent merges and organises the results from each collection.
- The agent does not travel sequentially through the collection sites. Instead, it sends out child agents to search the collections in parallel.
- May modify its behaviour dynamically. For example, the parent agent may choose to migrate to a proxy within the network if the link from the laptop is unreliable or slow; it may decide not to send a child agent but instead use messaging if the resident information retrieval agent is sophisticated.

## Example Application: Electronic Commerce

- Mobile agent originated on Mary's work PC but has a regular commitment to organise her "Friday Night". It is resident in an electronic marketplace remaining dormant for most of the week.
- Friday morning it meets with the Video-on-demand agent and selects a film to be screened on Mary's TV at home at 9pm that evening. It keeps track of which films she has already seen and knows to avoid anything violent.
- It then meets with the Pizza agent and orders Mary's favourite pizza to be delivered to her home at 8:50pm.
- The agent then sends messages to Mary at work and her husband Chris to remind them of the evening's plans and inform them of the movie selected.

## Example Application: Information Dissemination

- Mobile agents embody the so-called *Internet push* model of information dissemination.
- When a vendor wants to disseminate a software update, it creates an agent for each customer containing the new software together with installation procedures.
- These procedures may be tailored to the customer's individual machine based on information stored on the vendor's customer database.
- The agent travels to the customer's personal computer bringing with it the new software components and the installation procedures.
- Once relocated the agent autonomously carries out the update and manages the software on the computer, possibly replacing a previous agent dispatched with an earlier release.

## Java Support for Agent Development

Although there are exceptions, (e.g. *D'Agents* from Dartmouth College), mobile agent systems have been predominantly developed using Java.

**Platform Independence** This is crucial if agents are going to move freely around a heterogeneous network.

**Secure Execution** The Java security model is taken as the basis for securing servers against agents.

**Dynamic Class Loading** Loading and defining classes at run-time provides a protective name space for each agent, thus allowing agents to execute safely and independently of each other.

**Multi-threaded Programming** Threads are used for the separate computation elements represented by agents, thus several agents may be concurrently active within a server.

**Object Serialization** Serializability is a key feature of mobile agents so the built-in serialization mechanisms in Java are a big bonus.

**Reflection** Reflection allows agents to discover information about themselves and other agents.

## Drawbacks of Java for Agent Development

Some features of Java can be viewed as a disadvantage. They can be circumvented but they have influenced how systems have developed.

**Inadequate Support for Resource Control** Java provides no way for the host to limit the processor or memory resources allocated to a given object or thread. Allocation of some resources such as files, sockets and windows can be controlled but are not bound to the agent responsible, meaning that they may be simply "abandoned" when an agent moves on.

**No Object Ownership of References** Objects/agents cannot explicitly void themselves until all references to the object have been voided. This means that some other agent may keep an agent alive by retaining a reference to it.

**No Support for Storing of Execution State** Since information such as the program counter and frame stack is not accessible to Java programs the full execution state of a program cannot be stored and retrieved automatically.

## The Aglet Model: IBM's mobile agents

**An Aglet** is a mobile Java object that visits aglet-enabled hosts in a computer network. It is autonomous and reactive.

**A Context** is an aglet's workplace. It is a stationary object that provides a means for maintaining and running aglets in a uniform execution environment. One node in a network may host many contexts.

**A Proxy** is a representative of an aglet. It protects the aglet from direct access to its public methods and hides its true location, providing location transparency.

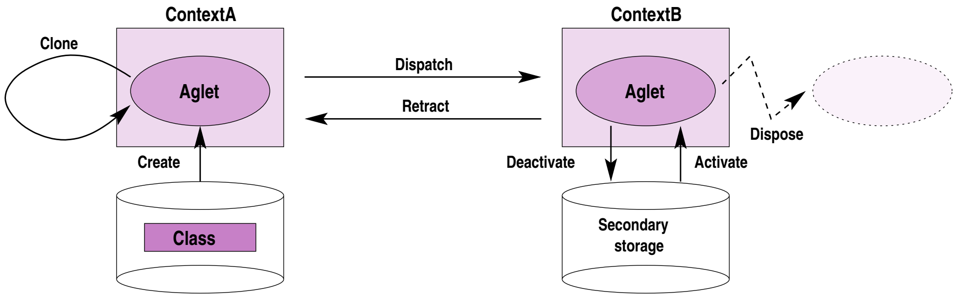
**A Message** is an object exchanged by aglets. It allows for synchronous and asynchronous message-passing between aglets.

**A Message Manager** provides concurrency control of incoming messages.

**An Itinerary** is an aglet's travel plan, providing a convenient abstraction for non-trivial travel patterns and routing.

**An Identifier** is bound to each aglet; it is globally unique and immutable.

## Basic Aglet Behaviour




- The **creation** of an aglet takes place in a context. The new aglet is assigned an identifier, inserted in the context, and initialised. The aglet starts executing as soon as it has successfully been initialised.
- The **cloning** produces an almost identical copy in the same context. The only differences are the assigned identifier and that execution restarts in the new aglet. N.B. execution threads are not cloned.
- **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 will not migrate).

## Basic Aglet Behaviour (2)

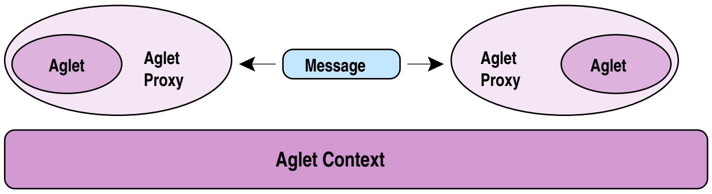
- The **retraction** of an aglet will remove it from its current context and insert it into the context from which the retraction was requested.
- The **deactivation** of an aglet is the ability to temporarily remove it from its current context and store it in secondary storage. **Activation** of an aglet will restore it in a context.
- The **disposal of** an aglet will halt its current execution and remove it from its current context.

| **Event** | **(as event occurs)** **Methods** | **(after event)** |
|---|---|---|
| Creation | | `onCreation()` |
| Cloning | `onCloning()` | `onClone()` |
| Dispatching | `onDispatching()` | `onArrival()` |
| Retraction | `onReverting()` | `onArrival()` |
| Disposal | `onDisposing()` | |
| Deactivation | `onDeactivating()` | |
| Activation | | `onActivation()` |
| Messaging | `handleMessage()` | |

## Sample Aglet

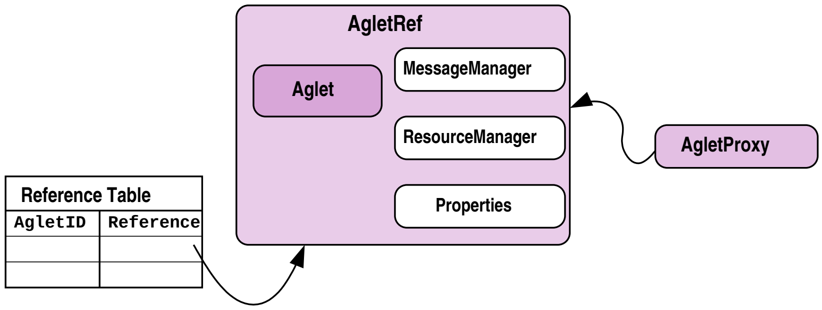
```
public class HelloAglet extends Aglet {
   public void onCreation (Object init) {
      System.out.println("created!");
   }
   public void run() {
      System.out.println("Hello!");
   }
   public boolean handleMessage(Message msg) {
      if (msg.sameKind("sayHelloAgain") {
          System.out.println("Hello!);
          return true;
      }
      return false;
   }
   public void onDisposing() {
      System.out.println("bye!");
   }
}
```

## Aglet Proxy(1)




- The `AgletProxy` object acts as a handle for an aglet for security reasons. Any aglet that wants to communicate with an aglet has to first obtain the proxy object, and then interact through this interface.
- When invoked the proxy object consults the SecurityManager to determine whether the caller is permitted to perform the method.
- Also provides location transparency for remote invocation.
- When the `Aglet.clone()` primitive spawns a new instance of the aglet it returns an AgletProxy object not an Aglet object.
- AgletProxy objects are automatically generated.

## AgletRef Object




- The `AgletRef` class is the true implementation of the `Aglet` class, e.g. key functions such as dispatching, retraction, cloning.
- The AgletRef object also maintains aglet properties such as identity and address.
- Each AgletRef is stored in the *aglet reference table*.
- AgletProxy objects also contain references to the AgletRef object; multiple proxy objects can refer to the same AgletRef object.

## AgletContext Interface

- An aglet **context** provides the **agent execution environment**.
- An aglet uses the `AgletContext` interface to get information about its environment and to send messages to the environment, including other aglets currently active in the environment.
- This interface provides means for maintaining and managing running aglets in an environment where the host system is secured against malicious aglets.
- The `Aglet` class has a method for gaining access to its current context: `context = getAgletContext();`
- Once it has access to the context, it can create new aglets, retract remotely located aglets into the current context and retrieve a list of proxies of its fellow aglets in the context.

## Aglet Creation

An aglet must be instantiated within a context: an aglet relies on services provided by the context to be properly instantiated and initialised. The `createAglet` method is used to create new aglets:

```
AgletContext.createAglet(URL codeBase,String code,Object init);
```

The three arguments are:

`codeBase` specifies the base URL of the aglet class file. If `codeBase` is `null`, the context will search for the code in the local system's aglet search path `AGLET_PATH`).

`code` gives the name of the class of aglet to be instantiated.

`init` is an object passed to the aglets `onCreation` method.

After creation, the aglet is inserted into the context and started—invocation of the method `onCreation()` followed by `run()`.

## Messaging in Aglets

There are three types of messages, and the `AgletProxy` class provides the following methods for sending them:

**Now-type:** (*blocking, synchronous*) `sendMessage(Message msg)`

**Future-type:** (*non-blocking, asynchronous*) `sendAsyncMessage(Message msg)` returns a `FutureReply` object which may be used to collect the response later.

**One-way:** (*non-blocking, asynchronous, no response*) `sendOneWayMessage(Message msg)`

- Every aglet object has a message queue object.
- Messages are handled in *transmission order* subject to priorities.
- A message marked as `NOT_QUEUED` will be handled concurrently.
- Messages are transferred by the Agent Transfer Protocol (ATP) (uses URLs for agent resource location) by default.

## Agent Transfer Protocol (ATP)

An ATP request consists of a request line, header fields, and a content. The request line specifies the method of the request while the header fields contain the parameters of the request. ATP defines four standard request methods.

**Dispatch** requests a destination agent system to reconstruct an agent from the content and to start executing it.

**Retract** requests a destination agent system to send a specified agent back to the sender.

**Fetch** is similar to Get in HTTP: it requests a receiver to retrieve and send any identified information (normally class files) back to the sender.

**Message** is used to pass a message to an agent (proxy) identified by an identifier and to return a reply value in response.

## Object Mobility

- When an aglet is dispatched, cloned or deactivated, it is marshalled into a byte array, from which it will be unmarshalled later.
- Aglets use the standard Java ObjectSerialization mechanism for marshalling and unmarshalling
- Through the serialization/deserialization process ordinary objects that are part of the state of an aglet are *moved by value*. Thus an object which was shared within a context by multiple aglets is *copied* and *is no longer shared* after `dispatch`, `clone` or `deactivate/activate` operations.
- When an aglet proxy is transferred it keeps the aglet id and its address and uses that information to restore the correct reference to the original aglet. Therefore AgletProxy objects keep the reference to the actual aglet even if the proxy is transferred to a remote host or deactivated, as long as the aglet remains at the same location.