Architectural guidelines for Mobile Agent Systems

M.A. Schoeman & E. Cloete
School of Computing
University of South Africa
March 2003
{schoema,cloete@unisa.ac.za}


1. Introduction

In [12], Lange claims that lack of a programming model for agent-based applications prevents wider mobile agent deployment, while Johansen argues that it is easy for a systems programmer to build and deploy a mobile agent, but that novice users need a better way to create agents. Kendall et al [9] reason that agent development to date has been done independently, leading to problems such as
could assist in a more widespread use of mobile agents. The purpose of this literature survey is to gain a knowledge base of the current practices in programming mobile agents in order to formulate such a set of guidelines for creating/programming and using a mobile agent efficiently.

2. When Should a Mobile Agent be Used?

Mobile agents are seen as a new design technique [11] or a new design paradigm [9, 14, 17], not yet fully incorporated in mainstream programming, and as such it follows that guidelines for creating/programming and using a mobile agent efficiently, should include some reference to the circumstances under which mobile agents are beneficial, though it is not the main focus of this literature survey.

Two main areas in which mobile agents proffer considerable advantages have been identified, namely: systems and distributed management [3, 4, 6, 9, 12, 14, 15, 19] and information retrieval [3, 5, 12, 15]. Other areas where mobile agents are seen as offering potential advantages, are disconnected computing, also known as wireless or mobile computing [8, 12, 15], dynamic deployment of code [9, 12], ‘thin’ clients or ‘resource-limited’ devices [12, 15], personal assistants (used in e-commerce for example) [8, 15], and mobile agent-based parallel processing [8, 15].

The claim by Wong et al [19] that mobile agents overcome the inherent limitations in the client-server paradigm, namely that this paradigm breaks down for distributed applications when dealing with highly distributed problems, slow and/or poor quality network connections and in the maintenance of constantly changing applications, attests the previous paragraph. Gschwind et al [6], on the other hand, maintain that there is a class of tasks where mobile agents are more beneficial than other approaches (e.g. client-server paradigm): relatively simple management actions need to be conducted, the agent’s itinerary is well-defined and the global operation and side-effects of the algorithm are well understood.

Gray et al [5], focusing on information-retrieval, argue that mobile agents allow unpredicted information-retrieval tasks, but will not always perform better than a client-server solution. If the agent code is larger than the intermediate data, for example, the mobile agent must perform worse. If the network speed is high enough the mobile agent might perform worse even if the code is smaller, because mobile agents are typically written in interpreted languages. Lower network speed and reliability, or larger data volumes, can change the situation notably, however.

Picco [14], while acknowledging the advantages of mobile agents, points out that mobile agents are only one of a number of design paradigms for code mobility, and that a naive use of mobile agents may lead to inefficient designs. Kotz et al [11] confirm that different forms of mobility are useful in different situations. In [12] Johansen also maintains that we should make sure that it is beneficial for the design and the design’s outcome before we deploy mobile agents. This concept is implemented by the distributed management architecture defined by Feridun and Krause [4], that does not force the programmer to follow a specific programming paradigm such as client-server or mobile agents, but instead allows him to use the most appropriate paradigm or combination of paradigms.

In [12], Johansen, Kotz and Petrie all indicate performance as motivation for deploying mobile agents, while (in the same paper) Lange [12] adds flexibility, which Johansen [12] (also in the same paper) terms ”the ability to customize for the Internet”. In [12] Kotz also cites scalability as motivation for using a mobile agent. Kotz et al [11] maintain that we need quantitative measurements of the value of each form of mobility as well as implementations of actual applications to demonstrate the benefits of mobile agents by means of meaningful analysis. Picco [14] agrees that a body of literature providing evidence about when and to what degree mobile agents are useful, is needed.

Kotz et al [11] point out that it is important to distinguish between using mobile agent concepts as an architecture for applications, and using a specific mobile agent system to implement applications. To add mobility to an existing application with a mobile agent system, programmers typically have to change their application significantly to conform to the mobile agent system’s language and constraints.
The agent infrastructure includes both the network components as well as the software (i.e. services and components) used to ensure agent computing [2, 15, 16].

According to Yaridor and Oshima [2] a basic mobile agent infrastructure consists of “agent servers” and “agent clients”. The agent servers are hosts with fixed network connections, which run agent engines to create, run, and launch agents onto the network or to receive agents from the network and allow them to continue their execution. The agent engine consists of a language interpreter and an agent system composed of a security manager, shared resources (e.g. whiteboards) and runtime services for agents (e.g. dispatch and communication). Agent clients are software environments that allow users to create and control agents locally or directly at remote hosts. Harrison et al [7] call agent clients the API (application programming interface) to mobile agents, while agent servers are termed the agent execution environment. They point out that the agent execution environment (agent servers) needs to bind to the message transport service (i.e. the network components) in order to send and receive mobile agents in the communication structure. Gray et al [5] also indicate that the D’Agents mobile-agent system uses the TCP/IP transport mechanisms to transfer mobile agents.

Mobile agent facilities are described by various authors, and often each determine their own terminology to define elements of the infrastructure. Some examples of mobile agent facilities follow:

1. Wong et al [19] developed a MAF compliant mobile agent platform, called Scalable Mobile and Reliable Technology (SMART), incorporating dynamic aggregation (not yet part of MAF). SMART consists of four layers built on a Java virtual machine. The Region Administrator is the lowest layer (built on the JVM), and manages and enforces security policies on a set of agent systems. The Finder module in the Region Administrator provides the naming service to the region administrators and the layers above. In the second-lowest layer, the Agent System layer, mobile agents can create, migrate and destroy themselves. Agents execute in the following layer, the agent context layer, called the Place. Multiple agent contexts (places) can exist. The topmost layer, the Agent Proxy layer, provides the mobile agent API for applications written in SMART.

2. Gray et al [5] describe the D’Agents (formerly known as Agent Tcl) mobile-agent system, developed to support distributed information retrieval and to characterize the mobile agent performance space. The D’Agents’ architecture has five levels: transport mechanisms (TCP/IP), a server running on each machine to accept mobile agents, a shared C++ library that implements agent functionality, an execution environment for each of the three supported languages (Java, Tcl and Scheme) and the agents themselves. Each execution environment includes the interpreter/virtual machine for the supported language, stub routines allowing the agent to invoke functions in the C++ library, a state-capture module to support mobility and a security model to enforce resource limits. The server, (a multi-threaded process) executes multiple Java agents as threads inside a single process, while each Tcl and Scheme agent is executed inside its own Unix process. Of interest is that the server spawns some ‘hot’ interpreters on start-up and assigns an incoming agent to the first free hot interpreter. This provides a dramatic performance improvement.

---

1 A common mobile agent facility (MAF) specification came out early in 1997 to allow mobile agents developed on different commercial mobile agent platforms, to interact.

2 Aggregation is defined as “a-part-of relationship” in which components of different objects are associated as a unit. Dynamic aggregation is seen as a practical implementation of subject-oriented programming, a programming composition technique that supports building object-oriented systems as compositions of subjects. A subject is a collection of classes/class fragments whose hierarchy models its domain in a subjective way and can be a complete application or only a fragment that needs to be composed with other subjects to create a complete application. In subject composition class hierarchies are combined to produce new subjects that include the functionality of the existing subjects. Dynamic aggregation therefore enhances the properties of an object at runtime in unforeseen ways, by e.g. forming relationships with...
4. Gschwind et al [6] developed a framework in which mobile agents used to assist with network and systems management, is composed of a set of categories of components: navigators, performers and reporters. The framework/architecture consists of an agent construction toolkit (Agent Bean Development Kit (ADK)) and a plug-in abstraction layer for existing mobile agent platforms to allow execution of mobile agents created with the ADK.


MASIF (the Object Management Group (OMG) Mobile Agent System Interoperability Facility) [13] defines an agent system as a platform that can create, execute, transfer and terminate agents. A host can contain more than one agent system. Yaridor and Oshima’s [2] definition of an agent engine, SMART’s Agent System layer [19] and D’Agents’ [5] execution environment correspond with this. According to the tasks executed in it (a security manager, shared resources and run-time services for agents), Yaridor and Oshima’s [2] agent system also corresponds to MASIF’s definition of an agent system, though apparently it does not include a place.

MASIF [13] defines a place as a context in which an agent executes. Agents migrate between places, a place can host one or more agents, and one or more places can exist in an agent system (using MASIF’s definition of an agent system). If an agent system does not support places, it acts as default place itself. Wong et al [19] also calls the context in which an agent executes, a place. While Tripathi [17] does not distinguish between agent servers and places, he indicates that they provide a confined execution environment for visiting agents, grant access to local resources in a selective manner, and allow agents to migrate to other servers, to communicate and query their environment.

The Agent server accept mobile agents and run one or more agent systems for the agents to execute in. Horvat et al [8] describe the agent server as a sandbox which provides all the services necessary to execute a mobile agent, but limits their actions. It enables agent transfer, allows multiple agents to live and execute on the same server without interfering with each other (by providing agent systems or places) and allows agents to communicate with each other. This is inline with the purpose of the agent system. According to Wong et al [19] the agent server itself will manage and enforce security policies and provide a naming service, though Gray et al [5] provide a security model to enforce resource limits in the agent system. Yaridor and Oshima [2] also locate the security manager and shared resources in the agent system. The agent system allows agents to create, migrate and destroy themselves [19], and provides run-time services for agents (such as dispatch and commmunication) [2]. The agent system also contains an interpreter and one or more places providing a context for agents to execute in. Mobile agent systems that obtain classes on demand, has a network class loader installed as part of the place, to download class information and define the classes [19].

Though various agent infrastructures may place services in different environments, most mobile agent systems provide access to local resources, mobility, communication and at least some form of security.

3.2 Features of Mobile Agent Systems

The most important characteristics of a mobile agent is its mobility and autonomy [8]. To support this, mobile agent
As most mobile agent systems are Java-based, the majority of them use the weak mobility model [8, 14, 17]. Systems not based on Java often provide strong mobility [14], though D’Agents support strong mobility for both Java and Tcl through modifications to the respective interpreters [5]. Systems with weak mobility vary in the way agents request to move. Some use a ‘go’ statement causing the agent code and data to be moved to a new host, where a predetermined procedure or method is invoked. Others associate an itinerary listing the succession of hosts to visit, with each agent, and the method to invoke on each host. Some systems invoke the same method at each stop, while others allow the agent to specify different methods for each stop. Systems using itineraries are usually event-driven [5, 17].

According to the MASIF standard [13] for interfaces at agent system level, to process a mobile agent’s request to move, the source agent system executes the following algorithm:

1. Suspend the agent (halt the agent’s thread)
2. Identify the agent’s state
3. Serialise the Agent class and state
4. Encode the serialised Agent class and state according to the transport protocol
5. Provide authentication information to the server
6. Transfer the agent.

Before accepting the agent, the receiving agent system verifies that it can support the agent profile (agent system type, e.g. Aglet, language and serialisation method). If it does, the following algorithm is executed:

1. Authenticate the client sending the agent
2. Decode the agent
3. Deserialise the Agent class and state
4. Instantiate the agent
5. Restore the agent state
6. Resume agent execution.

The agent code may require class transfers when an agent is created remotely or transferred - the Agent class is needed to instantiate the agent and must be transferred from the source if it does not exist on the host. Other classes used during agent execution, if they are not available at the destination, must also be transferred from the source or from a server [13].

All classes required by the agent code can be transported as part of the agent transfer protocol, or classes can be obtained on demand from a designated code-base server during execution [5, 13, 17]. D’Agents use the first approach [5]. Though this makes agent transfer heavyweight, since more classes than the agent actually need, may be transferred, it has the advantage that the agent needs no further communication with its previous host for code transfer [17]. SMART uses the second approach, through the process of dynamic aggregation, which allows the agent to attach new code and data at runtime, reducing the amount of code transferred with the agent. It also reduces bandwidth requirements and speeds up the packing process before transmitting an agent to another host [19], but slows down agent execution [17] and is not suitable for disconnected operations [5, 13, 14, 17]. Alternatively a third-party could be asked to serve as code cache to allow the sending host to disconnect, or code could be cached at each host [5]. Variations of the two approaches occur, such as (a) using automatic transfer of all classes when creating an agent remotely and automatic transfer of the Agent class with transfer on demand of other classes when transferring an agent, or (b) sending a list of class names of all the classes necessary to execute the agent, allowing the destination to only request those classes not cached [13]. Another variation is mentioned by Gray et al [5] and Picco [14]: \mu code examines the bytecode for each class it transfers and thereby determines which additional classes should be send.

To execute an agent, the agent system creates an instance of the Agent class within a default place or a place specified by the client. The Agent class specifies the interface and the implementation of the agent, which executes on its own thread
The actual agent migration between hosts is accomplished either with RMI, or using sockets. In RMI the sending agent server initiate agent transfer by invoking a Java public method on the remote host to request the transfer. The remote host directs the transfer by invoking the \texttt{beginTransfer} method on the sending host; the sending host serializes the agent, and prepares it for transfer. The remote host uses RMI to transfer the serialized agent and its data. Once the transfer is completed, the remote host informs the sending host and restarts the agent. When using sockets, the agent data and code is converted to a protocol-independent byte array and transferred with standard transport protocols (e.g. TCP/IP) [8].

3.2.2 Naming Services

A global naming scheme and name service are needed to locate resources, specify agent servers for migration and to establish inter-agent communication [17], while agent names are used for identification, controlling and locating agents [13]. A naming scheme defines a namespace, which may be hierarchically structured. A location-independent naming scheme in which the name of an agent does not change on migration, simplifies programming since current locations of entities need not be taken into account [2, 13, 17]. MASIF [13] specifies that agents are named by their authority (person/organisation for whom the agent acts), identity (a unique value within the scope of the authority) and agent system type (e.g. Aglet). This combination provides a globally unique name for each agent. A name service supports mapping a resource name to its physical address and could also indicate the type of resource and the protocol to access it [17].

MASIF [13] defines a MAFFinder interface as a naming service. This interface provides methods to maintain a database of agents, places and agent systems, as well as operations to register, unregister and locate these objects. A range of techniques to find an agent are provided:

- \textbf{Brute force} - The agent is found by searching for it in every agent system in the region. (A region is a set of agent systems of the same authority.)
- \textbf{Logging} - The agent is followed by its trail, indicated by leaving its next destination at each server it visits.
- \textbf{Agent registration} - Every agent registers its current location in a database that keeps the latest information about agents’ locations.
- \textbf{Agent advertisement} - Only places are registered, and an agent’s location is registered only when the agent advertises itself. Non-advertised agents are found by brute force searching or logging.

Yaridor and Oshima [2] also discuss brute force, logging and registration as methods for locating agents to establish inter-agent communication and remote agent management. They add to the above that brute force searches can be done in parallel or in sequence, and describe the database used for agent registration as a predefined directory server used to register, unregister or locate agents. Other agents use this directory to find the agent. Since communicating agents need to agree in advance on the naming server, an architecture where agent servers share a default naming server simplifies the registration system.

In SMART, the Finder module in the Region Administrator provides the naming service to the region administrators and the layers above. SMART does not allow the originating agent system to keep a remote reference to the agent once the agent is dispatched. On its return the agent registers itself with the originating agent system. Users can access the parked agent at a later stage with an agent-key, an identifier given to the agent by the user, that must be remembered by the user [19].

The DMF defined by Feridun and Krause [4] uses a distributed directory composed of the local directory subtrees of all DMNs, to share information. Each local directory tree holds the location of the current distributed directory node, definitions of the management tools supported by the DMN and management components installed at the DMN, with their state information. The directory supports a generic search mechanism to locate nodes and also provides an event notification service.
communication (RMI, RPC or CORBA-IOP) and indirect communication (event notification and shared group/meeting objects or global shared tuple-spaces) [17]. The classification by Horvat et al [8] correlates: procedure call mechanisms correspond to connection-less communication, while callback mechanisms correspond to connection-oriented communication and the mailbox mechanism to indirect communication. Gray et al [5] mention four approaches to communication: passing messages containing strings or arbitrary data, passing messages containing serialized objects, invoking methods in the other agent and publishing events to some sort of channel. The first two approaches correspond to connection-oriented communication, the third to connection-less communication and the last to indirect communication.

For both connection-oriented and connection-less communication, agents need to know each other’s names to establish communication. In connection-oriented communication, the session will be disrupted if a participating agent migrates, while in connection-less communication, if the agent is transferred, an exception will be thrown, and the initiating agent can re-execute the lookup and binding protocol to re-establish communication with the other agent at its new location. Mutual authentication (how to identify the other party) is difficult to support, since a challenge-response based protocol is difficult to implement as agents cannot carry their private keys when executing in foreign environments [17].

Most mobile agent systems use more than one of these mechanisms, as well as broadcasting/multicasting when sending the same message to multiple receivers [8]. D’Agents, for example, use message passing (connection-oriented communication), streams (connection-less communication) and events (indirect communication). A simple directory service, implemented as a collection of stationary cooperating agents, supplements the other mechanisms to support information retrieval applications [5]. Telescript agents ‘meet’ other local agents (the meet primitive belongs to the default interface of agents and can be invoked by other agents to establish a meeting), and can then invoke methods on objects in the other agents (connection-less communication) [5, 14]. Telescript agents can also communicate by sending events [5].

The DMF developed by Feridun and Krause [4], described on under section 3.1 and elaborated on below, as well as the Food Web agent evolution computing model described by Shih [15], described below, are two more examples of mobile agent systems using more than one mechanism to communicate.

In the DMF developed by Feridun and Krause [4], the communication service provide seamless remote and local communication between management components (agents). Communication includes messaging (connection-oriented communication), remote method invocation (connection-less communication), event forwarding and exception forwarding (indirect communication). The distributed directory in the DMF is used as a means to allow components to share information. It provides an event-notification service and supports a generic search mechanism for locating nodes. The DMF is implemented in Java and based on the Objectspace VoyagerTM platform, which provides seamless remote communication between objects through its object request broker (ORB). The DMF extends the distributed hierarchical directory structure provided by Voyager with additional search facilities, an ActiveNode, a special kind of node registered in the directory to act as an interface to management components in the DMF, and an event notification mechanism to inform subscribers about changes in the directory.

In the Food Web agent evolution computing model (Agent Communication Network (ACN)) described by Shih [15], agent evolution computing is performed via actions in response to agent messages. The ACN is implemented in JATLite. Messages can be passed between agents or from a host station to an agent. A docking agent (an agent daemon) is run on each host station to clone the agent when a copy of the agent is requested via a clone message. Docking agents also send host station messages to search agents. Message routers (provided as part of JATLite) allow agent name registration, maintain agent message queues and send/receive agent messages (connection-oriented communication). A message router also holds a repository with useful information such as network states (indirect communication). JATLite uses Knowledge, Query and Manipulation Language (KQML) as the agent communication language. JATLite consists of five layers: the lowest layer is the abstract layer which uses TCP/IP; the base layer handles basic communication; the KQML layer parses and stores KQML messages; the router layer send/receive agent messages and register agent names; and the protocol layer support some standard Internet services such as FTP, SMTP, HTTP and POP3.
Yaridor and Oshima’s [2] description of using HTML as a user interface to deliver messages directly to remote agents in Aglets, is an example of communicating with a moving agent, as, by using naming services, the messages can be forwarded to agents who have already migrated. This is implemented with HTTP messages: HTTP requests are used to deliver messages directly to remote aglets. The HTTP request uses a URL that includes the location of the receiver aglet (the host part of the URL), its identity, the message type and parameters. Upon receiving the HTTP request, the aglet server locates the the receiver aglet and delivers the message directly to it. The aglet’s reply is returned by the corresponding HTTP response.

Maintaining security while providing remote communication to visiting agents, is an important factor. A visiting agent could copy or transfer unauthorized information, gain access to protected resources, or launch denial of service attacks. An agent also needs to be protected from other malicious agents. Therefore, encrypted and authenticated inter-agent communication needs to be supported [17]. Authentication is discussed in more detail in section 3.2.8.

### 3.2.4 Access to Local Resources

Access control policies are used to decide which resources an agent can access, based on its own or its owner’s credentials. Access control policies should define access control for application-defined resources as well as for system level resources such as files, disk storage, I/O devices and network ports [5, 17]. Ajanta [17] and D’Agents [5] both base access control on the owner of the agent, while some systems, such as Aglets, take authorship of the agent also in consideration in determining the access policy for an agent [17]. D’Agents uses resource-manager agents to maintain access control lists and enforce the policy [5]. The DMF developed by Feridun and Krause [4], provides access control by means of the security service, which is part of the core services for each distributed management node (DMN). Also installed on each DMN, is a service manager responsible for loading service interfaces that enable access to local operating system tools or third-party software packages available on the host.

Resource consumption, such as CPU time, disk storage, number of threads, number of windows, network bandwidth and the like, should be limited too [5, 17]. Resource control in Java-based mobile agent systems may pose a problem, as a Java thread, the way a mobile agent is typically implemented, can run indefinitely and consume all the resources in the associated JVM [4, 14]. The DMF developed by Feridun and Krause [4], relies on the ‘cooperative nature’ of management components (MCs) to stop a thread: to terminate a MC, a flag is set to indicate to other threads that they should stop executing. The Nomads system currently offers the most control over the use of CPU cycles, network bandwidth and disk space by means of a custom-designed JVM [5, 17].

### 3.2.5 Fault-tolerance/Persistence

Picco [14] points out that, while improved fault-tolerance is cited as one of the benefits of mobile agents, this is only true if agent migration itself is fault-tolerant with proper mechanisms for local recovery in place.

Various situations, such as breakdown of connections or hosts, destruction of the agent or network errors causing the agent to get lost, can prevent an agent from migrating successfully [8, 18]. While most systems offer little support for failure detection and recovery [14, 17], a number of systems provide some form of persistence for mobile agents by means of a checkpoint-restore mechanism to restart agents [5, 8, 14]. The agent’s state information is checkpointed before and after execution on a host server, and when the server is restarted, a recovery process restarts any agents left on the server at last shutdown [8]. Tacoma achieves persistence by allowing agents to create folders inside cabinets (storage space on disk); on boot-up, Tacoma examines the ‘system’ cabinet and launches new agents for all the folders detected there [5]. The DMF developed by Feridun and Krause [4] offers a persistence service that allows distributed management nodes (DMN) components to be stored in persistent storage and reloaded to recover from a system crash. The DMF is based on the Objectspace Voyager platform that provides persistence through its object request broker (ORB). Neither D’Agents [5] nor SMART [19], on the other hand, offer any persistence, while SMART relies on Java’s exception handling system to deal
and users of agents must register with the trust service, and must possess a certified public key. A user must also inform his bank and obtain a virtual account number (VAN). The trust service allocates a unique ID to each new agent registered. This ID and the VAN will allow the agent to execute e-commerce transactions. Special traders for agents will supply agent hosts to agents. DTP and encryption in combination with the trust service creates a reliable protocol for agent transfer, while the recovery and rollback mechanisms of the DTP, in conjunction with additional checkpoints, allow recovery if a host crashes. Once transferred, the agent can execute business transactions for its owner. An agent that has completed its work, either terminates at the current host or returns to its owner. This is logged by the trust service and invalidates the agent. The trust service can also terminate the agent by itself or on behalf of its owner.

3.2.6 Interoperability

Mobile Agent System Interoperability Facility (MASIF, originally MAF) was developed by the Object Management Group (OMG) to address the interoperability of different mobile agent platforms written in the same language (Java) and to support integration with legacy applications by means of CORBA. MASIF specifies two interfaces: MAFAgentsystem and MAFFinder. Agent management and agent transfer are standardised through the MAFAgentsystem interface, while MAFFinder facilitates naming services and tracking agents by standardising agent and agent system names, as well as agent system type and location syntax. Agent communication is addressed by CORBA [13, 17].

The SMART (Scalable, Mobile and Reliable Technology) mobile agent facility developed by Wong et al [19], is an example of a mobile agent system based on MAF (MASIF) specifications. SMART incorporate dynamic aggregation, that are not yet part of the MAF specifications, but can build mobile agent applications to run in MAF compliant environments [19].

FIPA (Foundation for Intelligent Physical Agents) also developed specifications for external behaviour of agents and interoperability with other agents, non-agent software, humans and the physical world. FIPA’s model includes a Directory Facilitator, an Agent Management System and an Agent Communication Channel. These are specific types of agents residing on every agent platform to support agent management [17].

Kotz and Gray [10] point out that, since MASIF addresses only cross-system communication and administration, a mobile agent may migrate only to machines running the ‘right’ agent system instead of to other desired machines. They advise that the mobile agent community standardise on some specific execution environment (e.g. a particular virtual machine), as well as on the format in which the code and state of migrating agents are encoded.

3.2.7 Agent Management and Control

The MAFAgentsystem interface in MASIF standardises controlling agents of another agent system by specifying interfaces for creating, suspending, resuming and terminating agents [13]. This is also one of the functions of the Agent API, which represents software that allows users to create and control agents locally or directly at remote hosts.

According to Aridor and Oshima [2], it should be possible to control agents from thin-client environments such as handheld devices or web-browsers, and to incorporate support for mobile agents in familiar user interfaces, so that users need not be restricted to proprietary software or have to master new user interfaces to use mobile agents. To this end they describe a desktop-like tool for agent management and an HTML-based user interface for mobile agents. They propose a desktop aglet viewer in which aglets and locations are represented as icons, to act as Agent API. By dragging and dropping icons of aglets onto icons of locations or the Thrash icon, clients can dispatch or dispose local agents. A description of the HTTP messaging used to implement HTML as a user interface to agents is available under the section 2.3.3.

Transactions guarantee the consistency of data records when multiple users/processes use them simultaneously, due to the
3.2.8 Security

Though security is perceived as one of the main issues in preventing a general acceptance and use of mobile agents [12], and a significant amount of research is done in this area, there is also claims that in some situations or environments (e.g. in Intranets), the importance of this issue is overestimated [10, 11, 12, 14].

Nevertheless, the two most important problems in this area are mutual protection for the host and the mobile agent against each other. The host needs to be protected against attacks by unauthorised or malicious mobile agent code, while an agent needs to be protected from the host that is executing it [11, 14, 17].

Attacks on the host may take the form of unauthorised access or modifications through the agent system to data and the resources available at the server, denial of service, spoofing and the like [13, 17]. Most of these problems can be addressed adequately [10, 14], with the possible exception of resource control for Java-based systems, as a Java thread (the way in which an agent is usually implemented) can run indefinitely and consume all the resources available to its JVM [14]. See section 3.2.4 for a description of the methods employed by Feridun and Krause’s DMF [4] and the Nomads system [5, 17] to prevent an agent implemented in Java of consuming all available resources.

Security policies and agent authentication are used to protect the host against agents. Agents are authenticated to verify their ownership, as they access a host on behalf of their owner. This is done by authenticators, algorithms that determine an agent’s authenticity by examining the agent’s credentials. The agent’s credentials identify its owner (the authority it has been sent by) as well as the rights assigned to it by its owner, and, in some mobile agent systems, the programmer of the agent too. Security policies with rules for restricting or granting agent capabilities and access, as well as setting limits to agent resource consumption, are based on agent credentials [5, 13, 17].

Protecting an agent against a malicious host, pose more of a problem, since the host needs access to the agent code in order to execute it. Initially this was seen as impossible, but some researchers have shown that tampering can be detected or prevented [14]. Agents, as well as the information they collected, need to be protected from unauthorized access or modifications by malicious intermediate servers during migration. Secure communication channels and message authentication codes or encryption are usually used to protect the agent state during transmission [5, 17], but this does not guarantee the integrity of the data collected by the agent, or protect the agent’s credentials against theft or tampering [17].

Securing the naming services is indispensable in providing a secure mobile agent platform. The entries in the name registry need to be protected against unauthorized modifications and the namespaces assigned to principals or authorities have to be protected so that a user cannot create names in other users’ namespaces without authorisation [17].

Many security-related aspects and examples have been discussed in more detail in the previous sections:

- Section 3.2.2 elaborates on namespaces and name services, and lists some examples.
- Section 3.2.3 points out the importance of providing security during remote communication.
- In Section 3.2.4 access control policies and resource consumption are discussed, with some examples.
- Section 3.2.5 offers a description of the DTP-based enhancement for host and agent security built by Vogler et al [18].
- Section 3.2.7 lists two examples of remote control of agents through secure mechanisms (Ajanta [17] and the DTP enhancement [18]).

Further elaboration on the above examples of how existing mobile agent systems handle security, follows:

- The DMF developed by Feridun and Krause [4], uses a security service as part of the core/essential services for each distributed management node (DMN) to provide secure communication and transport between DMNs and to protect different parts of the DMN by means of access control. The DMF is developed on the Objectspace Voyager TL platform, which includes a customized security manager to protect local system resources. Feridun and Krause...
3.3 Constructing the Mobile Agent Itself

This section investigates the process of constructing a mobile agent, more closely by examining an agent’s lifecycle, agent design patterns, the computational model and programming primitives used to construct a mobile agent, as well as languages supporting mobile agent programming.

3.3.1 An Agent’s Life Cycle

Horvat et al [8] describe an agent’s life cycle as follows: A new agent is *created* only once, when it receives a unique ID and initial state, and is prepared for further instructions. Each time the agent arrives at a new host, it is *started* with its own thread of execution. The server initializes the agent, which can execute asynchronously. On *deactivation*, the agent stops processing, and stores its state and intermediate results to disk through object serialization, which allows exporting it to a byte stream, and reconstructing it from the byte stream on another host after *migration*. On *disposal*, the agent stops all its operations and frees all resources it has been using, causing its state to be lost permanently. When more than one agent is needed to complete a task, multiple copies can be *cloned* through object serialization.

The description by Wong et al [19] of how a mobile agent is implemented in SMART, corresponds largely to Horvat et al [8]. SMART implements a mobile agent as a Java thread, as this enables place servers and agent systems to *start*, *suspend*, *resume* and *stop* an agent easily. The agent is executed as a new thread on each host, and suspended, serialized and *transmitted* as a series of bytes. On arrival at the new host, it is deserialized and started as a new thread, similar to the original agent.

In the DTP-based enhancement of existing software for host and agent security built by Vogler et al [18] (see section 3.2.5), the third-party trust service generates a unique ID for the new agent, confirming Horvat et al’s [8] description. The agent searches for a new target host with the assistance of special traders for agents. A copy of the agent is *transferred* using distributed transaction processing and encryption. On confirmation that the agent has been transferred successfully, the old agent is deleted. Once the agent has accomplished its task, it can return to its owner or *terminate* at the current host. In either case an appropriate message has to be sent to the trust service, where this information is logged, turning the agent invalid. An agent can be deleted on behalf of its owner or the trust service, by means of the trust service sending a message to the current host, which deletes the agent and renders it invalid.

Shih [15] describes a theoretical computational model for mobile agent evolution on the Internet, called Agent Communication Network (ACN). Mobile agents live on the Internet, where they migrate or move around automatically or semi-automatically via some communication paths. While agents with the same goal can cooperate by sharing information, they also compete for resources such as bandwidth or disk space. This situation corresponds to the balance in the ecosystem in the natural world (the food web or food chain), and is described in the ACN. Agent evolution computing consists of simultaneously searching for a goal and cloning agents. If a system resource meets a basic requirement, an agent is *activated* to search in the local station. If the goal is achieved, all agents searching for the same goal is set dangling and the query result is send back to the original query station. If the goal cannot be achieved, the agent is *cloned* in another station. If, however, the system resource is below the basic requirement, the agent is either *suspended* or *killed*. During the evolution process, agents in the ACN have different internal states: *searching* for a goal, *suspending* while waiting for enough resources to resume searching, *dangling* while waiting for a new goal, *mutating* when changing to a new species with a new goal and possibly new host station [15].

Both the DTP-based enhancement of existing software for host and agent security built by Vogler et al [18] and Shih’s [15] Agent Communication Network (ACN) confirm that even though the mobile agents act automatically, there is some element of control over them. In the DTP-based enhancement, an agent can be deleted on behalf of its owner or the trust service, while in the ACN, an agent will be killed or suspended if system resources falls below the basic requirement.
Each phase. The higher phases is agent-platform independent and can easily be reused. Three levels are distinguished:

- System Environment Analysis Phase - In this phase the developers identify the existing infrastructure (e.g. physical networks and databases) that is available in the agent system environment. This simplifies the specification of the agent functionalities needed and identify what additional infrastructure should be developed.

- Macroarchitecture Decision Phase - In this phase the macroarchitectures representing the global structure of the system and the agent behaviours that will satisfy the system requirements by using the existing infrastructure, is identified. The macroarchitectures are general purpose and agent-platform-independent. As it is generally assumed that agent systems offer mobility, cooperativity and copying facilities, the mobility, cooperation and mobility/cooperation combination patterns are used in this phase.

- Microarchitecture Decision Phase - In this phase more detailed agent behaviours than in the previous phase is specified, with the intention of utilising the facilities offered by the agent platform to support the macroarchitectures and the behaviour expected at each network node. This phase is therefore dependent on the agent platform, and typically use basic action, execution continuation, plan generation/execution and security/safety patterns.

Kendall et al [9] present another view of a mobile agent system in their application framework for intelligent and mobile agents called JAFIMA1, in the form of a layered agent architecture. The framework is documented in patterns. They argue that an agent should be decomposed in layers, as higher level behaviour depend on lower layer abilities, levels only depend on their neighbours and information flow both ways between neighbouring levels. Seven layers are presented in the framework, with mobility as the seventh layer the highest one and the sensory layer the lowest one:

- Sensory - This layer senses changes in the environment and updates beliefs. It uses the Adapter pattern to cooperate with domain specific interface classes.

- Beliefs - The agent’s beliefs is stored here and whenever a belief is updated through sensory input or agent actions, the next layer, the Reasoning layer must be informed. The Composite pattern is used to treat beliefs uniformly and the Observer pattern is used to notify the Reasoning layer when a belief is updated.

- Reasoning - Beliefs and requests are interpreted to determine an appropriate action. This layer uses the Interpreter and Strategy patterns.

- Actions - The plan selected by the Reasoning layer is executed in this layer. It uses the Command pattern to turn a plan into a command, the Abstract Factory pattern to create a plan object, the Factory Method to create intention threads dynamically, the Decorator to implement a prioritizer for scheduling and prioritizing actions; and the Future and Observer patterns for asynchronous method invocation.

- Collaboration - This layer is responsible for cooperation and for the exchange of services with other agents. The Collaboration layer sends, receives and rejects requests, as well as replies to messages. It uses the Synchronized Singleton pattern to manage collaboration threads, Decorator to change the behaviour of the thread dynamically, Active Object to schedule requests from the Actions layer, Future and Observer for asynchronous method invocation and the Strategy pattern to convert a message into the language of the destination agent.

- Translation - Translates incoming and outgoing messages to the appropriate semantics.

- Mobility - This layer supports virtual migration by providing location transparency between agents and it supports actual migration by providing a structure for cloning an agent in another environment. It uses the Factory Method, Visitor, Proxy and Active Object patterns.

Yaridor and Lange [1] present a set of agent design patterns for creating mobile agent applications. They argue that agent design patterns capture solutions to common problems in agent design and expect that it can fill the void between very high-level agent-specific languages and system-level programming languages like Java. It can also serve as a base for visual agent development environments. Three classes of patterns are distinguished: travelling, task and interaction.

The Travelling patterns handle the movements of mobile agents and consist of:

- The Itinerary pattern objectifies agents’ itineraries (a list of destinations, routing schemes, instructions for special cases e.g. when a destination does not exist), allowing it to be saved and reused.

- The Forwarding pattern allows a given host to mechanically forward all or specific agents to another host.
The Interaction Patterns deal with locating agents and enabling their interactions.
• The Meeting pattern provides a way for two or more agents to arrange local interaction at a given host.
• The Locker pattern allows agents to temporarily store data in private while it visits another destination.
• The Messenger pattern uses a messenger agent to transfer messages between agents.
• The Facilitator pattern defines an agent that provides a naming and locating service for agents.
• The Organized Group pattern gathers multiple agents within a group in which all members of the group migrate together, a fundamental way of collaboration among multiple mobile agents.

The Facilitator pattern listed under the Interaction patterns corresponds to the naming services discussed in section 3.2.2, while the Meeting, Locker and Messenger patterns (also listed under the Interaction patterns) correspond to the design choices in facilitating communication discussed under section 3.2.3.

Though, according to their bibliography, it does not seem as if Gschwind et al [6] has used the patterns identified by Yaridor and Lange [1] as a base for a visual agent development environment, the components in their AgentBean Development Kit (ADK) mirror the classes of patterns identified by Yaridor and Lange. Gschwind et al [6] propose a framework in which a mobile agent is composed of a set of categories of components: navigators, performers and reporters. Navigators determine and manage the agent’s itinerary, corresponding to the Travelling patterns identified by Yaridor and Lange. Performers execute management tasks at hosts, similar to what is expected of Task patterns. Reporters manage the delivery of the agent’s results to the designated destinations, which is covered by Interaction patterns. Using components allow a modular definition, assists agent creation and encourages reuse. Interaction between components is based on an event/action-based communication, as an event generated by one component may trigger an action by another component. Agents are created by selecting components from a list in a visual design tool and specifying their interaction. The tool generates the source code for the agent. The visual design tool is based on Sun’s Java Beanbox, but adds support for the construction of mobile agents.

In [11] Picco presents a different view on mobile agent components. He proposes structuring mobile agent systems as a set of mobility components in which mobile code and mobile agents interoperate with existing mechanisms, such as remote method invocation, wherever possible. The current features in mobile agent systems would also be available as a set of orthogonal components that could be plugged in by programmers to provide for example mobility, security or communication. This would provide much more flexibility than current mobile agent systems allow, but it could be a quite difficult challenge to convert mobile agent system features into standard, reusable, orthogonal components to be combined as needed.

Ajanta is an example of a system in which patterns are applied: Ajanta uses composable migration patterns to program an itinerary allowing a complex travel plan to be constructed from some basic patterns [17].

The Actions layer in JAFIMA [9] and the Task patterns presented by Yaridor and Lange [1] both fit in under Tahara et al’s [16] Microarchitecture Decision Phase. JAFIMA’s Collaboration and Mobility layers respectively correspond to the Interaction patterns and Travelling patterns presented by Yaridor and Lange [1]. These will all fall under Tahara et al’s [16] Macroarchitecture Decision Phase. Though none of Yaridor and Lange [1], Kendall et al [9] or Tahara et al [16] use the same names to describe the patterns in their classifications, using agent design patterns certainly merits further investigation.

3.3.3 Computational Model and Programming Primitives

Many mobile agent systems use a simple event driven model, with handlers for different types of events such as migration, dispatch and arrival at a server [17]. In some systems, such as for example Aglets and Grasshopper, the same method is always invoked as entry point, and the state captured at the previous host determines the actions to be executed. Other mobile agent systems (Ajanta, Concordia, Voyager, Jumping Beans) allow any public method as entry point [5, 17].
to slave (dispatching the agent, initiating task execution, handling exceptions occurring during task execution). Master and slave agents are defined as subclasses of Master and Slave, which implements the varying parts such as what task to perform and how to handle the task’s results. This allows the reuse of code and simplifies agent design and implementation as developers only need to implement the variable parts of already defined agents (available from libraries or agent builder tools).

Odyssey, a Java-based mobile agent system developed by General Magic, provides two kinds of agents: ‘real’ agents and workers, but workers do not seem to be child agents. A worker is constructed of a set of tasks, each to be executed at specific hosts. On completion of the tasks at a specific host, the worker migrates to the following task and host on its list. The worker can change its task list at any time during its migration by adding more destinations. A ‘real’ Odyssey agent has more control in executing its tasks, can move during execution and is not bound to the system where it was created [8].

In the theoretical computational model for mobile agent evolution on the Internet, called Agent Communication Network (ACN), described by Shih [15] (see sections 3.2.3 and 3.3.1), three strategies for cloning an agent, are examined: brute force agent distribution, semi-brute force agent distribution and selective agent distribution. Brute force agent distribution simply clones an agent on all remote stations that contains information that can help the agent achieve its goal. This creates multiple copies of agents with the same goal and priority in the ACN, increasing network traffic and agent societies’ sizes. Despite its lower complexity, it generally delivers a bad performance. Semi-brute force agent distribution does not make a copy of the agent, but gives the goal to an agent on all stations that contains information that can help achieve the goal. Agent societies increase in size due to different users launching the same goal with different priorities. Agents from the same user will not increase the size of the agent societies, however. Selective agent distribution will not clone an agent if another agent at the destination station shares the same goal. If there is no agent sharing the goal, an agent on the destination station is asked to search for this additional goal. Only if no agent exists on the destination station, will a new agent be cloned, so that agent societies’ sizes increase due to demand. Its higher complexity notwithstanding, this reduces the number of agents and network traffic, and offers the best performance. A docking agent, an agent daemon run on the host station, assists with copying the clone to the host station and activating the cloned agent. The cloning process itself commences when the agent that needs to be cloned, sends a query to a commercial search engine. The commercial search engine responds with a set of URLs, which the agent uses to make a copy of itself by sending a clone message to the docking station on the remote host. The docking station activates the clone to start executing [15].

Of the mobile agent systems covered by the articles perused for this literature survey, only Ajanta [17] and the ACN [15] refers to cloning, confirming that to date very few mobile agent systems allow child agents.

3.3.4 Languages Supporting Mobile Agent Programming

Early mobile agent systems, with the primary focus of demonstrating the agent paradigm, used scripting languages such as Telescript (developed by General Magic in the early 1990s), Perl and Tcl [17]. Telescript supplements systems programming languages like C and C++. In a typical application the stationary software (e.g. software to allow agents to interact with users, and places to interact with databases) is written in C and the mobile agents as well as the interfaces of places to which they are exposed, are written in Telescript [8]. Languages like Tcl, Scheme, Oblique and Rosette were the result of attempts to improve distributed programming and to enable mobile agent programming. Even C and C++ were used in these attempts [8].

Today most mobile agent systems use Java [5, 8], listing the following reasons therefore:

- **Object-orientation** - Java’s Remote Method Invocation allows object methods to be called over the network [4, 5, 8, 17].
- **Code mobility** - The serialization of objects allows objects to be converted to byte streams for network transmission [5, 8, 17].
- **Dynamic code download** - Source code for software can be downloaded transparently from anywhere on the Internet [4, 8].
a double-edged sword for the field of mobile agents, since it provides many features that simplify the programming of mobile agents, while this very fact often bias designers towards adapting their design to be implemented in Java. Mobile agent systems built on the Java Virtual Machine (JVM), for instance, usually support only weak mobility [8, 14, 17], unless the JVM is modified [14, 17], as is the case in D’Agents [5]. Java provides the mechanisms to implement weak mobility (the ability to program the class loader), but cannot handle the execution state [14]. Tripathi et al [17], on the other hand, claim that it is generally felt that program-controlled migration under weak mobility suffices for most applications. Another example is the way the Java class loader is used within Web browsers to support applet downloading - only the agent’s root class is transferred with the agent, additional classes needed for execution of the agent have to be downloaded from the agent source host or some other code archive. This assumes that the code archive will always be available, and so denies one of the most important advantages of mobile agents, namely the ability to support disconnected operations [14]. Java-based mobile agent systems relying on serialization allows the full object closure of an agent to be copied, while enabling the programmer to tag the fields that should not be serialized. In this way data management upon migration such as for example binding to resources, has to be handled entirely by the programmer. Some exceptions, such as Fargo and 
\mu code, do exist, but an additional model or package is implemented in each case, while Telescript has a built-in mechanism controlling migration and security through a single notion of ownership [14].

Ideally a mobile agent system should support several languages to accommodate different application needs [5, 11]. Ara, Bond, D’Agents and Tacoma support multiple languages. D’Agents implements an execution environment for each of the three supported languages (Tcl, Java and Scheme). Each Tcl and Scheme agent is executed within its own Unix process, while multiple Java agents are executed as threads inside a single Java process [5].

Most mobile agent systems either interpret their languages directly or compile their languages into bytecodes before interpreting it. This presents both portability and security advantages [5, 7]. Interpreted code supports heterogeneity, and the late binding offered by interpreted code enables a mobile agent to contain references to functions or classes not present on the system from where it is despatched, but which are available at the destination. If the agent is built from object classes and in an interpreted language, with identical classes available at the destination, it can be reduced to object references, instance data and process state data. For agents in an interpreted language, state data is captured on the stack and registers need not be saved. It is also easier to secure interpreted languages, since the language developer explicitly controls what system resources are accessible [7].

Most of the mobile agent systems covered by the articles examined for this literature survey use Java, confirming its popularity as a mobile agent programming language. Ajanta [17] and SMART [19] have been programmed in Java. Feridun and Krause’s DMF [4] is based on Voyager, which is implemented in Java. The agent construction toolkit developed by Gschwind et al [6], AgentBean Development Kit (ADK) uses Java Beans. The Agent Communication Network (ACN) described by Shih [15] uses JATLite, a software library developed at Stanford University for building software agents that communicate over the Internet. JATLite is developed in Java and uses Knowledge, Query and Manipulation Language (KQML) as the agent communication language. D’Agents supports three languages, one of which is Java [5].

3.3.4.1 Implementation in Java

A brief summary describing the typical implementation of a mobile agent in Java, based on its life cycle, follows.

A mobile agent is implemented as a Java thread [8, 13, 19]. To create an agent, the mobile agent system creates an instance of the Agent class within a default place or a place indicated by the client (creation). The Agent class specifies the interface and implementation of the agent, and executes on its own thread, initialized by the server (started) [13]. Mobility is achieved through object serialization, capturing the data and state of the agent as a sequence of bytes to disk (deactivation), which can be transmitted over the network through RMI and deserialized at the destination (migration) [8, 19]. The destination place server calls the run() method to start a new thread [19]. The thread can be stopped by
3.4 Tools Available

Horvat et al [8] describe four commercially-available Java-based mobile agent systems: IBM’s Aglets, Object Space’s Voyager, General Magic’s Odyssey and Mitsubishi Electric ITCA’s Concordia. These mobile agent toolkits provide all the Java classes necessary to create a mobile agent - the programmer only need to provide the algorithms to achieve the objectives. They exhibit some common characteristics: All provide an agent server, agents and their state can migrate from server to server, agent code can be loaded from various sources (file systems, the Web and ftp servers) and they are all implemented purely in Java.

Aglets Software Development Kit (ASDK) has been developed to add mobility to applets and to build a network of Aglets hosts. It consists of the Aglet API, called Aglets Runtime Layer, documentation, the Aglets Server named Tahiti and the Agent Web Launcher known as Fiji. Tahiti provides a GUI to monitor, create, dispatch and dispose of agents, and to set the agent’s access privileges to the agent servers. Fiji is a Java applet that can create aglets or retract an existing aglet into the client’s web browser, using the agent’s URL as parameter [8].

Aglets migrate through sockets, use weak migration, and assume that all system classes are available at the destination, which means it can only transfer to systems running the Aglets Server. Synchronous (connection-less) and asynchronous (connection-oriented) communication are supported locally, while Message Aglets is send to remote objects, though messages cannot be send to moving aglets. Indirect communication is supported through a subscription model for receiving specific types of messages [8].

The latest versions of the Aglet system are ASDK Version 1.0.3 and ASDK 1.1 Beta, and they can be downloaded from the IBM website [8]. Yaridor and Oshima [2] propose an interactive desktop Aglet viewer to allow clients to dispatch or dispose local agents by dragging and dropping icons of aglets onto icons of locations or the Thrash icon.

In Voyager all serializable objects can become mobile by using the Virtual Code Compiler to create a remote-enable “virtual class”. Voyager uses weak migration and RMI for transfer. Communication with moving agents is enabled through the forwarder address left by each agent when it moves. Four types of messages is supported: synchronous (connection-less), one-way (connection-oriented), future, one-way multicast and selective multicast (indirect communication). Voyager provides persistence with an explicit SaveNow() method that saves a copy of the agent to the Voyager database. When the agent migrates, the copy moves with it to the remote database, leaving a forwarding address in the local database. Voyager agents can only execute restricted operations on the host, since it is not necessary to run the Voyager server on all nodes in the network. Five different life-span schemes are available for agents: until there are no references to it (default), for a specific time limit, until an explicit point in time, until it has been inactive for a specific time-limit, and forever [8].

Voyager Core Technology Version 2 is free for most commercial applications and can be downloaded from the Object Space website [8]. Wong et al [19] used Voyager to implement SMART, with the addition of dynamic aggregation. Feridun and Krause’s [4] DMF is also based on Voyager, but extends it capabilities to include additional search facilities and an event notification mechanism.

Odyssey provides a set of Java class libraries for developing distributed mobile applications, and includes agents, an agent system and places. There are two kinds of agents: ‘real’ agents and workers. A worker is constructed of a set of tasks, each to be executed at specific hosts. On completion of the tasks at a specific host, the worker migrates to the following task and host on its list. The worker can change its task list at any time during its migration by adding more destinations, while a ‘real’ Odyssey agent has more control in executing its tasks, can move during execution and is not bound to the system where it was created. Odyssey offers three interfaces: AgentSystem, Finder and Transport which allow the user to customize an Odyssey agent system. It uses weak migration and RMI. No security mechanisms, except that provided by
provides the interface between Concordia agents and local resources. The Agent Tools Library contains all the classes required to build Concordia mobile agents [8].

Concordia uses RMI to transfer agents and the agent’s itinerary to determine the next destination. Two types of inter-agent communication is possible: distributed asynchronous events (indirect communication) and collaboration for complex agent coordination, though no direct agent to agent communication is possible. MEITCA provides a free 30-day evaluation kit (without the Security Manager) of the Concordia Version 1.1 on their website [8].

The Ajanta mobile agent system used by Tripathi et al [17] is available for educational and non-commercial use from http://www.cs.umn.edu/Ajanta/new.html. Ajanta has been programmed in Java and uses weak mobility. Agent classes are downloaded from a code-base server on demand during execution. Ajanta allows multi-threaded agents, but it remains the programmer’s responsibility to ensure that all threads have terminated or reached a state safe to terminate and transfer the agent. Itineraries control agent migration and any public method can be used as entry point on arrival at the host. Child agents can be created to execute tasks in parallel. Agents communicate via RMI (connection-less communication). Access control policies for an agent’s access to files and network resources, are based solely on the agent’s owner, and a proxy based resource access mechanism is used for access to local resources. The integrity of information collected by an agent is protected by using a secure append-only container. A guardian is used to assist in error recovery (see section 3.2.5).

The D’Agents (formerly known as Agent Tcl) mobile-agent system, developed to support distributed information retrieval and to characterize the mobile agent performance space, is publicly available. D’Agents supports three languages: Tcl, Scheme and Java, and provides an execution environment for each of the three languages and the agents themselves. D’Agents use strong mobility, and allows agents to create and use multiple threads, but only the calling thread can migrate. The entire code-base for the agent is transferred during migration. No persistence is provided. D’Agents use three low-level communication mechanisms: message passing (connection-oriented communication), streams (connection-less communication) and events (indirect communication). A simple directory service, implemented as a collection of stationary cooperating agents, supplements the other mechanisms to support information retrieval applications. D’Agents protects a host from malicious agents, and encrypts an agent in transit. Resource-manager agents maintain access control lists, which is based on the owner of the agent, and enforce the access policy for an agent [5].

D’Agents software and documentation for the Tcl and Scheme modules can be downloaded from the D’Agents website at http://agent.cs.dartmouth.edu/, and the authors can be contacted for the Java module [5].

Shih [15] developed an agent evolution computing model (Agent Communication Network (ACN)) in JATLite, a software library developed at Stanford University for building software agents which communicate over the Internet, but whether JATLite is publicly available is unknown. JATLite is developed in Java and uses Knowledge, Query and Manipulation Language (KQML) as the agent communication language.

Other mobile agent tools, to be used on top of existing mobile agent systems, and which may not necessarily be commercially available yet, include Vogler et al’s [18] DTP-based enhancement of existing software for host and agent security, and Gschwind et al’s [6] visual design tool to compose agents and specify their interaction.

The DTP-based enhancement can be used on top of existing mobile agent systems, enhances host and agent security and has additional features to allow agent management and control [18]. See section 3.2.5 for more information on the DTP-based enhancement.

Gschwind et al’s [6] visual design tool consists of an agent construction toolkit (Agent Bean Development Kit (ADK)) and a plug-in abstraction layer for existing mobile agent platforms to allow execution of mobile agents created with the ADK. This can be used to compose agents used to assist with network and systems management, from a set of categories of components: navigators, performers and reporters. Agents are created by selecting components from a list in a visual
Several areas in which mobile agents offer advantages, have been identified: systems and distributed management [3, 4, 6, 9, 12, 15, 19]; information retrieval, especially for unpredicted information-retrieval tasks [3, 5, 12, 15]; disconnected computing [8, 12, 15]; dynamic deployment of code [9, 12]; resource-limited devices [12, 15]; personal assistants [8, 12] and mobile agent-based parallel processing [8, 15]. The parameters influencing the circumstances under which the mobile agent are employed, however, should also be taken into account, as this affects scalability and performance. Network speed and reliability versus the size of the agent code and the volume of data to be collected, are some of the factors to be taken in to account [5].

No clear guidelines as to when mobile agents, rather than another paradigm, should be used, could be distilled from the articles covered. Picco [14], Kotz et al [11] and Johansen [12] all advise against indiscriminate use of mobile agents. Though Johansen warns in [12] that we should make sure that using mobile agents will benefit the design and its outcome, Picco [14] points out that no clear guidelines are provided on when to use which design paradigm for code mobility. Picco [14] indicates three design paradigms for code mobility: Code on Demand (COD), Remote Evaluation (REV) and Mobile Agents (MA). In the COD paradigm, the client lacks the know-how (the code) to perform the service, while owning the necessary resources. The code is then downloaded from a remote server and executed. In the REV paradigm, the client owns the know-how, but lacks the resources, which are owned by the server. In this case the client includes the code needed to perform the service, in its request to the server. In the MA paradigm, the client has the know-how, but lacks some of the resources. Here the client autonomously moves to the server to use local resources to perform the service.

Kotz et al’s [11] description of the various forms of mobile code corresponds, and elaborates by indicating that mobile agents with weak mobility and mobile agents with strong mobility also offer different forms of mobility. In a system with strong mobility, migration is completely transparent to the migrated program and all data is transferred with the mobile agent. Strong mobility reduces programmer-effort as well as the size of the transported code [14]. Tripathi et al [17], claim that program-controlled migration under weak mobility suffices for most applications, though this argument is not augmented with examples or a description of the type of application suited to weak mobility.

Whether all classes required by the agent code should be transported as part of the agent transfer protocol, or whether classes should be obtained on demand from a designated code-base server during execution, is another area where the purpose and circumstances under which the application will be used, have to be taken into account. Transferring all classes required by the agent code as part of the agent transfer protocol, supports disconnected operations, but makes agent transfer heavyweight [17]. Obtaining classes on demand, whether through dynamic aggregation or downloading from a code cache, can be used when lack of connectivity will not pose a problem as this will reduce bandwidth requirements and speed up the packing process before transferring an agent [19]. However, it has to be kept in mind that obtaining classes on demand slows down agent execution [17]. Optimizing agent transfer, taking the application circumstances into account, is recommended.

The name of an agent should not change during migration, as this simplifies programming since current locations of entities need not be taken into account. A location-independent naming scheme will achieve this [2, 13, 17], and an architecture where agent servers share a default naming server will simplify the registration system [2]. Using multiple locating schemes (e.g. brute force, logging, agent registration or agent advertisement) depending on security and performance requirements will also enhance mobile agent applications [2].

According to Wong et al [19], a well-designed mobile agent application will not need remote agent references and the user should be able to log out after the agent has been launched. In SMART, the originating agent system is not allowed to keep a remote reference to the agent once the agent is dispatched. On return, the mobile agent registers itself with the originating agent system, and users can access the parked agent at a later stage with an agent-key, an identifier given to the agent by the user, which must be remembered by the user [19]. Yaridor and Oshima [2] describe a facility to assist mobile agents unable to reach destination hosts, called an agent box, which is essentially a queue in which agents destined for hosts
Mechanisms to provide persistence for mobile agents and local recovery to ensure fault-tolerant agent migration should be in place, since, while improved fault-tolerance is cited as one of the benefits of mobile agents, it is only true if agent migration itself is fault-tolerant with proper mechanisms for local recovery in place [14].

Kotz and Gray [10] advise that the mobile agent community standardise on some specific execution environment (e.g. a particular virtual machine), as well as on the format in which the code and state of migrating agents are encoded, to ensure interoperability and promote a more widespread use of mobile agents.

Users should be able to create and control agents locally or directly at remote hosts through the API in the mobile agent infrastructure. According to Aridor and Oshima [2], it should also be possible to control agents from thin-client environments such as hand-held devices or web-browsers, and to incorporate support for mobile agents in familiar user interfaces, so that users need not be restricted to proprietary software or have to master new user interfaces to use mobile agents. To this end they describe a desktop-like tool for agent management and an HTML-based user interface for mobile agents.

It is essential to provide security in a mobile agent system. This consists of protecting the host through agent authentication and access control, which include controlling resource consumption, protecting the naming services, protecting the agent and the data it collected during transmission, as well as during execution, and providing secure remote communication.

Yaridor and Lange [1] argue that agent design patterns capture solutions to common problems in agent design and expect that it can fill the void between very high-level agent-specific languages and system-level programming languages like Java. It can also serve as a base for visual agent development environments. Though apparently not based on the patterns identified by Yaridor and Lange [1], the components in the AgentBean Development Kit developed by Gschwind et al [6] indeed reflect these patterns, and so allow a modular definition, assists agent creation and encourages reuse.

Though only a few mobile agents systems allow cloning or creating child agents to execute a set of subtasks in parallel, for instance for data-mining or information retrieval purposes, mechanisms to synchronize and coordinate activities as well as security considerations are important issues in this feature. Shih [15] describes three strategies for cloning an agent: brute force, semi-brute force and selective (see Section 3.3.3). Of these selective agent distribution delivers the best performance.

Ideally a mobile agent system should support several languages to accommodate different application needs [5, 11], especially since there is some doubt whether Java, the most commonly used language in mobile agent systems, is the most suitable [11, 14]. Most mobile agent systems use interpreted languages, as this offers portability and security advantages [5, 7]. A high-level mark-up language, like XML, can also be used to specify agent-based distributed applications, and can conceal many complexities of mobile agent programming from application programmers [17].

The guidelines given here, is an attempt to assist in creating/programing and using a mobile agent efficiently.

4. Conclusion

The variety in methods used to implement mobile agent systems confirm the problems pointed out by Kendall et al [9] due to the independent development of mobile agent systems, as discussed in Section 1.

Various areas in which mobile agents offer advantages, are pointed out, though no clear guidelines as to when a mobile agent rather than another paradigm, should be used, could be extracted.

A description of the infrastructure for a generic mobile agent system, as well as the features exhibited by mobile agent systems, such as mobility, the naming services, communication, access to local resources, fault tolerance/persistence,


