A programming language for service-oriented computing with mobile agents

Hervé Paulino¹, *, † and Luís Lopes²

¹Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Quinta da Torre, 2829-516 Caparica, Portugal
²Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto, Rua Campo Alegre, 1021/1055, 4169-007 Porto, Portugal

SUMMARY
We present MOB, a service-oriented scripting language, for programming mobile agents. We argue that combining the service-oriented and mobile agent paradigms provides a very natural model for programming a large class of distributed applications. In MOB, mobile agents in a network simultaneously provide and use services. The service interfaces constitute contracts that bind agents among themselves. The language features static type-checking to guarantee that contracts are respected at run-time. Other language features, such as redundant service providers, allow a certain degree of fault-tolerance by allowing applications to switch servers dynamically in case a failure is detected (e.g. a server crash). For these reasons, the target applications we envision more interesting to develop using MOB are services for networks with highly dynamic, volatile resources or simply highly adaptive, reconfigurable applications for more classical networks. The paper presents MOB and its implementation from an application programmer’s and a systems developer’s view. Copyright © 2007 John Wiley & Sons, Ltd.

INTRODUCTION AND MOTIVATION
Finding adequate paradigms for programming distributed applications is an open and particularly active research area. This situation stems not only from the exponential growth of the Internet but also, and perhaps mostly, from its changing form. In fact, current networks are quite flexible systems, allowing, for example, intermittent connection of resources, dynamic address attribution,
and dynamic binding to network resources. In this paper, we propose MOB, a language and a run-time system for programming with mobile agents under a service-oriented paradigm. We argue that combining the service-oriented and the mobile agent paradigms provides a very natural model for programming a large class of distributed applications.

When we introduced MOB [1], we were first motivated to produce a programming language for developing applications based on mobile agents. In [2,3], we extended MOB with another main abstraction, services, thus uniting two major paradigms for Web applications. Mobile agents in a MOB network simultaneously provide and use services and the service interfaces constitute contracts that bind agents among themselves. The language uses static type-checking to guarantee that these contracts are respected at run-time. Other language features, such as redundant service providers, allow a certain degree of fault-tolerance by allowing applications to switch servers dynamically in case a failure is detected (e.g. a server crash). For instance, using a service-oriented language, programmers are not required to keep track of agent names, just network-wide service identifiers. The same service may be provided transparently by several agents in the network, and in case of a server failure, the client may be re-directed to an alternative server without crashing the application. Note also that the implementations available for a given service may differ, provided that all implementing agents respect the interface contract. This allows a diversity of implementations for a service to co-exist, even with distinct mobility capabilities (e.g. a centralized vs mobile software installers). For this reason, we envision that a class of distributed applications may benefit from MOB’s programming model, namely, services developed for networks with highly dynamic, volatile resources or simply highly adaptive, reconfigurable applications running on more classical networks.

MOB delegates in the programmer the decision to migrate an agent to a new machine where local communication with a server may proceed. As a result of this option, MOB supports remote, peer-to-peer communication between agents. We feel that moving agents to impose local communication is detrimental in some network applications. So, we adhere to a less strict mobile agent paradigm for network programming. The user should, in this respect, have the last word.

Even classic client–server applications like web servers can benefit from the mobility features provided by MOB. Figure 1 illustrates the migration of a generic server that allows, for instance, to perform maintenance work on the original physical host, without losing service delivery. While it is true that this operation could also be performed by terminating the current server and launching a new instance of it in the new host, our approach is cleaner in that it ensures that the unavailability of the service is never announced to the user. In practice, the service will be unavailable during the migration, but that will be completely transparent to the user. The same statement cannot be made for the second approach, where at any given moment, either none or two instances of the server will be active. If no services are available, an error message will be caught by the user trying to access the service. If two instances are active, the original may receive a request just before its deactivation, causing the request never to be processed, since the new server will not receive it. With the use of mobility, the server resumes its execution in the new host in the exact same state on which it was suspended and, thus, no requests are lost.

From an implementation viewpoint, the front-end of the MOB compiler translates MOB code into DiTyCO source code with some extensions. DiTyCO [4] is a programming language based on a process-calculus with primitives for distributed computing and code mobility. This mapping allowed us to use the compiler developed for DiTyCO in previous work, with incremental changes, as the back-end of the MOB compiler. The run-time system was also adapted for the DiTyCO programming
language [5–7]. DiTyCO virtual machines are mapped one-to-one into MOB agents. A new layer was added to DiTyCO to handle the mobility of MOB agents between hosts in a MOB network. Reliable communication is ensured by a protocol that tracks agents and allows for the migration of endpoints.

MOB also features a run-time agent management infrastructure based on agent proxies and proxy connectors. Proxies are DiTyCO components that run attached to agents and filter interactions with remote clients, in particular, agent management operations. Proxy connectors are DiTyCO components that run on behalf of clients and enable them to establish a communication link and manage running agents in the network. Both proxies and proxy connectors are generated automatically, for each agent, by the MOB compiler.

To summarize, we feel that the main contributions of this work are the following:

- an expressive and dynamical model for programming service-oriented mobile agents;
- a concrete implementation of the programming model;
- an infrastructure for the run-time management of agents.

The remainder of the paper is structured as follows: the next section describes the syntax and semantics of the MOB programming language; section Programming with MOB presents two programming examples to demonstrate the expressiveness of MOB; sections The Compiler and The Run-time system describe the implementation of the language; section Related work compares our approach to existing work in the area; and, finally, we present some conclusions and directions for future work in section Conclusions and future work.

AN OVERVIEW OF MOB

The main programming abstractions in MOB are agents and services. Agents are objects with an associated run-time. Following an object-oriented approach, they are abstracted into classes defined by the agent construct, while instances are created with the new construct. Definitions for common objects are given by the class construct, and instances are created also with the new construct.

Agents may move through the network and this is controlled explicitly, at high level, by the programmer using a primitive go (similar to the one found in Telescript [8]). A strong migration mechanism is used, thus the movement of an agent involves moving its code, data and execution
state, to the target host in the network. On arrival the execution resumes transparently at the exact point where it was interrupted. An agent always carries the closure for its code, thus enabling disconnected autonomous execution.

Agents may provide (provides) services to other agents, and simultaneously, require (requires) services provided by other agents. There is no distinction between clients and servers. As service providers, agents must be able to handle multiple incoming requests. To cope with this demand MOB agents are multi-threaded. Threads can be explicitly created by an agent, through the fork instruction, or implicitly created, e.g. in a remote method invocation. The threads running in an agent have unique identifiers and share the agent’s address space. Synchronization with the parent thread can be achieved with the join method. Synchronization is also provided for exclusive data access, with methods lock/unlock and wait/notify at low level. Higher level control (e.g. synchronized method execution) can be implemented as derived constructs.

The interface for services is defined by the service construct. The interface is the contract for the service, the base for all service-oriented programming in MOB. By providing and requiring services, agents become the components of a service-oriented architecture. Checking that the interface of a service is correctly implemented is carried out at compile time by connecting to a network name service. The types inferred by the MOB compiler are matched against those assumed for the service in the network.

To access a service, an agent must get a binding for an agent that provides that service. The binding is obtained dynamically through the primitive bind that asks the name service for an agent that provides the required service. When the binding is received, interaction through method invocation can happen (much like Java RMI [9]).

MOB can also be used as a coordination language. For that we support the interaction with external services through the exec primitive. We supply an interface that allows the creation of a session, interaction with the server and the termination of the session. This interface can be used, for instance, to execute services implemented in other languages, or interact with network services, such as WWW queries, FTP transactions or SMTP.

The remainder of the language constructs provide fairly standard support for control flow, expression evaluation and built-in types.

SYNTAX

We start by describing the syntax for the MOB programming language [10], obtained from a core language by providing derived constructs for higher-level programming. The derived constructs keep the underlying semantics of the core language and are used, for example, to introduce data types such as arrays and hashes (associative arrays).

The syntax for a MOB program is presented in Figure 2. The language defines a set of reserved words identified in boldface. The main syntactic categories are: constants (booleans, integers and strings) ranged over by $c$; variables, ranged over by $x$; agent and class identifiers, ranged over by $X$; service identifiers, ranged over by $S$; method names, ranged over by $m$; expressions, ranged over by $e$; and, instructions, ranged over by $P$. A sequence of elements of a syntactic category $x$ is denoted $\overrightarrow{x}$. With this syntax, constant arrays are denoted by $\overrightarrow{c}$ and constant hashes by $\{\overrightarrow{c_1} \Rightarrow \overrightarrow{c_2}\}$, where the size of $\overrightarrow{c_1}$ and $\overrightarrow{c_2}$ must be the same. The concrete syntax of MOB imposes some restrictions to this syntax (e.g. class, agent and service declarations are allowed only at the beginning of programs).
A SERVICE-ORIENTED COMPUTING WITH MOBILE AGENTS

Figure 2. The syntax of the MOB programming language.

class MobNativeServices(descriptor) {
  read(n) // Reads n bytes from the standard input
  readln() // Reads a line from the standard input
  print(data) // Prints data to the standard output
  println(data) // Prints data to the standard output followed by a new line
  exec(program) // Executes a binary file
  getTime() // Obtains the current time of the host
  sleep(n) // Suspends the execution of the calling thread for n milliseconds
}

Listing 1. The interface of the MobNativeServices class.

The mob keyword is used to denote a built-in object that is loaded automatically at the beginning of every program. The object is an instance of the MobNativeServices class (interface presented in Listing 1) and provides an implementation for basic operations (e.g., input/output) and a simple interface based on sessions to run external programs. For example, the exec method encapsulates the execution of a command in the local file system and returns a handle for a session. Methods can then be executed on this handle according to the kind of session provided. The underlying implementation resorts mainly to the exec instruction.

To make the syntax more clear, we now give two small example applications written in MOB: a Time service and a mobile Weather Forecast server.

A naive time service

The server (Listing 2) provides the service Time with a single method getTime(). The initial behavior of an agent is defined by the main method. Note that main may be empty since MOB agents behave as daemons and thus must be explicitly terminated either with an exit instruction, or with some external command issued from the agent management infrastructure. For simplicity, we allow empty main methods to be syntactically removed from the agent’s definition. The compiler will perform the task but still emit a warning.

The server uses the getTime method of the mob object to obtain the local time.

The client (Listing 3) requires the Time service and takes an array of hosts as an argument. The agent loops over hosts, each time moving to the corresponding hostname, and sets the current time...
service Time { getTime }

agent TimeServer() provides Time {
    getTime() {
        return mob.getTime();
    }
}
new TimeServer()

Listing 2. A time server agent.

agent TimeClient(hosts) requires Time {
    main() {
        timeServer = bind(Time);
        for (hostname in hosts) {
            go(hostname);
            proc = mob.exec("setTimeApp " ^ timeServer.getTime());
            proc.wait();
        }
    }
}
new TimeClient(["host1", "host2", "host3", "host4"])

Listing 3. A time client agent.

to the value retrieved from the TimeServer. This operation is performed by resorting to method exec of mob to execute application setTimeApp.

A mobile weather forecast server

In this example (Listing 4), we focus only on the server part of the application. The server provides a service (WeatherForecast) for obtaining weather forecast information. It poses as an intermediate between the client and the actual forecaster, since it retrieves the information from an external entity. Moreover, the service allows for an extra operation to migrate the server to a new location. The operation is privileged and, thus, it requires some kind of authentication. This information is stored in a file (authFile) that must be passed to the agent.

The migration of the server moves the entire computation to the new host. This could include the authentication information. However, to avoid having it sent across the network and to illustrate how rebinding to local resources can be achieved, we choose to assume that the file (contents of argument authFile) is present in every host target of the migration.

The actual implementation of the service uses two classes: Auth that provides the authentication mechanism for operation migrate; and, the actual forecaster, a parameter of the agent that denotes an API to interact with the remote Web server, which provides the forecasting information. Note that the binding to the remote server will not be affected by the migration.

To obtain a forecast, the client must invoke one of the three methods: today, tomorrow or next5days. This will cause the agent to relay the request to the forecaster, and to send the result, using the inverse path, back to the client.
service WeatherForecast { today, tomorrow, next5days, migrate }

agent WeatherForecastAgent( forecaster, authFile ) provides WeatherForecast {
    auth = new Auth(authFile);

    today( city ) {
        return forecaster.today( city );
    }

    tomorrow( city ) {
        return forecaster.tomorrow( city );
    }

    next5days( city ) {
        return forecaster.next5days( city );
    }

    migrate( hostname, passwd ) {
        if ( auth.check( passwd ) ) {
            auth = null;
            go( hostname );
            auth = new Auth(authFile);
            return 0;
        }
        return -1;
    }
}

new WeatherForecastAgent( new BBCWeatherForecaster(), "/etc/mobwf/authFile" );


SEMANTICS

The semantics of the MOB programming language is given informally, by describing the transitions between network states. Each network state consists of a set of agents, running in parallel within the boundaries of hosts, and the name service (Figure 3). The name service‡ keeps two maps: one to keep track of the current location (host) for each agent, and a second that keeps, for each service, the type and the set of agents currently running in the network that provide that service.

The semantics specifies the changes in the network when a computation advances, one instruction at a time, in a chosen agent. Here, we focus on the most important network-wide operations in MOB: agent creation, migration, binding and remote method invocation. We refer the reader to [10,11] for a complete description of the semantics of the language.

Each agent (Figure 4) has an unique global identifier (a key used to identify the agent in the network but that is otherwise transparent for the programmer). The agent is internally composed of the code required for its execution, a heap to keep run-time data-structures, a pool of running threads (the agent’s execution units) and a set of suspended threads (waiting on heap references).

‡Here, we depict the name service as a centralized entity. This is how the service is currently implemented in our prototype. However, from the viewpoint of scalability, performance and resilience, it should be distributed. Such an implementation is future work.
Each thread (Figure 5) in an agent is composed of a stack of closures (some code plus bindings) waiting to be executed, a heap reference for synchronization operations and a heap reference where the result of the execution of the thread is to be placed.

**Agent creation**

Agents can be created by an user running a MOB program (Figure 6(a)) or by some autonomous agent during its execution (Figure 6(b)). The underlying mechanism is the same, a call with the `new` construct triggers the creation of a new run-time engine that holds the new agent’s data-structures. A closure with the state and references to all the codes that the agent requires is placed in the heap of the new agent. The execution of the agent begins with the `main` method. The agent is always created in the current host. Only through subsequent `go` instructions can the agent move to other network locations.
Figure 5. A MOB thread.

Figure 6. Creating agents.

Figure 7. Migrating agents.

Agent migration

Figure 7 illustrates the migration mechanism. To distinguish between agents in execution and in a migration process, the former are surrounded by a continuous line, while the latter are surrounded by a dashed line.

Migration is triggered by the go instruction that halts the execution of the agent and contacts the MOB service (more on this latter) at the host (Figure 7(a)). The service packs the code and the
state of the agent in a message data-structure and sends it to the host given as argument to the go instruction (Figure 7(b) and (c)). On arrival, the MOB service re-builds the agent from the message data-structure and resumes its execution at the point where it was suspended before migration took place (Figure 7(d)). Note that migration is subjective, that is, it is the agent internally that decides to migrate. The event is not triggered by external entities (e.g. other agents).

**Agent binding**

An agent may invoke a method in another agent only if it has a binding for the target agent. Agent discovery in MOB is service oriented, meaning that agents are picked for the services they implement, regardless of their individual identifications. The instruction bind (Figure 8(a)) consults the name service and retrieves an agent that implements a given service given as argument to the call. The name service returns a binding for a running agent that provides the service (Figure 8(b)). Once the binding is obtained, a communication link between the agents can be established (Figure 8(c)). The query may be refined by indicating the name of the host where the agent must be located. This is important in applications where the agent moves to a host and wishes to establish a local session with the server in a pure mobile agent style.

Although it may not be apparent from Figures 6 to 8, each of these three actions involves the name service. The creation of a new agent updates the name service to indicate in which host it is executing, and which are the services it provides. The migration updates the agent’s location in the name service. The binding operation reads service information from the name service and provides it to the agent.

**Method invocation**

Method invocation in objects can only be carried out from within the agent that owns them. All objects are encapsulated within agents and, thus, invoking a method in an object located in the address space of some other agent is not possible, unless the target agent’s interface provides some means to access the object. Methods of the agent itself can of course be invoked both from within the agent, and from other remote agents. This is after all how services are provided.

A method invocation in a remote agent always spawns a new thread (at the remote agent) that executes the corresponding code. The calling thread suspends until the result is available. In a remote method invocation, the result of the thread spawned in the target agent must be placed in the heap of the calling agent, in a distinct address space. Thus, once the thread at the target agent
terminates, a message is sent to the calling agent with the result and the latter is written to the heap. In the case of a local method invocation (an agent invokes one of its methods) simply written to the heap is sufficient to provide the result.

In MOB, all remote method calls are made by value, except for agents\(^\S\). When a call is made, the local agent computes the arguments of the call and builds a message data-structure with the computed values, all the code required by the values (e.g. classes), and a heap reference to hold the result of the call. The calling thread suspends on the heap reference for the result (Figure 9(a)). Using call by value and sending all the code required by the arguments avoids network references, and consequently all subsequent method invocations performed on the arguments of the calls are local. Note that classes are only uploaded to the target agent if the latter does not already hold them.

Once the message is received at the host of the remote agent, it is unpacked. The code is appended in the tables of the remote agent’s run-time engine and the arguments placed in a closure in the heap of that same agent. A new thread is started by the run-time of the remote agent and the code for the method is executed (Figure 9(b)). When the method returns, a message data-structure with the result and the heap reference to locate it is sent to the calling agent, again with the intervention of the MOB service at each host (Figure 9(c)). Finally, the returned value is written to the heap and the execution of the calling thread is resumed (Figure 9(d)).

Most of the remainder of the MOB instructions do not involve network interaction, changing only the state of the agent that executes them, and thus have simple semantics.

PROGRAMMING WITH MOB

In this section, we present two larger programming examples that illustrate the expressiveness of the language.

\(^\S\)Using call by value on agents would, in practice, define another form of agent migration.
A mobile number crumber

The example implements an agent that performs some computationally intensive task (e.g. number crunching) that requires a large amount of CPU time. The target hardware system in this example is a common LAN, in which computers are idle for substantial periods of time (e.g. during the night, weekends, lunch hour). To obtain the resources it requires to complete the computation, the agent moves through the LAN looking for idle machines that voluntarily donate their CPU cycles. This example illustrates how MOB may be used to program just the coordination between the agent and the underlying network infrastructure, thus completely encapsulating the computation to be performed. Allowing agents to dynamically move to computers where the required resources are available, increases the usefulness of the platform considerably and potentially allows almost uninterrupted execution.

The computation encapsulated by the worker agents may be implemented in a language other than MOB. The only assumption we make on the application is that it periodically checkpoints its state into a file and that this file may be used (as input) to resume the computation once the agent has moved to another host. The application is divided into three MOB modules.

The PortalAgent (Listing 5) supplies an entry point to the network. Its purpose is to welcome the registry of new hosts to the network and to allow the execution of new applications. The Portal service defines methods to allow new hosts (monitors) to join the network (enter), to retrieve the set of registered hosts (getHosts), to obtain the number of hosts participating in the cluster (getNoOfHosts), and the number of hosts available to receive work (getNoOfHostsAvailable). These data are collected by interacting with a MonitorAgent installed on each host of the LAN.

A MonitorAgent (Listing 6) begins by locating the portal and joining the network. At start-up, it specifies the maximum computational load allowed for applications in the local node. Then, it periodically checks the load of the host and uses it to decide whether it is possible to host an application. If the load is higher than a threshold, the parameter maxLoad, and a worker is running,

```moây
service Portal (enter, getHosts, getNoOfHosts, getNoOfHostsAvailable)
agent PortalAgent() provides Portal requires Monitor {
    members = []; // The hosts (monitors) registered in the portal
    enter(h) { members.put(bind(Monitor, h)); }
    getHosts() { return members; }
    getNoOfHosts() { return members.size(); }
    getNoOfHostsAvailable() {
        result = 0;
        for (monitor in members)
            if (monitor.getLoad() != -1)
                result++;
        return result;
    }
    new PortalAgent();
```

Listing 5. The PortalAgent.
service Monitor (getHost, reserve, getLoad, registerAndStoreFile, loadFile)

agent MonitorAgent(portalHost, maxLoad) provides Monitor requires Portal {
    worker = null; // The worker currently executing in the host
    load = 0; // The current work load
    available = true; // Availability of the host

    main() {
        portal = bind(portalHost, Portal); // Locate portal
        portal.enter(host()); // Enter the network
        while (true) {
            load = readLoad(); // Read the host's load
            if (worker != null) {
                if (load > maxLoad) { // If a worker is running
                    worker.leave(); // Command agent to leave
                    worker = null;
                }
            } else { // If no a worker is running
                if (load > maxLoad) available = false; // Not available
                else available = true; // Available
                mob.sleep(1000); // Sleep 1 second
            }
        }
        registerAndStoreFile(w, fileName, data) {
            worker = w; // Register worker
            // Open file and store contents
        }
        loadFile(fileName) { /* Read and return file contents */
        }
        getHost() { return host(); }
        reserve() { available = false; }
        getLoad() { if (available) return load; else return -1; }

        readLoad() {
            proc = mob.exec("getLoad"); // Obtain the load of the host
            result = proc.readln().toInt(); // Convert it to an integer
            proc.kill(); // Terminate execution of getLoad
            return result; // Return result
        }
    }

    new MonitorAgent($1, $2.toInt());
}

Listing 6. The MonitorAgent.

the latter is notified that it must leave the host. The Monitor service provides methods to retrieve
the name of the host it is monitoring (getHost), to reserve the host for an incoming agent (reserve),
to allow a worker to register itself in the host and store a file with given contents in the local
file system (registerAndStoreFile) to load a data file to memory (loadFile) and to retrieve the
computational load of the host (getLoad). The reserve method ensures that the migration from
one host to another is atomic, setting the host as unavailable to any other request during the
operation.
Finally, a WorkerAgent (Listing 7) encapsulates the application to be executed in a given host. It is a mobile agent whose parameters are: the name of the portal, the name of the application to execute and the name of the checkpoint file for the application.

The main method locates the portal and calls findHostAndGo to find a suitable host to install the computation. Once the host is selected in the latter method, the agent moves to it, saves the checkpoint data to a file and executes the application. The host where the application is deployed is selected based on the loads reported by the monitors of each host registered in the portal. In this example, the host with the lowest load is reserved. The reservation will only fail if another worker has already reserved the host. The leave method is used to suspend the execution of the service Worker { leave }

agent WorkerAgent(portalHost, app, dataFile) provides Worker requires Portal {
    portal; // The agent supplying the Portal service
    monitor; // The monitor of the current host
    data; // The data collected by the agent
    runningApp; // The process running the application
    MAX_LOAD = 1000 // A top for the load

    main() {
        portal = bind(portalHost, Portal); // Locate portal
        data = monitor.loadFile(dataFile); // Load file to memory
        findHostAndGo(); // Find a host to execute the application
    }

    leave() {
        runningApp.kill(); // Terminate the execution of the application
        data = monitor.loadFile(dataFile); // Load file to memory
        findHostAndGo(); // Find new host
    }

    findHostAndGo() {
        found = false;
        hosts = portal.getHosts(); // Get hosts in the network
        monitor = null;
        while (!found) {
            load = MAX_LOAD;
            for (h in hosts) { // For each host in the network
                aux = h.getLoad(); // Obtain load of the host
                if (aux != -1 & aux < load) { // Check if it is the one with a lower load
                    monitor = h;
                    load = aux;
                }
            }
            if (monitor != null) {
                found = monitor.reserve(); // Mark host as reserved
            }
        }
        go(monitor.getHost()); // Migrate to host
        monitor.registerAndStoreFile(self, dataFile, data); // Store data to file
        runningApp = mob.exec(app); // Run application
    }
}

new WorkerAgent($1, $2, $3); // Create worker agent

Listing 7. The WorkerAgent.
application in the current host when it ceases to provide the computational resources necessary to run it (for example, the owner of the host restarts using it). The method suspends the execution of the application and looks for a new host by querying the portal. Note that the method only returns once the migration is complete, and thus the latter is a synchronous operation.

**A mobile RPM installer**

In this section, we present an example that provides a simple way to automatically update the software (in the form of RPM files) in a network. The use of mobile agents in this example allows for the caching of the software to install in the agent, thus reducing the number of travels to the software repository. To have a similar functionality without mobile agents, a local mirror of the repository is required. The local hosts would then update by connecting to this local repository, avoiding the multiple download of the same information.

For simplicity, we assume that all target hosts have the same operating system and distribution, and thus no version control is required. We also do not implement user authentication, since it is not the focus of the example. The application is divided into four components.

The RepositoryAgent manages a repository of software files (Listing 8). It implements the Repository service that provides methods to store (put) and retrieve files (get), and to obtain the agent’s location (getHost). Method get takes a list of file names and creates and returns a hash in which the keys are the file names and the values are the contents of the files. The method put adds files to the repository.

Every host willing to be updated must execute an agent that implements the Installer service. The InstallerAgent (Listing 9) is the base station for other agents to dock and install the software. The Installer service is composed of two methods: install and getDeps. The install method performs the actual installation of the software given as argument. It creates a new file to hold the RPM to install, installs it by executing the rpm-i command, and finally removes the RPM file from the file system. It returns the message given by the RPM installation, or an error message, if the file could not be created. The getDeps method, returns an array holding the dependencies of the given file. The implementation is similar to install.

```
service Repository { put, get, getHost }

agent RepositoryAgent() provides Repository {

get(files) {
    result = {};
    for (fileName in files) // Place the contents of the files in the map
        result.put(fileName, readFile(fileName));
    return result; // Return map
}

put(fileName, fileContents) { ... }

getHost() { return host(); }

readFile(fileName) { /* Read the file to memory */ }
}

ew RepositoryAgent();
```

Listing 8. The RepositoryAgent.
service Installer { install, getDeps }

agent InstallerAgent(targetHost) provides Installer {
  main() { go(targetHost); }
  install(fileContents) {
    fileName = createFile(fileContents); // Create file
    proc = mob.exec("rpm -i ^ fileName"); // Install file
    proc.wait(); // Wait for completion
    result = proc.readln();
    removeFile(fileName); // Remove file
    return result; // Return result
  }
  getDeps(fileContents) {
    fileName = createFile(fileContents); // Create file
    proc = mob.exec("rpmDeps ^ fileName"); // Get dependencies
    proc.wait(); // Wait for completion
    result = proc.readln().split(" "); // Create array from string
    removeFile(fileName); // Remove file
    return result; // Return result
  }
  createFile(fileContents) { /* Create new temporary file */ }
  removeFile(fileName) { /* Delete file from folder */ }
}

for (i = 1; i < $$; i++) // For each host given as argument
  new InstallerAgent(i); // Launch installer

Listing 9. The InstallerAgent.

A DeployerAgent (Listing 10) is launched into the network and waits for installation requests. As the name suggests, the agent deploys the files required by the local instances of InstallerAgent. Requests get exclusive access to the agent and thus are serviced one at a time\(^1\). A call to the deploy method first resolves the location of the software repository and the Deployer agent moves to that host, in order to obtain the desired RPM files. Then, it moves through the list of hosts given in the call. At each host it obtains a bind for a local agent implementing the Installer service. Then, for each file to install, it checks for unresolved dependencies. The result (deps) will be matched against the list of files installed in the previous host (the keys of the cachedFiles hash). Once this operation is completed, the deps list will contain the name of the files required from the repository, and the cachedFiles hash the dependencies for the files that are already owned by the agent. If deps is not empty, it means that some dependencies cannot be satisfied with the files held by the agent and, thus, another trip to the repository is required to obtain the missing ones. Once retrieved, these will also be placed in cachedFiles, which keeps track of the dependencies to be resolved before installing the software.

The actual installation is performed by iterating through the cachedFiles and toDeploy hashes and calling install in the local InstallerAgent to install the packages involved. The use of both

\[^1\] A keyword to define synchronized methods can be easily derived with syntactic sugar as

\[
synchronized m(\varnothing) \{ P \} \overset{\text{def}}{=} m(\varnothing) \{ \text{self.lock(); } P; \text{self.unlock(); } \}
\]
A SERVICE-ORIENTED COMPUTING WITH MOBILE AGENTS 721

service Deployer { deploy }

agent DeployerAgent() provides Deployer requires Repository, Installer {

main() {

deploy(fileNames, hosts) {
    self.lock();
    rep = bind(Repository);  // Discover the repository
    repHost = rep.getHost();  // Obtain the host where it is located
    go(repHost);  // Migrate to its location
    toDeploy = rep.get(fileNames);  // Get files to install
    cachedFiles = {};
    deps = {};
    for (h in hosts) {
        go(h);  // Migrate to a target host
        installer = bind(Installer, h);  // Obtain a binding for the local base station
        for (file in toDeploy) {  // Get dependencies
            deps = installer.getDeps(file);
            if (! deps.isEmpty()) {  // Dependencies are not fulfilled
                go(repHost);  // Migrate to the repository
                cachedFiles.put(rep.get(deps));
                go(h);  // Migrate back to the host
            }
        }
        for (file in cachedFiles)  // Install dependencies
            installer.install(file);
        for (file in toDeploy)  // Install requested files
            installer.install(file);
    }
    self.unlock();
}
}
new DeployerAgent();

Listing 10. The DeployerAgent.

requires Deployer
bind(Deployer).deploy( $1.split(" "), $2.split(" "));

Listing 11. The Client script.

hashes separates the static set of requested files from the host-dependent, dynamically gathered dependencies.
Finally, Client is a component that locates a network agent that provides the Deployer service and
instructs it to install a set of files in a set of hosts. Its implementation (Listing 11) receives a string
with the hosts ($1), and a string with the names of the files to install ($2) from the command line.
THE COMPILER

The MOB programming language is compiled into an intermediate language called DiTyCO extended with a special object called mob. As we mentioned before, DiTyCO is a programming language based on a process calculus with support for distributed computing and resource mobility [4].

The language describes computations as concurrent interactions of processes through communication reflecting the paradigm of the modern theory of concurrency and mobility [12,13]. DiTyCO does not fully support the mobility of computations so we had to extend it with the built-in object mob that, among other things, provides this functionality.

The option to use DiTyCO as an intermediate language for the compilation of MOB applications was based on the following principles:

- the MOB abstractions, agents and services are mapped in a straightforward way into objects and their types, respectively, in DiTyCO;
- such mapping provides us with a far higher-level language for encoding new constructs than we would have if we decided to map MOB directly onto the low-level instruction set of the underlying virtual machine (to be discussed in the next section);
- finally, since an infrastructure for DiTyCO was already implemented in the previous work [6,7,14], if we compile MOB into DiTyCO, we could take advantage of the compiler and run-time environment developed for DiTyCO.

The compilation of a MOB program is a two-stage process (Figure 10). The first stage of the compiler takes the MOB source code and outputs the corresponding code in the DiTyCO language. The second stage of the MOB compiler is just the compiler for the DiTyCO language [14]. The DiTyCO compiler outputs code written in an intermediate language called Multi-threaded Intermediate Language (MIL [15]). This is a textual representation of the executable programs written in the instruction set of the DiTyCO virtual machine. This code is compiled just-in-time (JIT) into a byte-code internal representation by a DiTyCO virtual machine before being executed.

The first stage of the compiler performs type inference on the MOB source code and in particular finds the types for both the services provided and required by each agent defined in a program. At this point of the compilation, the name service is contacted and a type-check is performed. The types locally inferred for the services by the compiler are matched against those assumed

Figure 10. The compilation and execution of a MOB program.
for the services in the network. If a given service is being registered for the first time in the network, the interface provided becomes the interface for that service. If the types do not match an exception is raised and the compilation aborts. This level of type verification provides some form of program security for remote method invocation. If the type-check succeeds, the source MOB program is transformed into a program written in the DiTyCO programming language. To allow for off-line compilation, this stage could be pushed to the beginning of the execution, namely, during the JIT compilation of the DiTyCO code into the internal byte-code representation. This has the disadvantage of making it much harder to provide meaningful type error messages to programmers since part of the compilation context may by that time be lost.

For each agent definition in a program, a proxy in DiTyCO may be generated. These proxies are located at the same site as the agent, and are components that act as intermediaries between the agent and clients in the network. However, a proxy is only useful if it can be reached by the user. For that purpose, the compiler generates an extra DiTyCO file, which we will refer to as proxy connector. This file, when compiled and executed, locates the proxy for the agent and allows clients to interact with the latter.

Proxy connectors use an agent’s unique network-wide key to connect to its associated proxy. Thus, clients communicate with agents by deploying a proxy connector that establishes a connection to the proxy for the agent. After this is accomplished communication between the client and the agent (server) proceeds with the proxy acting as a mediator. Details of DiTyCO site discovery and proxy implementation will be given in the next section.

The second stage of the compiler, takes the DiTyCO code generated by the first stage and processes it with the DiTyCO language compiler to produce .mil files for each agent, and, eventually, for their proxies. Only well-typed DiTyCO programs are successfully compiled into the final MIL code. The MIL code thus generated is executed using the run-time system for the DiTyCO programming language [6] extended with support for strong mobility (Figure 10).

THE RUN-TIME SYSTEM

Agents run within the boundaries of hosts. In MOB, this host layer is implemented on every MOB-enabled host in a network and on top of the DiTyCO run-time system. This layer is responsible for managing agents within hosts and supporting their mobility. It is implemented as a network service and provides the means to create, run, marshal and move agents. The implementation takes advantage of the fact that the MIL virtual machine [6] is implemented on top of the Java Virtual Machine (JVM). This, for instance, enables the access to the full state of an agent and makes the marshaling operations quite straightforward to implement besides preserving portability. For example, the MOB run-time system can provide a strong migration mechanism in a simple and transparent way by stopping and serializing a running agent that executes the go primitive.

The next layer corresponds to the agents themselves and the name service. Both the agents and the name service are implemented with DiTyCO run-time systems. Each DiTyCO run-time system has two components (Figure 11):

- an instance of the MIL virtual machine that runs the MIL code for the agent or for the name service;
- a communicator that is responsible for handling network communication for the MIL virtual machine.
Internally, each MIL virtual machine is composed of a heap, a hash of MIL program fragments and a pool of JVM threads that consume/produce tasks from/for a set of local run-queues. The tasks are activation records for fragments of MIL code generated by the compiler from the original MOB program. They are scheduled for execution in the run-queues and executed by the available threads [6]. A shared run-queue also exists to allow for load-balancing between the threads running the virtual machine (Figure 11).

Besides providing a communication interface between the local agent and its remote counterparts, the communicator also keeps track of the bindings for outgoing and incoming resources by mapping them in the heap of the MIL virtual machine and keeping a marshaling table using heap addresses for keys.

**AGENT MIGRATION AND COMMUNICATION**

The migration of agents introduces the problem of maintaining a coherent state in the network, namely in what concerns inter-agent communication. In fact, an agent may end up with an invalid binding for another agent if the latter migrates to another host in the network in between connections (Figure 12).
A SERVICE-ORIENTED COMPUTING WITH MOBILE AGENTS

MOB uses a protocol [10] that ensures reliable communication in the presence of agent mobility. The protocol addresses:

- getting bindings for agents;
- tracking the location of an agent in the network;
- allowing the migration of one, or both, peers (agents) in a communication link.

A general overview of the protocol is as follows.

**Binding services**

As we have described in previous sections, agents use the **bind** construct to get bindings for agents in the MOB network that implement the services they require. The **bind** construct uses the name service to get the bindings for agents (Figure 13). As a result of these queries to the name service, a client agent receives a unique network agent identifier that is used internally by the MOB run-time system to locate and establish communication links between agents. Note that the agent identifier is used only internally by the run-time and remains the same at least until the agent rebinds that service. The programmer is never aware of its existence.

**Tracking agents**

We track the movements of agents using the name service [10] that holds the updated locations of the agents running in the network. Unlike other approaches [16], we do not propagate the changes in the network’s topology resulting from agent movement. We choose a **lazy reconfiguration** protocol, where: (a) a connection between an agent and the name service is established only when the
resolution of a location is necessary for communication to happen, and; (b) a moving agent registers its new location with the name service as soon as it is installed in the new host.

When a client (b) tries to establish a communication link with a server (a), it uses the agent identifier it obtained from the last bind of that service. Two situations may arise: (a) the agent identifier is up-to-date and the run-time establishes the link (Figure 14); or (b) the run-time system is not able to use the agent identifier to establish the link. In the latter case, two situations may occur: (a) the agent in question moved to another host in the meantime since the last connection between the agents was established; or (b) the agent in question is in transit. In the first case, an extra query to the name service will update the agent identifier with the new location and the link can be re-established by the underlying communication protocol (Figure 15). Note that after migrating, an agent must always register with the name service and provide its new location. In the second case, the client agent (b) must wait until the server agent completes its migration and registers again with the name service. Here, the agent identifier associated with the client (b) is stored in a table in the name service that keeps the agent identifiers of all the agents waiting to establish a connection with the server agent. When the server agent finally registers again with the name service, the table is read and the corresponding agents are notified with the updated agent identifier data-structure for the server agent (Figure 16).

This protocol only allows us to locate agents in the network in the presence of migration. It is not clear, in this scenario, what happens to the existing communication links when the endpoints migrate. This is addressed with a protocol that ensures the reliability of the communication links in the presence of migration.

**Communication reliability**

The protocol we propose is an extension of the one proposed by Zhong and Xu [17]. Their work provides a middleware layer (NapletSocket) placed between the application and the Java sockets API, that provides reliable communication on an established TCP connection. They bring concepts
used to keep TCP connections alive during physical mobility, used, for instance, in TCP-R [18] and M-TCP [19], to the field of logical mobility. They add two new operations over the socket: suspend and resume. Before migrating, an agent suspends all the connections it holds resuming them once it reaches its new destination.

Our protocol differs in the sense that it does not impose any restrictions on the communication protocol. The NapletSocket is a middleware between the application and the TCP protocol, while our approach works on top of the protocol used by the application, be it synchronous or asynchronous, connection oriented or not, transport or application layer, as long as it respects a given interface. Furthermore, our approach allows for overlapped migration, which is not supported by the NapletSocket, where only one of the agents involved in the communication can migrate at a time. This introduces the overhead of resuming a connection only to be suspended again.

Thus, as in [17], our protocol extends the interface of the communication link with the suspend and resume operations. A call to suspend is made when one of the endpoints (b) of the communication link wants to migrate. The call sends a control message to agent (a), at the other endpoint, informing it that the link is temporarily unavailable, and that it will be resumed as soon as the moving agent resumes its execution at the new host. The resume operation is used by agent (b) to re-establish a communication link with (a) once it reaches its new host (Figure 17).

**Overlapped migration**

When agent (b) tries to re-establish a communication link with agent (a), it may happen that the latter is in transit or actually moved to a distinct host. In this case, agent (b) must contact the name service to first locate agent (a) (Figure 15), before re-establishing the link (Figure 18).
As we described above, if the query to the name service to obtain the new location of (a) fails then, the identifier for agent (b) is placed in a table holding agents waiting for agent (a) to register again with the name service. Agent (a) will in turn know the new location of (b) when it registers its new location in the name service and the latter notifies all the pending agents.

AGENT EXECUTION AND MANAGEMENT

To run a MOB script we use a `mob` command that connects to the underlying MOB service at a given host. A MOB script is run by specifying its .mil file, the host where it should run (which must have the MOB service running on it) and the command line arguments.

As we mentioned in section The compiler, agent management in MOB is based on proxies that manage communications between proxy connectors and the agents. The code for the proxy and the proxy connector associated with an agent is generated automatically by the MOB compiler. Both proxies and connectors can run anywhere in a network with an underlying infrastructure for MOB. A proxy connector is a client application that allows an user to interact with a remote agent by finding it in the network and establishing a session with its proxy. Once the binding for the proxy is obtained, the user may interact with the agent through a shell interface that transparently uses the proxy to relay the communication. The proxy connector manages to find the remote agent and its proxy since they share an unique key. In Figure 19, we illustrate the procedure for the WeatherForecastAgent example from Listing 4, using a key `myWeatherForecast` (1), as follows:

```
> mob WeatherForecastAgent.mil myWeatherForecast
```

If a system administrator wishes to move the server to another host to perform maintenance on the current host, it runs the proxy connector with the key `myWeatherForecast` as an argument (2):

```
> mob WeatherForecastAgentConnector.mil myWeatherForecast
```

The proxy connector finds the location of the proxy for the web server using the name service. The name service replies a binding for the proxy (3) and a session is established. The client may
now interact with the agent using the proxy as intermediate (4), for example, causing it to move to another host, newHost (5):

> migrate newHost

RELATED WORK

*Service-oriented computing* builds on the pre-existing concepts of object-oriented and component-based programming and the client–server paradigm. The programming model is borrowed from the object-oriented paradigm, as services are accessed much in the same way that objects are. Services are described in a platform-independent way by contracts (service interfaces), which are negotiated between the components of an application. Inter-component communication is based on the client–server paradigm. However, unlike typical client–server architectures where a client is linked to a given server during the entire operation, in service-oriented architectures the client–server model is used to request services in a peer-to-peer organization. A component is not tied to a single server and there are no client–server hierarchies between the components that provide and require services. In service-oriented computing, the component-specific implementation details are hidden from the world, and only the interface to interact with the component is published. Services may be provided by many different components and, thus, components may bind to different implementations of a given service, in distinct moments in time. This is a major advantage. A component may fail
without disrupting the entire system. When a service provider fails, another instance of the service may be found (Figure 20). A particular application of this feature is the ability to replace service implementations on-the-fly, without losing the capability of supplying the service to clients. Most of the first service-oriented architectures were built resorting to DCOM [20] or to CORBA [21]. Component-based systems have recently received a lot of attention for distributed systems, namely with the .NET [22], Jini [23] and Openwings [24] platforms.

*Mobile agents* are computations that have the ability to travel through a network, by halting their execution, saving their state and then restoring it in a new host. As they travel through the network, mobile agents use resources (e.g. data, servers) thus focusing on local, rather than remote, communication. This contrasts with the usual communication paradigms (e.g. client–server), which require costly remote sessions to be maintained. As illustrated in Figure 21, once an agent is launched into the network the communication link may be disconnected. Further communication is only needed to check if the agent has completed its execution, and to retrieve the results, if any.

Programming languages and frameworks for mobile agents may be divided into two classes. Those in the first class use some existing networking infrastructure or middleware (mostly Java and its networking facilities) and implement the programming model top-down. Examples of such systems are Aglets [25], Mole [26] and Voyager [27] that mostly extend Java classes to define an agent’s behavior. Since it is not possible to access the state of the JVM, and modifying it would mean losing portability, all these systems resort to a weaker kind of migration. Instead of moving the whole computation to the new location and resuming the execution in the exact same point where it was interrupted, only code and data are moved, forcing the programmer to implement *receiver code* that re-activates the agent after a migration. Another approach, followed in systems such as D’Agents [28] and Ara [29], fully supports agent migration but requires specific virtual machine support.

![Figure 20. Replacing a component in a service-oriented architecture.](image1)

![Figure 21. Disconnected execution of mobile agents.](image2)
These Java-based systems usually supply a set of Java classes which the programmer must extend to implement applications. As in MoB, the run-time architecture of these systems is based on abstractions for IP nodes running a hosting service.

In Aglets [25], inter-agent communication is performed by dispatching instances of a Message class, which are relayed by a proxy on the target agent’s side, much like in MoB. Communication is point-to-point and group communication is not supported. Each agent is executed in an independent Java thread.

Mole [26] agents are clusters of objects. References to agents in Mole are symbolic, each agent being identified by a unique network-wide name. In MoB, references to agents are true heap references and are kept by the agent as it moves through the network. Communication in Mole uses a set of pre-defined mechanisms supported by Java such as RMI or asynchronous message passing.

In Voyager [27], an agent is also a collection of objects. However, objects (including agents) may be created remotely without implementing any special interface. Voyager generates all the required files, and handles all messaging. Communication is carried out through method invocation or through a shared space. As in MoB, Voyager supplies basic communication mechanisms, reserving higher-level communication or protocols to the application programmer.

A mobile agent in D’Agents [28] (former Agent Tcl) may be written in any language, although TCL is the one mostly used. All the services and resources required by an agent (e.g. state capture, migration, communication, disk access) are made available at each host by a dedicated server. D’Agents implements a weak form of migration since, although agents resume execution in the next instruction of the program after a migration, no state is carried by the agent.

Mobile agents in Ara [29] are programmed in an interpreted language. The interpreter for this language is a special run-time system for agents called the core. A special core call allows agents to migrate at any point of their execution, and the agent carries the full state of the computation with it (as opposed to D’Agents).

Systems composing the second class define their own language based on some formal system, usually some form or extension of the π-calculus [12,13], and follow a bottom-up approach to design. Process calculi provide a theoretical framework upon which researchers can build solid specifications for programming languages. Languages can thus be proved correct by design relative to some base calculus with a well-established theory, by providing adequate encodings. Examples of such languages have been implemented in recent years, namely, JoCaml [30], TyCO [4], X-Klaim [31], Nomadic Pict [32] and Acute [33]. The bottom-up development of these languages has its drawbacks. In fact, although process calculi are good formal tools for the development of mobile agent frameworks, their constructs are very low-level and high-level idioms that provide more intuitive abstractions for programming are desirable.

The first such system we describe more in depth is Jocaml [30], a programming language based on the Join calculus, which provides support for distributed and mobile agent-based applications. The language uses a custom virtual machine and supports the migration of trees of computations (an agent and its tree of sub-agents) between hosts in a network. Just as in MoB, type-checking is mostly performed at compile time except for interaction with other modules which is performed dynamically. The required type information is annotated in the source program. In Jocaml, communication is preserved in agent movement by resolving the location of agents and letting them forward the messages to the corresponding channels.
The M-Calculus [34] is an extension of the Distributed Join calculus that incorporates several new notions, such as programmable agents and dynamic binding. Agents cannot move in the network but they may exchange data and code. Its focus is on resource access control and safe communication, defining agents composed of a membrane (to control accesses) and a content (where the actual computation occurs). The current implementation uses a distributed abstract machine called CLAM (CeLular Abstract Machine), and provides a centralized implementation, the C-VM (Cellular Virtual Machine) [35], which was developed on top of the JVM. Each agent is executed by a single dedicated Java thread. In MOB, we defined the MOB abstract machine, and implemented it resorting to the DiTyCO distributed implementation. Each MOB agent is mapped into a DiTyCO run-time system, which is executed by the MOB host service. Each agent is multi-threaded [6].

The X-KLAIM [31] programming language is an implementation of the KLAIM model with ad hoc extensions to incorporate higher-order constructs, asynchronous reading of tuple-spaces and hierarchical structured networks. Programs in X-KLAIM are compiled into Java classes that resort to a package, Klava [36], to run. A mobile agent in X-KLAIM is a process with a single execution flow, rather than the multi-threaded agents found in MOB. This makes the migration of a multi-threaded agent, scattered among several processes at a given site, a complex and user-aware operation. Since KLAVA uses the JVM as the underlying virtual machine, it cannot support mobility of computations or agents directly. The X-KLAIM compiler transforms the programs so that it is possible to support a limited form of mobility for agents even without access to parts of the state of the JVM.

Nomadic Pict [32] is perhaps the closest to our work in that it grows from another process calculus-based language, Pict [37], and adds primitives for programming mobile computations, such as agent creation, agent migration and asynchronous communication between agents. Despite the similar approach, our emphasis is on producing a compact, user-friendly, scripting language that abstracts away from network location-dependent information. In this respect, we feel that MOB, even in its more basic form, is higher level than Nomadic-Pict, providing built-in multi-threaded agents and a service-oriented paradigm.

Acute [33] is a programming language for mobile agents built on top of the Objective Caml programming language. The language provides type-safety through a mixed static/dynamic type-checking scheme, e.g., on module linking. Moving agents is achieved through an atomic operation that captures a collection of threads in a structure (a thunk) that can afterwards be marshaled and moved across the network. This contrasts with MOB where marshaling of objects or agents is transparent to the programmer. In the case of MOB agents the primitive go implements the marshaling required for sending the agent to another node in the network. The MOB service on that node will be responsible for unmarshaling the agent and restart it. In this respect, Acute provides a finer, lower level, control over migration and marshaling than MOB.

CONCLUSIONS AND FUTURE WORK

In this paper, we described MOB, a programming language for developing applications based on mobile agents, which, uses a service-oriented approach for resource discovery.

The combination of both the service-oriented and mobile-agent paradigms in a single programming language and run-time system provides a very high-level model for programming distributed applications for today’s networks. MOB agents simultaneously provide and use services in a network.
The service interfaces constitute contracts that bind agents among themselves. Bindings for agents are obtained based on the services the agents provide rather than an agent identification scheme. This allows programmers to abstract away from the physical locations and from the identities of individual network agents. We envision that a class of distributed applications may benefit from MOB’s programming model, namely, services developed for networks with highly dynamic, volatile resources or simply highly adaptive, reconfigurable applications running on more classical networks.

One of the major hurdles of developing MOB was the impossibility of doing it within the framework of a high-level language such as Java or C#, given the restrictions imposed on the access to the state in the corresponding virtual machines. Such an approach is desirable given the huge amount of resources available for these languages that would automatically be available to MOB programmers. Unfortunately, moving entire Java or .NET virtual machines has not been feasible up until now. We are currently studying the possibility of using virtual machine technology, such as Xen [38] and Linux Kernel-based virtualization, KVM [39], to support agent movement.

Regarding the MOB language, we wish to upgrade it with code reuse and exception-handling mechanisms, as well as with a number of APIs. Class inheritance, mixins and inner-agent components are probable paths for a code reuse mechanism. Regarding API provision, our priorities are the implementation of external services, such as language-independent service description and communication standards, and interaction with local resources, such as databases and printers.

REFERENCES