CHAPTER 4

THE MOBILE AGENT FRAMEWORK

4.1. INTRODUCTION

This chapter comprises the introduction to a core Mobile Agent (MA) framework design and implementation issues. The framework is intended for use in Network & Systems Management (NSM) applications, hence, a number of management-related issues have been taken into account and several optimisations have been performed to achieve NSM orientation. The presented infrastructure exploits the benefits of MAs to carry out semantically rich management operations in a highly scalable, flexible and efficient manner.

The design and implementation ideas introduced in this chapter have been originally published in the proceedings of the *IEEE International Conference on Communications* (ICC'99). The description of several extensions have been included in papers published in the proceedings of the *4th IEEE International Symposium on Computers and Communications* (ISCC'99) and the *3rd International Workshop on Intelligent Agents for Telecommunication Applications* (IATA'99). Full references are given in Appendix A.

The remainder of this chapter is organised as follows: Section 4.2 highlights the main goals identified for the framework's design, with special focus on the particular needs and requirements of management applications. Section 4.3 summarises our Mobile Agent Platform (MAP) design. Section 4.4 comprises the core of this chapter, describing the implementation of the introduced infrastructure. The features of the main building components are presented in detail. Special focus is given on the framework's fault tolerance features and its class loading mechanism. A quantitative evaluation of the framework is presented in Section 4.5, where simple mathematical formulations are devised to model the response time and network overhead of centralised and MA-based management.

To thoroughly evaluate the performance of the proposed framework, a number of timing and network overhead experiments have been conducted and presented in Section 4.6. A first set of measurements compares the performance of MAs against that of Java Remote Method Invocation (RMI) both in terms of response time and network overhead. Following that, we present experiments that evaluate the dependency of multi-hop MAs response time on the amount of collected data and the itinerary length. The empirical results of these experiments are contrasted with the theoretical findings of the quantitative evaluation and ways for improving the MA framework's performance are identified. Finally, Section 4.7 summarises and draws the conclusions of this chapter.

4.2. MAIN GOALS OF THE MOBILE AGENT FRAMEWORK DESIGN

The following points have been identified as the main goals regarding the design and development of our MAP:

- *Network management applications orientation*: Unlike publicly available general-purpose MAPs, the introduced architecture should take into account the special characteristics and requirements of NSM applications and therefore provide a flexible, scalable and lightweight environment tailored to management operations. In addition, a variety of features typically offered by general-purpose MAPs may be omitted as they do not add any value to an NSM-oriented framework, while increasing the complexity of the platform and its requirements on system resources.
- *Mobility support*: The ability of objects to move from host to host offers a powerful abstraction that should not be disregarded when building distributed management applications. The architecture should therefore provide a set of services to allow the *migration* of management agents, regardless of the underlying management models. *Cloning* of MA objects should also be supported in order to allow instant creation of MAs, when it is required to share the responsibility of a management task among multiple MAs with identical functionality or minimise MAs deployment latency.
- *Modularity*: Maintaining an architecture with intrinsic modularity eases the addition of new services or the modification of existing ones.
- *Support for existing management standards*: Our architecture encompasses the dominant NSM framework of Internet world, i.e. the Simple Network Management Protocol (SNMP). Due to its huge installation base, integration with SNMP was considered of vital importance to maintain compliance with legacy management systems. The importance of

integrating MA-based management environments with SNMP is also discussed in [SIM99, PAG00].

- *Support for 'disconnected' operations*: Management operations dependency on network resources should be minimal, in other words, it is essential that management tasks execution is not vulnerable to link or node failures or sensitive to traffic conditions. The required support for disconnected operations can be guaranteed by using autonomous MA entities, which are able of performing their decentralised management tasks without requiring communication with a central manager station. These MAs should be able to continue their execution even when the communication link with the manager is disrupted or the manager station itself fails.
- *High performance*: A variety of network performance management tasks have very demanding time requirements. It is therefore of critical importance to design an infrastructure that guarantees prompt execution of time sensitive monitoring tasks, allowing sophisticated Network Management Systems (NMS) to foresee possible congestions or failures and take preventive measures before the actual error occurs. As a result, MA migrations should be optimised in terms of their latency and MA objects assigned the highest priority among the processes executing on the same system. In addition, the itineraries of multi-hop MAs carrying out management tasks should be optimally designed so as to minimise the overall response time.
- *Scalability*: Scalability is of major concern when designing distributed management applications. MA-based approaches in NSM promise scalable solutions, however frequent transfers of MAs may cause excessive use of network bandwidth and result in poorer performance than centralised frameworks. Hence, it is essential to ensure that MA migrations take place only when they are absolutely necessary, whilst imposing minimal burden on network resources when they occur.
- *Lightweight footprint*: The applications required to receive, instantiate and launch MAs (Mobile Agent Servers) should be designed so as to be as lightweight as possible and therefore provide execution environments that can be installed on any network device, including those with limited storage capacity and processing capabilities.
- *Security*: One of the main arguments commonly used against MAs is the security threats this technology brings forth. Hence, security concerns should be carefully taken into account, with the designed architecture offering protection against MA attacks and the MA objects shielded from tampering and eavesdropping. In addition, access control on MAs actions on system resources and services should be supported.

- *Fault tolerance*: A MAP should be able to cope with situations where link or node failures disrupt the normal migration process of roaming MA objects. Fault tolerance features should also cope with the scenario where the failed node is the manager station itself and ensure that the valuable management information collected by the MAs is not lost.
- *Dynamic class loading*: Existing management tasks often need to be modified. These actions should be carried out in a user-transparent manner, without disrupting the NMS operation. That could be achieved through a network class loader (CL) installed on every managed device, able to receive at runtime the definition of an MA object representing a management task. CLs should allow the NMS not only to introduce new services, but also to replace existing ones; in other words, CLs should be able to distinguish different versions of the same MA class.
- *Ease in introducing new services*: It is essential to provide an open architecture in which the administrator can easily add new services at runtime. In our framework, there is a direct mapping of management functions to certain MA objects. The effortless introduction of new services is accomplished through a graphical tool that automates MA code generation and eases the deployment of new MAs that carry out specialised management operations.
- *Platform Independence*: Last, but not least, the implementation of a framework that supports any network device, regardless of the underlying hardware platform or operating system, is crucial in order to cope with the heterogeneity of modern multi-vendor networking environments. The "write once - use everywhere" slogan of Java promises to achieve the required degree of platform independence.

4.3. Overview of the Introduced Mobile Agent Platform

4.3.1. Implementation Language

The introduced platform is entirely developed in Java due to its inherent platform independence that makes it portable across distributed heterogeneous environments. This and other features, highlighted in Section 2.7.1.1 and 3.5.2, suggest Java as a suitable platform for developing MA-based applications. Hence, the selection of Java as implementation language for our framework has been a natural choice. In fact, the vast majority of MAPs developed over the past few years are implemented in Java (see Sections 3.5.6 and 3.6.1.).

The framework's source code has been developed using the Java Development Kit (JDK) 1.1 library, which was the most recent JDK version at the time the implementation of the framework commenced. Although the compatibility of the prototype with the JDK's latest

version (1.2) has not been tested, it is expected that only minor changes are required to achieve full compatibility.

4.3.2. Framework's Overview

The framework introduced in this chapter provides the basic functions required to program MA-based applications. Nevertheless, several optimisations have been incorporated during its implementation to achieve a flexible design that not only has minimal requirements on distributed system and network resources, but also integrates special features particularly suited for NSM applications. The reasons that led to the decision to implement a new MAP from scratch rather than using one of the many available general-purpose MAPs, have been highlighted in Section 3.5.6.

In our platform, the interface between the visiting agents and legacy management systems is achieved through the Mobile Agent Servers (MAS). A MAS provides incoming MAs entry points to services, represented by Java objects within the MAS. Security issues have also been addressed including authentication of visiting MA objects, encryption of sensitive management information and authorisation of MA actions.

A manager application has been developed to co-ordinate the policies of monitoring and controlling the Network Elements (NE). In this chapter, we investigate the performance of the "standard" use of MAs, i.e. multi-hop MAs, that is the agent code migrates *sequentially* between managed devices with data processing performed locally. This alleviates the need for broadcast polling and can result in a significant reduction in NSM data at the source, as only high-level management information is delivered to the manager.

An MA is an instance of a user-defined Java class. This class must inherit the properties of a generic class that provides all the core functionality to implement mobility. The platform implements *weak mobility*, namely the MA can carry its persistent state but not its execution stack. The implementation of *strong mobility*, apart from representing a non-trivial task as explained in Section 3.5.5.1, is not considered essential for performance management applications. That is, MAs are typically required to start execution from a specified entry point (method) rather than resuming their operation from the point their execution was interrupted on the previously visited host [CAB00].

Unlike all known publicly available MAPs, which involve the transfer of both the agent's code and state information on every migration, the framework introduced herein adopts a more lightweight mechanism whereby the bytecode is transferred to each managed device only once, at the MA construction time (we assume that NEs have fairly large storage capabilbity); following that, the transfer of the MA state is sufficient to carry out distributed management

tasks. That results in moderating the demand on network resources, since bytecode size is typically much larger than state size [BAL98].

We also introduce a novel tool prototype, the Mobile Agent Generator (MAG), which creates MAs in response to service requirements. Such MAs may be generated post MASs initialisation to accomplish intelligent management tasks, tailored to the needs of a changing network environment. Thus, the MAG realises a flexible infrastructure through the creation of new MA instances, which dynamically extend NMS functionality.

The introduced prototype is logically structured in *7 packages* [ARN96], each of which includes a number of classes implementing a specific part of the framework. A short description of these packages is given in Table 4.1.

Package	Number of classes	Description
`GUIs`	21	Implements the framework's user interfaces
`Manager`	15	Implements the manager application and its building sub-components
`MAS`	12	Implements the Mobile Agent Server and its building sub-components
`MCode`	10	The MA 'base' class, its service-oriented sub-classes and several classes for manipulating MA objects
`RMI`	6	Implementations of the RMI servers and their respective interfaces
`Security`	2	Implementation of cryptography algorithms and a customised SecurityManager
`Utilities`	6	Various utility classes

Table 4.1. Structuring of the prototype in packages

Full details on the MAP's building blocks implementation are given on Section 4.4.

4.4. INFRASTRUCTURE OVERVIEW - IMPLEMENTATION DETAILS

4.4.1. The infrastructure's main building blocks

The infrastructure consists of the following major components (see Figure 4.1):

- The Manager, responsible for launching and controlling MAs and displaying results;
- The MA objects, capable of migrating between the managed entities to collect information based on pre-defined policies;
- The Mobile Agent Server (MAS), capable of receiving MAs and providing an interface to the local physical resources;

- The MAG, a factory that generates service-oriented MA objects.

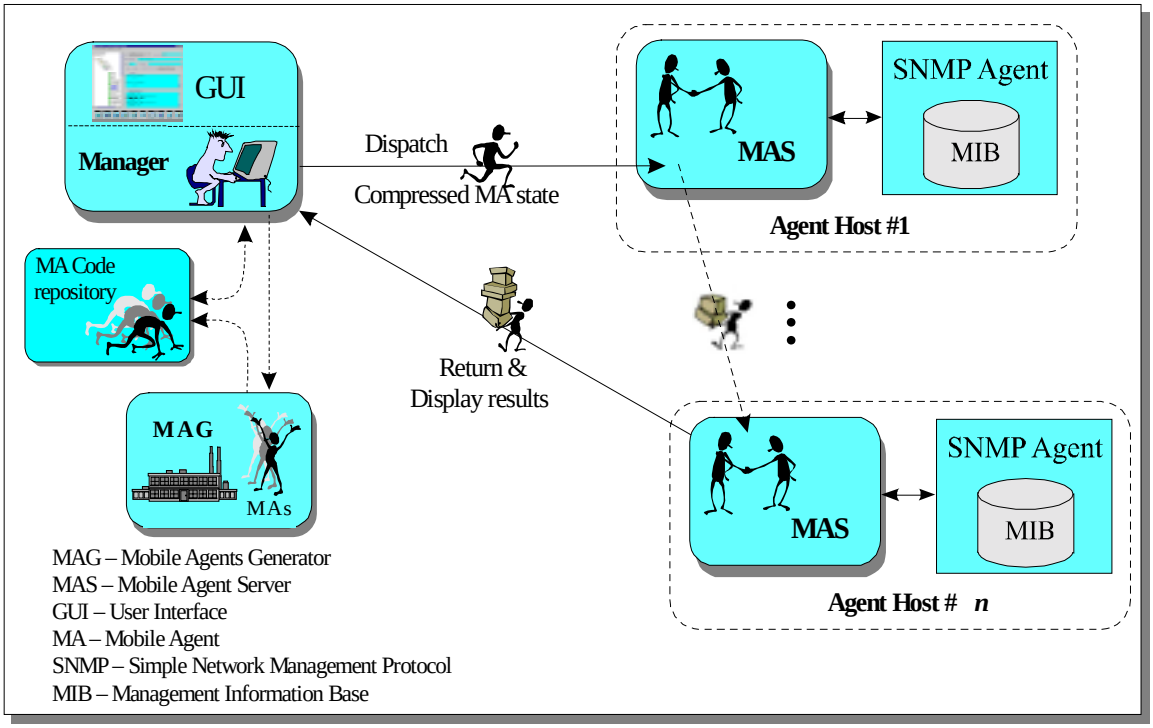


Figure 4.1. The Mobile Agents-based Infrastructure

The following sections elaborate on the design and implementation details behind its one of these components.

4.4.1.1. Manager application

The manager application performs monitoring and control operations through interacting with devices running agent processes. It composes a multithreading environment where a main thread instantiates, controls and co-ordinates the operation of a number of specialised threads (see Figure 4.2).

The manager maintains a 'discovered' list of active MASs. In particular, at startup time the *Network Discovery* thread parses a text file ("*network configuration*" file), which includes a list of the managed devices' IP addresses and information regarding the subnets they are physically connected to; it then 'pings' the corresponding hosts checking whether there is an active MAS server thereon. It is noted that active servers are discovered either at manager initialisation or whenever a new MAS starts operation. In the latter case, the manager application is notified and the host name appended to that manager's 'discovered' list. Clearly, the discovery of active agent servers based on parsing a text file is not sufficiently flexible and dynamic (it does not adequately deal with topology changes), however the design and implementation of a topology discovery algorithm was beyond the scope of this research work. The implementation of an automated naming service is not a trivial task, as it requires

configuration information obtained from various sources. That includes the Management Information Bases (MIB) of routers, bridges and switches, use of Internet Control Message Protocol (ICMP) messages, the Domain Name Service (DNS), etc. [LIN99].

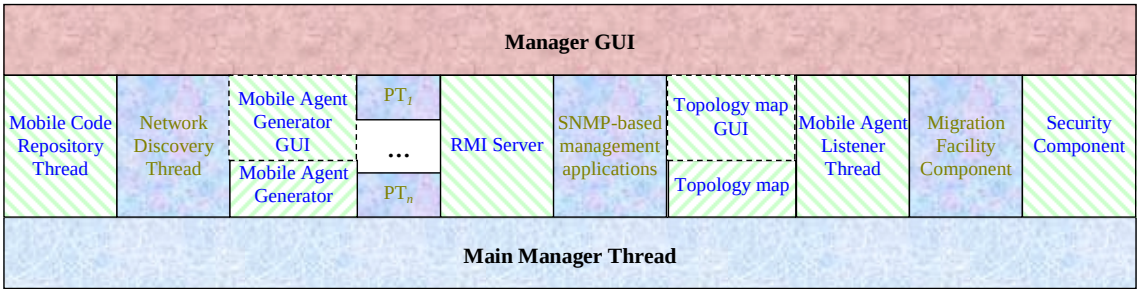


Figure 4.2. Break-down of the components that compose the manager application

In the event of a pending MA-based task, the manager retrieves the MA's definition from a Mobile Code Repository (MCR), it instantiates an MA object and assigns it an itinerary including all active agents hosts, unless a travel plan is manually specified. These actions are basically performed by one of the *Polling Threads*, whose operation is described in Section 5.2.1.1. The MA's state information is then compressed (using the Java *gzip* utility) and transferred through the Migration Facility Component (MFC) to the first destination host. When the MA returns back, it is received by the Mobile Agent Listener (MAL) thread, which retrieves the results carried by the MA and presents them to the user. The MAL and MFC components are basically identical to the ones used by the MAS entities; their implementation details are discussed in Sections 4.4.1.3.1 and 4.4.1.3.5, respectively.

The manager application is equipped with a user-friendly Graphical User Interface (GUI) including a MIB browser, 'event' and 'results' panels, a pull-down menu, several action buttons, etc (see Figure 4.3). The Manager's GUI also displays time statistics regarding the response time of completed management tasks; these statistics can optionally be written in a text file (this action is controlled through the GUI's menu), in a format easily extracted and inserted in a spreadsheet for further processing and analysis. Among the possible future extensions of the framework, is its integration with the JDBC[1] driver to allow automatic logging of operational results in a database that could be later used to generate high-level reports or graphs. Another direction of future work will include the design and implementation of an API used for developing manager applications, which will offer the manager GUI as an optional feature.

[1] The Java Database Connectivity (JDBC) [JDBC] kit allows Java programs to connect to any commercial relational database (given that it provides JDBC drivers) and query or update it using the industry standard query language (SQL). In the NSM context, JDBC makes it easy for Java-enabled management platforms to store or retrieve management data from databases and possibly present it to the administrator in a user-friendly manner.

Currently, additional facilities allow:

- polling of agents for specific object values; the fluctuation of arithmetic values may optionally be graphically illustrated using Java-enabled graphs updated in real-time (see Figure 4.4);
- initialisation of automated network discovery processes;
- acquisition of on-line MIB variable descriptions (see Figure 4.5);
- initialisation of MA-based or SNMP management operations. MA-based operations will be described in detail in the following sections. SNMP operations have been supported in order to compare their performance against MA-based solutions. These operations include simple ‘get’, ‘get-next’ and ‘set’ requests, retrieval of SNMP tables and periodic, synchronous/asynchronous polling of an arbitrary number of NEs for a set of MIB objects.

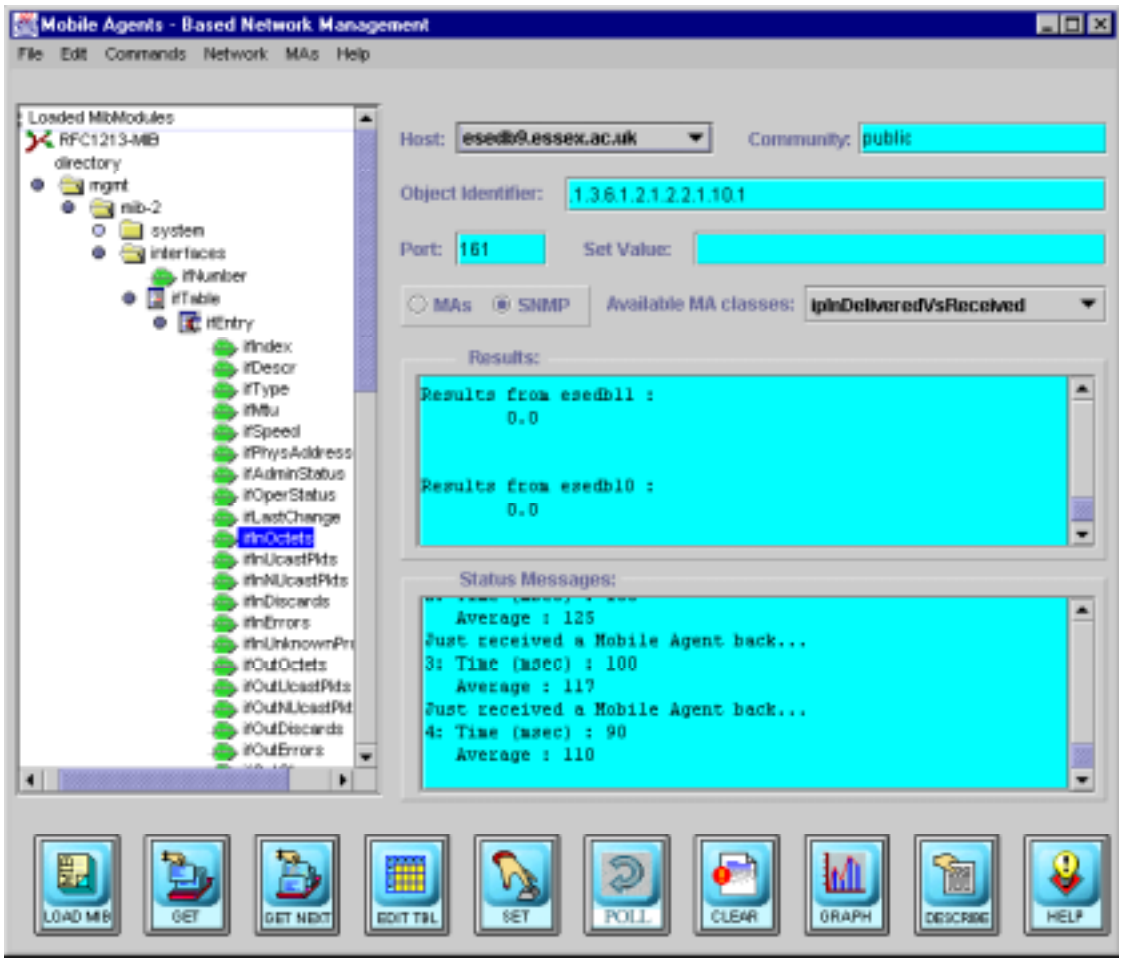


Figure 4.3. The manager application Graphical User Interface

Several classes of the Java-based AdvnetNet SNMP package [AdventNet] have been utilised to built high-level SNMP applications and also as a basis of the real-time line graph and MIB browser components.

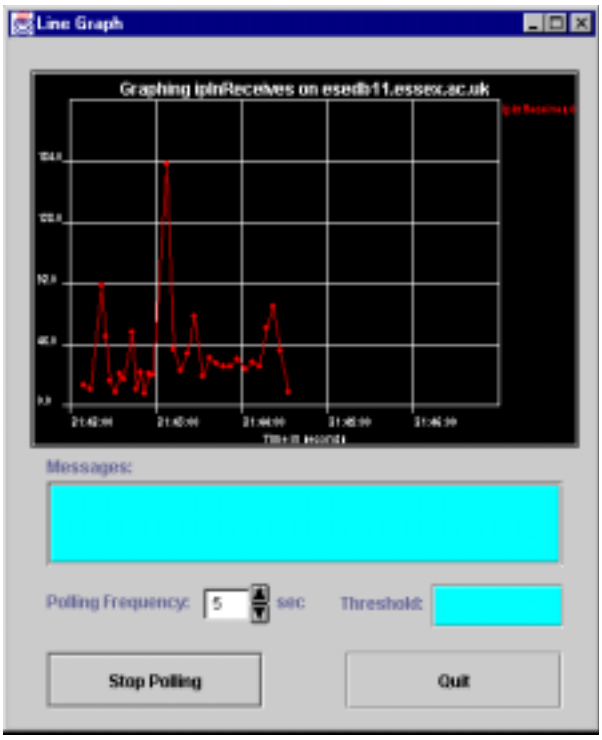


Figure 4.4. Polling of a MIB variable

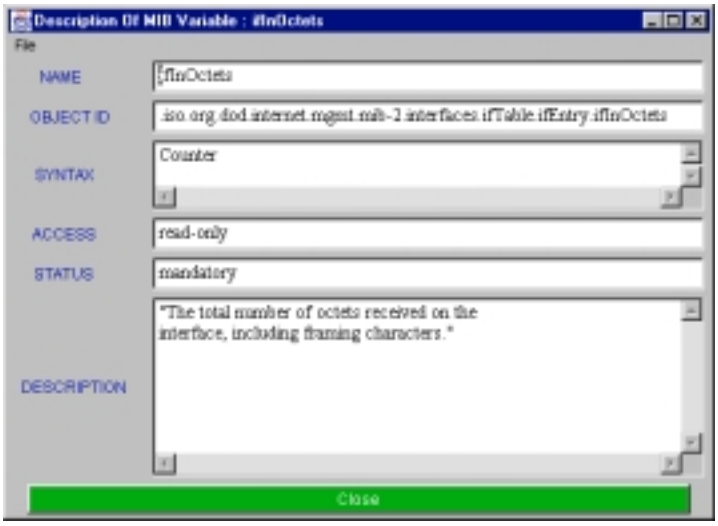


Figure 4.5. On-line acquisition of a MIB object description

The Manager application also starts and controls an RMI server; the role of this component will be discussed in Sections 5.2.2.1 and 6.3.2.

4.4.1.2. Mobile Agent Implementation

An MA object is identified by its code (description of its behaviour), state information (modifiable variables) and attributes (static/permanent information). In the context of management-related applications, MAs are Java classes supplied with:

- a unique ID string, which serves for distinguishing an MA object from others (the development of a naming architecture was beyond the scope of this work, therefore, the ID string is simple structured in a <origin_host:serial_number> format);
- an 'itinerary folder', including a list of hosts to be visited during the MA's operational travel; the itinerary may either automatically built or manually specified by the user (see Figure 4.6).
- a 'data folder', used to store collected data;
- a 'problems folder', used to report faults;
- a flag denoting the transport protocol used for the MA's transfers;
- a flag indicating whether encapsulated data should be encrypted or not;
- a byte array containing the MA's 'signature', used for authenticating the MA when it arrives at a destination host;

Itinerary, data and problem folders have been implemented as Java *vectors*, i.e. dynamic arrays (`java.util.Vector` class). At each visited NE, a data sample is typically appended to the data folder whereas the local NE address is removed from the itinerary vector. In this way (i.e. by reducing the itinerary vector size) and by applying data aggregation methods, we prevent the MAs state from growing too rapidly. MAs state is compressed (using the Java *gzip* utility) before their transfer to the next host to minimise communication overhead. In addition, the transfer of MAs over both TCP and UDP transport protocols has been implemented.

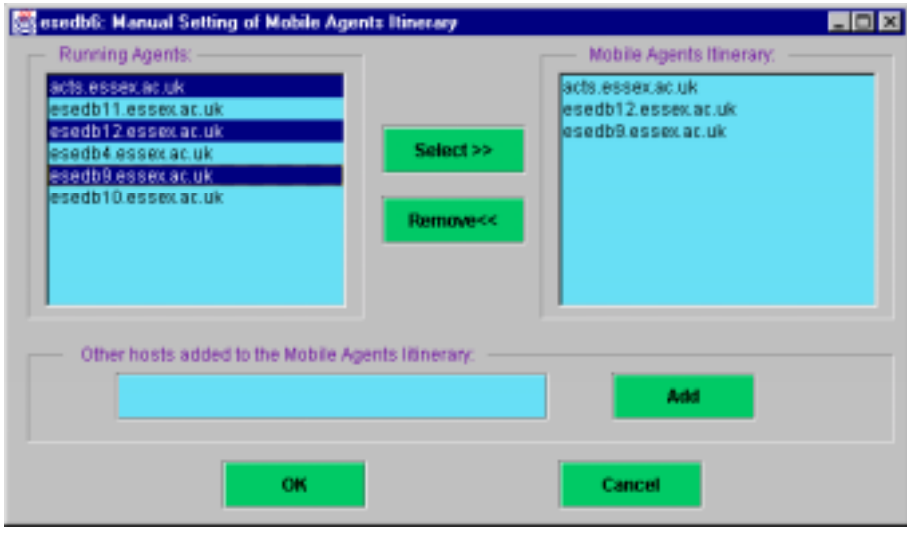


Figure 4.6. Manual setting of a Mobile Agent's itinerary

MA classes also include a number of methods that facilitate the interaction with polled devices. The MA code is minimised in order to reduce bandwidth requirements. Through the

process of *serialisation*, the state of an MA object can be saved, transferred through the network and reconstructed (de-serialised) at the receiving node (see Section 3.5.2).

In order to improve design flexibility, maximise code reuse and accelerate prototype development, an MA 'superclass' has been implemented, providing the root attributes listed above, which are essential for any MA object. The `MCode.MA` superclass implements the `java.io.Serializable` interface[2], allowing instances of the class to be serialised/de-serialised. Any class intended to employ mobility characteristics should necessarily sub-class the MA superclass. The methods of the MA superclass are shown in Figure 4.7.

```
public final void setParams(String ID, String creator, Vector itinerary, boolean TCP, String type,
                boolean encryption, byte[] sign){}
/* This method sets the MA's parameters; it can be invoked ONLY by the MA's creator (manager),
Otherwise, an Not_Authorised_To_Initialise_Exception exception is thrown. */

public void run(){}   // The main execution block for this MA
public void provideHandle (MAServer MaServer){}  // Binds the MA object to the local MAS entity
public final String getID(){} // Returns the MA's ID
public final String getCreator(){}  // Returns the name of the host where the MA was originally created
public final boolean getTransmissionProtocol(){}   /* Indicates the transport protocol used for the MA
                                                      transfers (true for TCP, false for UDP) */
public final Vector getData(){}  // Returns the data collected by the MA
public final byte[] getSignature(){}  // Returns the "signature" of the MA (used for authentication)
protected final void encapsulateData (Object value){}    /* Inserts a new sample of data into the data
                                                            vector */
public final Vector getProblemsFolder(){}  /* Returns the names of the hosts that could not be visited
                                              during the MA's travel */
public final InetAddress getNextHost(){} // Returns the next host included into the MA's itinerary
public void onStart(){}   // Invoked when the MA is first instantiated
public void onArriving(){}  // Invoked when the MA arrives at a host
public void onMoving(){}    // Invoked when the MA is about to be launched
public void onFailedMoving(String host) {}  /*   Invoked when the MA is not able to migrate to a
                                                 specific host */
public void onStop(){}   // Invoked when the MA thread's execution is about to stop
public void onSuspend(){}   // Invoked when the MA thread's execution is about to be suspended
public void onResume(){}  // Invoked when the MA thread's execution is about to be resumed
```

Figure 4.7. The methods of the MA super-class

The `setParams()` method is invoked by the manager application to set the MA's basic parameters (e.g. ID, itinerary, etc). To protect MAs against tampering, sensitive MA properties may be specified only *once*, when the MA is created. If a malicious host attempts to modify these properties, a `Not_Authorised_To_Initialise_Exception` exception is thrown. There are also methods used by the MA itself, for instance to obtain the name of the next host to be visited or request migration. Another set of methods is used to facilitate the interaction with visited hosts, for example to bind the MA to the local MAS server, encapsulate a sample of data into its data folder, return the MA's ID, signature, data, etc. A last set of

[2] A Java class cannot be serialised unless it implements the `java.io.Serializable` interface, or it extends another class implementing the `java.io.Serializable` interface.

methods are automatically invoked when the MA object is instantiated, arrives or moves from a host, fails to migrate, or when its execution is started, stopped, resumed or suspended.

In addition, we have implemented several service-oriented classes (discussed in Chapter 7) that extend the MA superclass. These in turn, are sub-classed by MA classes created by the MAG tool. This flexible hierarchical approach minimises MA bytecode and eases the creation of service-specialised MAs.

4.4.1.3. The Mobile Agent Server: Interface to Managed Resources

The interface between visiting MAs and legacy systems is implemented through MAS modules (Figure 4.8); we assume that a MAS server is installed on every managed device. Functionally, the MASs reside above standard SNMP agents, defining an efficient run-time environment for receiving, instantiating, executing, and dispatching incoming MA objects, whilst protecting the system against malicious agents attacks. The SNMP agent process is started automatically at MAS initialisation, if not already active. The port in which the SNMP agent listens for incoming requests can be specified by the user in the command line, otherwise the default UDP port 161 is used. In Windows platforms, the standard *SNMP service* provided with the operating system is used. In UNIX stations, the *snmpd* daemon that comes with the *UCD-SNMP* package [UCD-SNMP] has been utilised. This package is originally based on the earlier CMU-SNMP implementation, but has been greatly enhanced, ported and fixed so that it supports, among others, Solaris, Linux and HP-UX platforms.

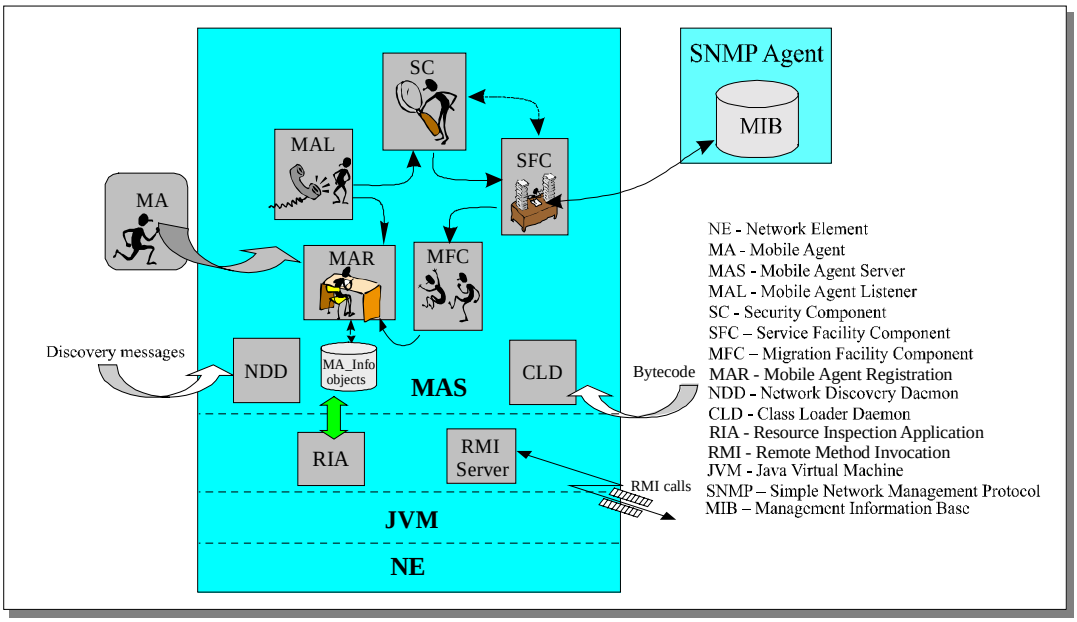


Figure 4.8. The Mobile Agent Server

Similarly to the manager application, the MAS class is designed so as to compose a multi-threaded environment. Apart from increasing the framework's modularity, this scheme allows

independent components to work concurrently and therefore decrease the overall delay when handling MA objects, especially when the rate of incoming MAs is relatively high or bursty. For instance, a MAS may receive or authenticate an MA at the same time that a second MA is dispatched to another host. The main building blocks of a MAS server are illustrated in Figure 4.8 with their implementation detailed in the following sections.

4.4.1.3.1. Mobile Agent Listener (MAL)

The MAL is a daemon, which basically includes two separate threads: the first (`MAS.TcpListener` class) listens for incoming MAs on a well-known TCP port, with the second (`MAS.UdpListener` class) listening on an advertised UDP port. To illustrate, let us focus on TcpListener's operation. TcpListener opens a TCP socket and blocks itself on listening mode waiting for incoming MAs. Upon the arrival of an MA, its state is decompressed and de-serialised. The MA is then authenticated by the Security Component (SC). If the authentication is successful (the MA has been created by a trusted host), the MA is first registered by the Mobile Agent Register (MAR) component. A separate thread is subsequently created for the MA's execution and assigned the *maximum possible priority* (`Thread.MAX_PRIORITY`). That way, MA threads are always prioritised against other pending executing threads on the local device, ensuring timely execution of their monitoring tasks.

TcpListener then binds the MA object to the local MAS server (it provides it a reference of the MAS class instance) to enable the interaction of the two parties (allow the MA to invoke MAS methods and vice-versa). Last, the MA's execution is started (its `run()` method is called). From that point onwards, the execution of TcpListener and MA threads is detached, with the first returning back to listening mode. The process of decompression and de-serialisation is carried out by the `receive()` method of the `TcpListener` class. The latter is called by its `run()` method which also controls the MA's authentication and registration process and starts its execution. The implementation of these two methods is presented in Figure 4.9. UdpListener's operation only differs on that compressed and serialised data are retrieved from a byte array, which forms the payload of a received UDP packet.

```
 // This is the main execution block
public void run() {
    while (true) {
        MA ma = receive();   // Receive (de-serialise) the MA object

        if (ma != null){
            // Authenticate the received MA
            if (sc.Authentication(ma)) {
                // Create a new thread for the MA's execution
                Thread MaThread = new Thread(mas.MAsThreadGroup, ma, "MA");
                // Asign the MA object the maximum priority
                 MaThread.setPriority(Thread.MAX_PRIORITY);
                 ma.provideHandle(mas); // Bind the received MA to the local MAS server
                 MaThread.start();  // Start the MA's thread execution
                 // Ask the Mobile Agent Register to register the received MA
                 mar.keepMaReport(ma, MaThread);
            }
        }
    }
}

// ---------------------------------------------------------------------

// Receives and de-serialises an MA object
MA receive() {
    // Establish a TCP socket that can queue up to 3 MA objects
    ServerSocket serSock = new ServerSocket(MA_TcpPort, 3);
    Socket connection = null;
    MasObjectInputStream = null;

    // Block to listening mode waiting for incoming MA objects
    try {
        connection = serSock.accept();
        connection.setTcpNoDelay(true);  // Do not use Nagel's algorithm

         // Obtain a reference to the socket's data input stream
        InputStream is = connection.getInputStream();
           /*  Redirect the input stream into a zip stream and then into an MAs' de-
                                        serialisation stream. */
        dataIn = new MAsObjectInputStream
                              (new GZIPInputStream(new BufferedInputStream(is)));

         // De-compress and se-serialise the MA's state
        try {
            ma = (MA)dataIn.readObject();
        } catch (ClassNotFoundException exc) {
             System.err.println("Problem in deserialisation: " + exc.toString());
        }
        connection.close();
        return ma;
    } catch (IOException e) {
        try {serSock.close(); connection.close();}
        catch (Exception e1) {
            System.err.println("Socket has been already closed");
        }
      System.err.println("Exception while listening for MA connections : " + e);
    }
}
```

Figure 4.9. TcpListener's `run()` and `receive()` methods

4.4.1.3.2. Security Component (SC)

This component acts as the system's protective barrier. Specifically, the SC verifies the authenticity of the received MA through the use of security keys[3], ensuring that only trusted agents, dispatched by authorised hosts, are instantiated. In addition, it authorises the actions performed by the locally executing MAs and insures the privacy of sensitive management data returned to the manager station. The RSA (Rivest-Shamir-Adleman) algorithm [RIV78], based

[3] *Keys* represent secret information used by cryptographic algorithms. In traditional cryptography, the sender and receiver of a message know and use the same secret key (symmetric cryptography); the sender uses the secret key to encrypt the message, and the receiver uses the same secret key to decrypt the message. *Public-private* key algorithms use a different key for encryption and decryption. The generation, transmission and storage of keys are termed *key management*.

on the '*public-private pair of keys'* paradigm, has been implemented providing both *authentication*[4] and *encryption*[5] features. The reasons for choosing RSA rather than other authentication (e.g. SHA-1, DSA, MD5) or encryption (e.g. DES) algorithms was that the design and implementations of RSA authentication and encryption algorithms were very similar; that characteristic helped to maximise code re-use and enabled rapid prototype development. RSA's security strength is related to the assumption that detection of the private key (needed to sign or decrypt the transmitted data) is highly improbable. In particular, the security of RSA relies on the difficulty of factoring large integers (dramatic advances in factoring large integers would make RSA vulnerable). In the current prototype, the encryption and authentication algorithm implementations are included in the `Security.Security` class, whose method definitions are shown in Figure 4.10.

```
// Encrypts the original message
public static byte[] Encryption(String message);
// Decrypts the encrypted message
public static byte[] Decryption(byte[] encrMsg);
// Generates a digital signature of the original message
public static byte[] Signature(String message);
// Checks whether the original message is recovered after processing its sugnature
public static boolean Authentication(byte[] signature, byte[] original);
```

Figure 4.10. The method definitions of the `Security` class

Thus, the steps followed during an MA object round-trip are:

- The manager application signs the MA using the public key. The digital signature generated is stored in a byte array, encapsulated and sent along with the MA.
- Upon arrival, the SC executes the authentication function in conjunction with the private key and verifies that the MA state has not be tampered (this ensures data integrity). If the authentication is successful the MA's execution is initiated, otherwise the manager is informed accordingly.
- The data encapsulated into the MA's state are first encrypted using the public key, given that data encryption has been requested by the administrator at the MA's creation time.
- The manager decrypts the data received by invoking the decryption method, which makes use of the private key.

[4] *Authentication* is defined as the action of verifying information in terms of identity or ownership.

[5] *Encryption* is the transformation of data into a form that is as close to impossible to read without the appropriate knowledge (a key). Its purpose is to ensure privacy by keeping information hidden from anyone for whom it is not intended, even those who have access to the encrypted data. *Decryption* is the reverse of encryption, i.e. the transformation of encrypted data back into an intelligible form.

The integration of the SC with the Java Cryptography Architecture (JCA) [JCA], which refers to a generic framework for accessing and developing cryptographic functionality for the Java Platform, will be considered in future extensions of the framework. JCA encompasses the parts of the JDK 1.2 Security API related to cryptography, as well as a set of conventions and specifications. However, only authentication algorithm implementations are provided in the current release. JCA also introduces a *provider* architecture [Providers] that allows for multiple and interoperable cryptography implementations. Implementation interoperability means that various implementations can work with each other, use each other's keys, or verify each other's signatures.

Authorisation: The security of MAS entities has been strengthened by introducing *authorisation* features that restrict the authority domain of visiting MAs on legacy systems. Specifically, the Java standard *Security Manager* (SM), (`java.lang.SecurityManager` class), has been extended to prevent MAs from directly reading/writing files, creating CLs or sub-processes, shutting down the MAS application, etc. The identification of illegal actions is achieved through registering the incoming MA threads to a given *thread group*, i.e. a batch of threads that eases the manipulation of active MAs. Based on the fact that an MA cannot change the thread group it belongs to, whenever a malicious action is detected, the SM checks the thread group of the action's originating thread; if this thread belongs to the MAs thread group, the action is not permitted.

An issue not addressed by the current implementation of the SM is that MAs cannot be restricted from entering an endless loop (consuming all CPU cycles), or creating thousands of new threads. However, in such an emergency situation, the SM could selectively destroy all threads not belonging to the MAS components' thread group. Alternatively, a resource accounting interface like *Jres*[6] [CZA98], a scheme for controlling allocation of different kinds of resources (CPU, memory, etc) within the Java runtime system, could be used.

In general, the implementation of an impenetrable security shield, has not been a principal aim during the development of our framework. Besides, agent security is a particularly complex subject and represents an active and evolving research area [VIG98]. However, with the security precautions described in this section, we consider the network devices to be reasonably safe from malicious MA attacks.

[6] Jres allows per-thread accounting of heap memory, CPU time usage and the number of bytes sent and received through a network connection. In addition to tracking resource consumption, tasks can be informed when new threads are created. Another interesting feature of Jres interface is that it provides mechanisms for setting limits to resources available to individual threads and associates an overuse callback method that is called whenever this limit is about to be exceeded [CZA98].

4.4.1.3.3. Mobile Agent Register (MAR)

MAS entities keep active control of the MA objects executing on their local devices. In particular, the MAR component maintains a hashtable[7] (`java.util.Hashtable`), using MA IDs as a primary key and including a list of "*MA_Info*" objects, each mapped to an MA object running on the local host. MA_Info objects are basically records that include a reference to the MA object they correspond to (allowing the MAS server to perform a number of actions upon it, i.e. invoke its methods) and additional information related to the MA, such as the MA's class name, the execution frequency of its management task, its arrival time, its execution status (activated, de-activated, suspended), etc. The hashtable is dynamically updated whenever an MA arrives or leaves the host. Specifically, the MAL thread inserts a new record when receiving an incoming MA, while the MFC thread deletes the corresponding record as soon as the MA object migrates to another host.

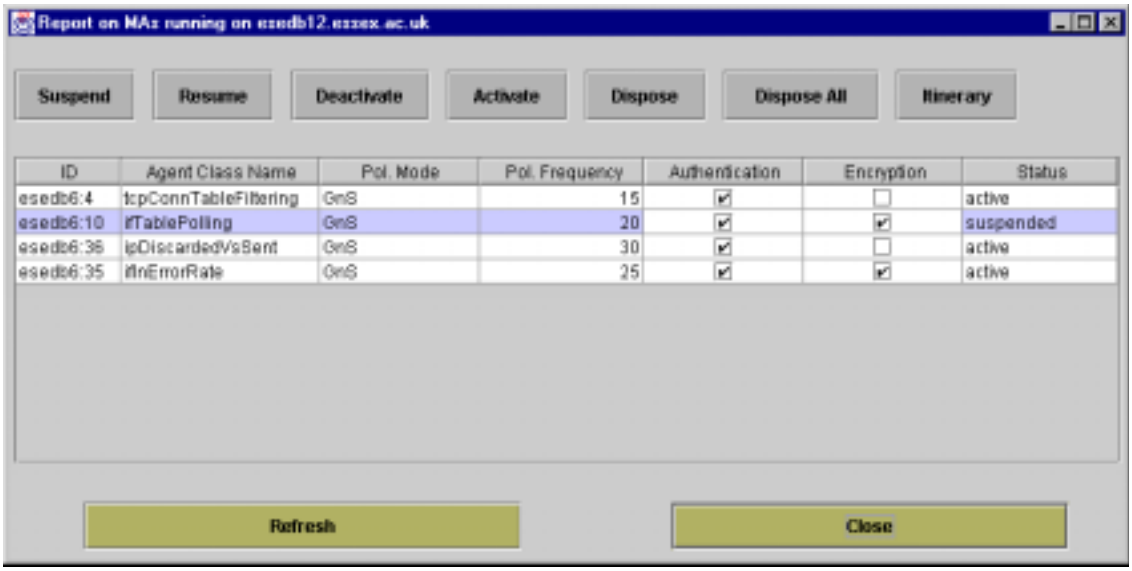


Figure 4.11. On-line visual profiling of Mobile Agent objects executing on a network device

The information stored within the hashtable can be remotely retrieved through an RMI call (the `getMAsReport()` method of the MAS's RMI server is invoked and a vector of MA_Info objects is returned). A report about the MAs executing on any remote device is then presented to the network administrator through the GUI shown in Figure 4.11. The administrator may then select any of the MA instances presented in this GUI and perform a number of actions upon them. In particular, he/she may suspend, resume, activate an MA or modify its itinerary; these actions are carried out through RMI calls (see Figure 4.14) that cause the automatic update of the remote hashtable and the invocation of a corresponding MA's method.

4.4.1.3.4. Service Facilitator (SF)

Upon successful authentication, the MA is activated and provided a handle to the Service Facilitator (SF) component, which serves as an interface between MAs and services offered to them. SF generally includes the 'know-how' of the services offered to incoming MAs, i.e. all the functionality needed by the MA objects to perform their decentralised management tasks. An alternative approach would be to integrate this functionality within the MA classes themselves; however, this would result in the unnecessary transfers of large pieces of code that would seriously overload network resources.

At its current implementation, the SF component is solely oriented to NSM applications; as such, it basically offers a library of methods that facilitate the interaction of MA objects with the local SNMP agent. In particular, it allows the MAs to perform from simple SNMP requests ('get', 'get-next', 'set', etc) to relatively complex tasks (obtain an SNMP table, a specific column or row of a table, etc). The Java-based SNMP package of AdventNet [AdventNet] has been used as a basis to construct the complex management applications mentioned above. For instance, the 'get-table' operation is implemented as a succession of 'get-next' requests, until all the values of an SNMP table are retrieved. Currently, only SNMPv1 is supported, mainly due to its simplicity. Although the first release of the SNMP protocol is known to exhibit severe security limitations, that does not represent a problem in this case as SNMP requests/ responses are not transmitted through the network, but locally exchanged.

In a typical scenario, an MA passes an arbitrary number of object OID strings to an SF method, which performs the SNMP query and returns the requested values. These values are subsequently processed, if necessary, by an automatically invoked MA method. The value acquired, either directly by the system or as a result of computation, is passed to the SC sub-system, encrypted and encapsulated into the MA's state.

4.4.1.3.5. Migration Facility Component (MFC)

The role of the MFC is to dispatch upon request an MA object to a specific network device. An MA transfer may be requested either by the MA itself or any other thread that has a reference to the MA object and the MFC component. MA transfers are triggered through calling the `move()` method of the MFC. The latter will invoke the `moveTcp()` or `moveUdp()` method, depending on which transport protocol is used for the MA transfers (see Figure 4.12).

---

[7] Hashtables can be thought of as key-indexed arrays that enable efficient search of information.

The MFC component is part of both the MAS servers and the manager application. In essence, its operation is the inverse of the MAL's operation. Namely, in the case that TCP is used for MA transfers, a connection is established, the MA state is compressed (`java.util.zip.GZIPOutputStream` class) and then serialised (`java.io.ObjectOutputStream` class) with the resulted byte stream directed into the TCP connection output stream. When the data transfer is completed, the network connection is released. In case that UDP is used as transport protocol, the procedure described above differs only on that the compressed/serialised data are stored in a byte array and packaged within one (or more) UDP packet(s) prior to their transmission (no connection needs to be established/released). Following the MA's migration, the MFC deletes the MA_Info object associated with to the dispatched MA by invoking the corresponding method of the MAR component.

```
/* Moves the MA object to the requested IP address. The returnToManager boolean shows whether the
operational travel of the MA has ended (the MA should return to the manager) while the tcp flag
denotes the transport protocol used for the MA transfers (true for TCP, false for UDP) */
public void move(MA ma, InetAddress addr, boolean returnToManager, boolean tcp);
// Invoked by the move() method should the transport protocol used is TCP
public void moveTcp (MA ma, InetAddress addr, boolean returnToManager);
// Invoked by the move() method should the transport protocol used is UDP
public void moveUdp (MA ma, InetAddress addr, boolean returnToManager);
```

Figure 4.12. The method definitions of the `MigrationFacilityComponent` class

The life cycle of an MA object and its interaction with the visited MAS servers is illustrated in the block diagram of Figure 4.13.

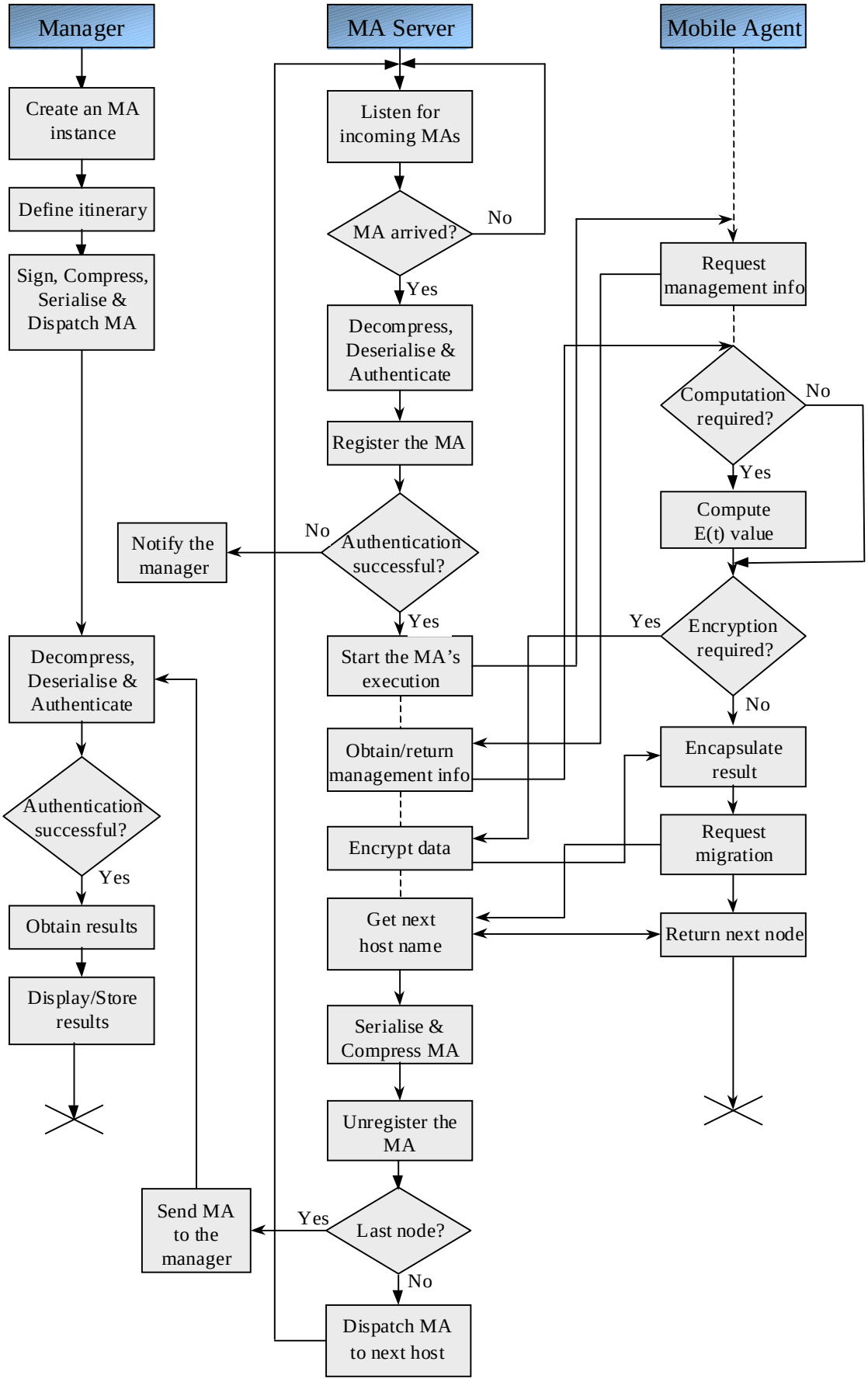


Figure 4.13: Flow diagram of a Mobile Agent's life cycle

4.4.1.3.6. Mobile Agent Server RMI Server

The RMI server is implemented as a separate thread, which can optionally be instantiated by the main MAS thread (this is indicated by the user in the command line that starts the MAS application). The reasons that the RMI server presence is not compulsory is that it does not directly interact with incoming MAs while it represents additional overhead on the local system resources (mainly because of the `rmiregistry` process). The role of the RMI server is to enable remote interaction of the manager with distributed MAS entities and, hence, allow the administrator to obtain reports regarding the CPU and memory load profile of network devices or the number and type of MA objects currently executing on them. It also allows the administrator to perform a number of actions upon the MAs, such as to change their execution status, modify their itinerary, etc (see Figure 4.14).

```
public Vector getResourcesReport (ReportDescription descr) {}  /* Returns a report on the local
                                                                  resources (CPU & memory)  usage */
public Vector getMAsReport () {}   /* Returns a report about the MAs currently executing on
                                      the local device */
public boolean suspendMA (String id) {}  // Suspends the execution of the MA with ID = id
public boolean resumeMA (String id) {}  // Resumes the execution of the MA with ID = id
public boolean activateMA (String id) {}  // Activates the MA with ID = id
public boolean deactivateMA (String id) {}  // De-activates the MA with ID = id
public boolean disposeMA (String id) {}  // Disposes the MA with ID = id
public boolean disposeAll () {}  // Disposes all running MAs
public boolean modifyItininerary (String id, Vector itinerary) {}      /* Modifies the itinerary of the MA
                                                                          with  ID = id. */
```

Figure 4.14. The methods of the MAS's RMI server class

4.4.1.3.7. Resource Inspection Application (RIA)

The RIA is an application developed in C programming language, which runs outside the boundary of the MAS server. Its purpose is to monitor the usage of local resources in terms of CPU and memory load. RIA is linked to the MAS application via the Java Native Interface (JNI) that enables the inter-operation of Java applications with programs written in other languages (see Section 2.7.1.1). That is, JNI allows RIA to invoke certain MAS methods and vice versa. The motivation behind RIA's development has been to allow the administrator but primarily the MA objects to obtain information regarding network devices load. This information is exploited (see Section 6.3.6) to provide even distribution of processing and memory load among distributed agent servers.

4.4.1.3.8. Network Discovery Daemon (NDD)

The NDD is another thread instantiated by the main MAS thread, whose purpose is to discover the station where the manager application runs, while also making the local host 'visible' to the manager. Its operation is essentially very similar to the Network Discovery

thread of the manager application (see Section 4.4.1.1). In particular, upon startup the NDD parses a text file including a list of the IP addresses of devices where manager applications are likely to run and 'pings' the corresponding hosts, checking whether there is an active manager thereon. When the manager receives the message, it appends the MAS's hosting device to its 'discovered' list and notifies the NDD accordingly.

4.4.1.3.9. Class Loader Daemon (CLD)

The CLD is a daemon controlled by the MAS thread, whose role is to wait for MA class definitions sent by the manager application at the time that a new management service is introduced. The received MA classes are stored in a designated place (directory). CLD's role is discussed in the following section, while the class loading mechanism used by our framework is described in detail in Section 4.4.3.

4.4.1.4. Mobile Agent Generator (MAG)

The MAG, is essentially a factory for constructing customised MAs. In the context of this thesis, generated MAs are designed to poll static management agents according to certain operational function requirements. A GUI, dedicated to the MAG tool, allows the operator to:

- assign a name to the MA;
- specify the generic type of service this MA is intended to carry out;
- define the MA's functional requirements, i.e. determine its operational behaviour;
- set the polling frequency to determine how often instances of the constructed MA will be launched;
- specify the transport protocol (either TCP or UDP) to be utilised for the MA transfers;
- determine whether the data collected by the MA are to be encrypted and the MA itself authenticated;
- specify the class of network devices to be polled;
- optionally, define the MA's itinerary, i.e. the order in which the MA will visit the managed devices.

Options for editing the attributes, deleting or updating an existing MA instance are also available. Snapshots of the MAG GUI are presented in Chapter 7.

4.4.1.4.1. MAG tool operation

The MAG uses a skeleton Java source code with empty slots filled with the user-specified MA's properties. The Java code created is then compiled through an invocation of the `compile()` method of `sun.tools.javac.Main` class (this class is offered by Sun,

although not included in the standard JDK library) [SunTools]. The generated Java bytecode is subsequently compressed and transferred (multicasted) through TCP connections to all operating agent hosts. The MA's properties are compared, prior to its construction, against those of the existing MA classes to ensure that there is no other with the same functionality; in case such a class exists, the user is notified through a message.

On the agent side, the CLD receives and decompresses the transmitted bytecode, validates the included Java class and stores it in a designated space. MAG's functionality is illustrated in the block diagram of Figure 4.15.

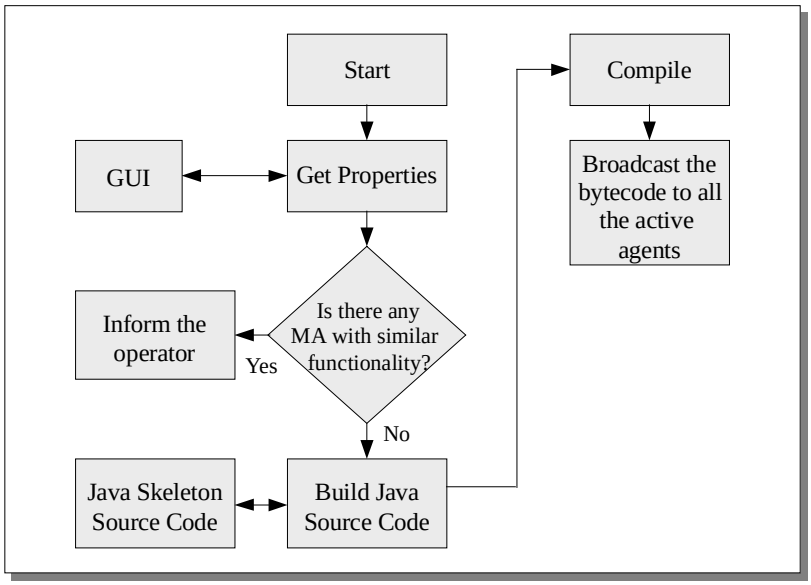


Figure 4.15: Mobile Agents Generator functional diagram

It should be emphasised that the transfer of the MA bytecode is performed only *once*, at the MA construction time. From that point onwards the transfer of persistent state, obtained from serialising the MA instance, is sufficient for MAS entities to recognise the incoming MA and recover its state. In contrast, most available MAPs apply a policy that requires the transfer of both the MA's bytecode and persistent state, resulting in higher demand on network resources. To illustrate, in our implementation the ratio (bytecode size):(state size) typically lies in the range 10:1 to 15:1.

It should be emphasised that MA code is not 'blindly' multicasted. For instance, in case that a monitoring task is intended to be performed upon a limited set of devices, it would not be meaningful to broadcast the bytecode of the MA to perform this task to every managed device, as that would create unnecessary overhead. Hence, the administrator can specify the set of devices to be monitored (through the GUI shown in Figure 4.16), with MA code being exclusively sent to them.

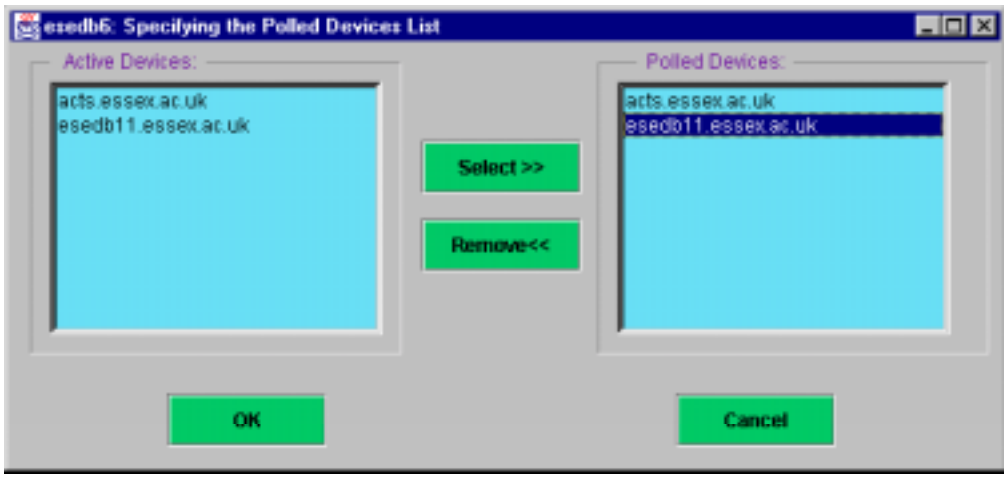


Figure 4.16. Specifying the devices to be monitored.

The functionality of MAs created by the MAG may only be an extension of limited generic service types. These types are designed as sub-classes of the MA 'super-class', specifying general patterns of MA-based NSM tasks. MAs constructed by the MAG tool extend one of these sub-classes, refining their functionality and defining service-specialised MAs. Examples of available generic NSM service types will be given in Chapter 7, that describes several MA-based network monitoring applications. An alternative approach for structuring MA-based applications would be to keep a unique MA structure (class), whose functionality would span a wide range of management operations. However, such an approach would result in the unnecessary transfer of dynamically growing code, much of which may only be occasionally executed.

4.4.1.4.2. Advantages of using the MAG tool

The focus of current research on MA-based management is on the development of execution platforms and applications for MAs. However, methodologies for building agents have received little attention. Creating MAs can be tedious and susceptible to errors and also requires programming skills and detailed knowledge of the MAP design. In particular, introducing a new management operation would comprise the following steps: (a) design and write the source code that defines the MA functionality; (b) compile the source code (significant amount of time is wasted for coding and debugging, where programming expertise is a prerequisite); (c) possibly modify and re-compile the core MAP component classes in order to make possible the instantiation of the new MA class; (d) possibly reboot the manager application to allow modifications to take effect; (e) reboot active MAS servers (unless their class loader is able to recognise that the MA classes have been modified).

To ‘condense’ this perplexing procedure in one sentence, we quote from [KIM98]: “If you want another management function, just write another mobile code and ship it again through the network”.

Therefore, the use of the MAG tool brings forth a number of advantages:

- The MAG ensures that the framework remains sufficiently flexible by enabling on-the-fly construction of service-oriented agents, without the need for reconfiguration, re-installation or re-instantiation of either the manager or the agent applications. The MAG functionality can easily be extended so as to cover a wider range of management tasks.
- Ease in introducing new management tasks in a user-friendly manner. The productivity of the development process is increased through reducing the time needed to develop an MA and improving reliability as a result of reusing proven components.
- The generation of the MA bytecode and its distribution among distributed managed hosts is automated and transparent to the user.
- The network administrator does not need to be aware of the implementation details behind the introduced management service or the management framework, nor to have programming experience.

Similar work has been recently reported in [GSC99], which describes the design and implementation of an agent construction toolkit, the AgentBean Development Kit (ADK), which is an extension of the Sun's Bean Development Kit (BDK).

4.4.2. Fault Tolerance: Tolerating Node Failures

A key aspect of our framework is its fault tolerance features. A MAP should be able to survive network and systems failures to secure MA migrations. In particular, the following two fault scenarios have been investigated:

Scenario 1: An MA should adapt to unexpected situations, such as the failure of a host MAL thread. In this case, if TCP is used for MA transfers, the TCP connection establishment fails (Figure 4.17a). The MA’s *onFailMigration*() method is then automatically invoked to record the unreachable host’s name into the MA’s *problem folder* and retrieve the next destination host from the itinerary vector (Figure 4.17b). The MA will then migrate to this host (Figure 4.17c). When returning to the manager host, the MA reports the failed devices to the manager application, which in turn will take any necessary regenerative actions.

If UDP is the transport protocol choice, the detection of a failed device is more complex as the UDP datagram carrying the MA’s state would simply be lost. A way to get around this problem would be to enforce the destination host to issue an acknowledgement when

successfully receiving an incoming MA. In case of a fault, the acknowledgement would never reach the originating node and after a given time interval the MA image would be transmitted to its next destination. However, that approach (proposed in [SAH98]) would create additional traffic load, counterbalancing the benefit of UDP lightweight nature.

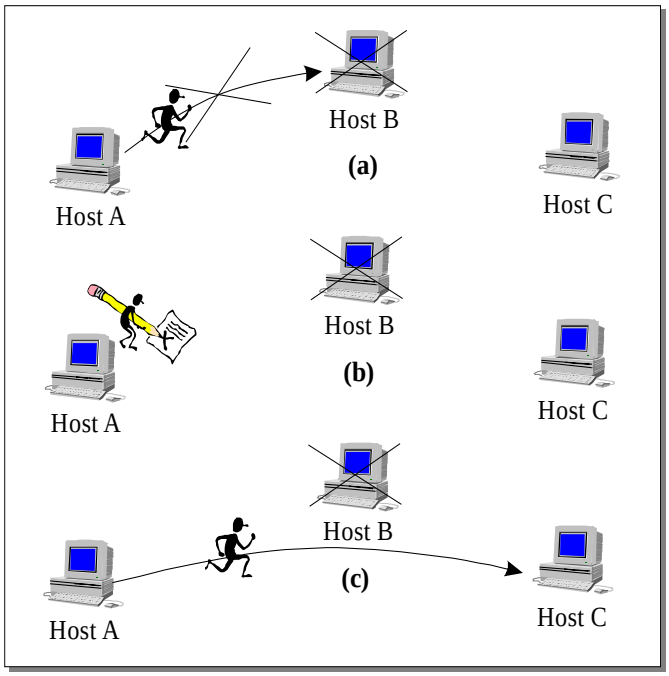


Figure 4.17. MA reaction to the detection of a failed MAL thread

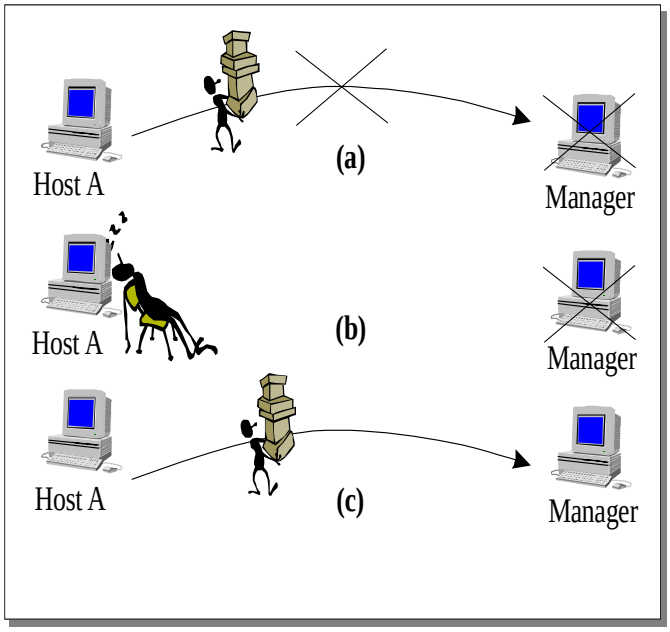


Figure 4.18. The specific case of MAL thread failure on the manager host

Scenario 2: A second scenario would be a fault in the MAL thread of the manager host itself (Figure 4.18a), which represents a special case. Since loss of management data carried by an MA should be avoided, an alternative approach to scenario 1 is employed. Upon detecting a

fault, the MA 'sleeps' for a given interval (Figure 4.18b) and then resumes execution to retry the connection. If the manager application recovers before a pre-determined number of retries elapses, the MA is transferred (Figure 4.18c), otherwise it is disposed of.

The two algorithms described above have been examined in [BRU01] (they are termed "Skip" and "Wait-and-Retry" respectively); the paper evaluates the algorithms through analytical models and simulation results. It is noted that the reliable operation of the described fault tolerance mechanism has been verified by creating 'virtual' failures. In particular, fault scenario 1 has been 'simulated' by instructing MAs to visit hosts that do not run a MAS server, while surviving fault scenario 2 has been confirmed by shutting down the manager application and restarting it, allowing travelling MAs to deliver their results.

4.4.3. Class Loading Mechanism

When an MA object arrives at a destination host, its bytecode is retrieved from the network stream, stored in the local disk and cashed within a hashtable of the default JVM's CL (`java.util.Hashtable`). Since memory access is typically much faster than disk access, the hashtable minimises the time needed to load the MA bytecode when another MA object of the same class visits the same host in the future (it is assumed that a class loaded once, is likely to be loaded again).

The default CL works fine as long as MAs' definitions remain unchanged; it is not able though to distinguish between two different versions of the same class, due to a limitation of the Java language that does not allow to define two classes with the same name within the same CL namespace [VEN98]. Hence, should an updated version of an MA visits a NE where the old class is already stored in the cache, the CL will load the cashed version, even if the updated class has been sent. This causes a problem when MA objects are being deserialised, as the serialised data of the new class are incompatible with those of the old class version. Most MAPs fail to cope with this MAs '*versioning*' problem and, as a result, the user is forced to 'reboot' all the MAS servers where the old version of the updated MA class is already stored to allow modifications to take effect. This solution is certainly not desired, especially for large-scale distributed systems where the hosting devices of MAS servers are geographically dispersed. The fact that MA classes carrying out management operations typically need to be modified/updated on a frequent basis, makes the need for developing a CL that efficiently deals with the versioning problem more pressing. The intelligent and lightweight class loading mechanism used by our framework represents one of its most distinctive features.

Among other requirements, the customised CL should be able to distinguish between two different versions of the same class, i.e. to force a class definition to be updated as it evolves,

while reducing the communication cost as well as moderating the usage of NEs computational resources. The Aglets MAP is the only known publicly available platform that deals with the versioning problem; it allocates different CLs to different sets of classes, with a new CL created for each updated version of an MA class [LAN98]. The distinction between the updated and older versions of an MA is achieved by sending information about the names and versions of classes along with the classes' bytecode. A similar approach is adopted in [ISM99] where a CL object is created for each MA class. However, that creates additional network overhead (class version information transferred on every MA migration), while frequent MA classes updates will trigger the creation of many CLs, with increased memory requirements.

A different approach is used in the MAP introduced in this thesis. In particular, a *customised* Mobile Agent ClassLoader (MACL) class has been designed, which extends the default (`java.lang.ClassLoader`) CL class and overrides its `loadClass()` method. The CLD thread of the MAS server maintains a list of the classes that have been already received; thus, when a class definition is received, the CLD checks the received classes list and should the same class has been received in the past, the MACL is forced to load the class definition from the local disk rather than from the cache.

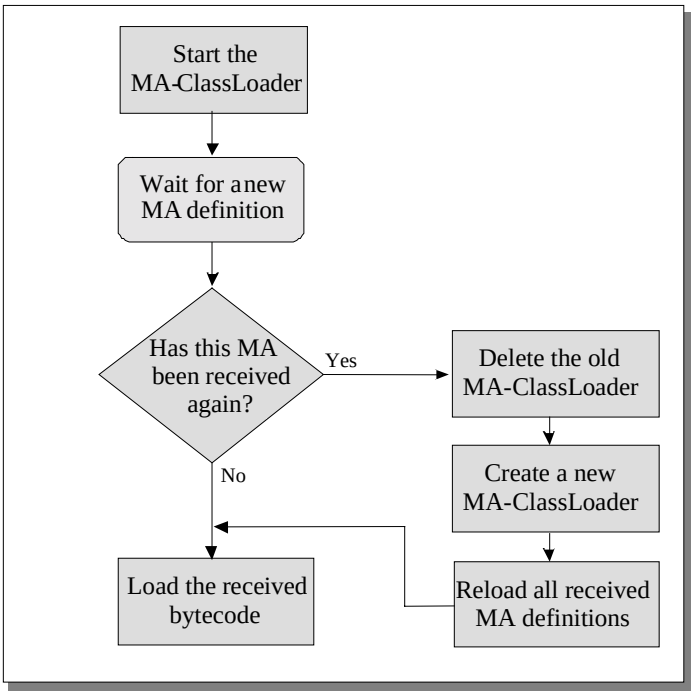


Figure 4.19. Class Loading mechanism block diagram

The main advantage of the proposed class loading mechanism is that a *single* MACL object is used, leading to minimal use of local resources. This object is "replaced", i.e. the old MACL object is garbage collected, with a new one taking its place whenever one of the existing MA definitions is modified. The new MACL object then loads the bytecode (definitions) of all the

existing MA classes in order to minimise the time needed to load these MAs when they first visit the NE (otherwise the bytecode would be loaded from the disk upon the MAs reception leading to increased delay). The "replacement" technique was necessary, as a single CL does not load the bytecode of a given class twice. The operation of the MACL is graphically depicted in the block diagram of Figure 4.19.

In order to force the MAL thread to load the incoming MA object using the customised MACL (and not the JVM's default CL), a new class (`MCode.MasObjectInputStream`) extending the `java.io.ObjectInputStream` class and overriding its `resolveClass()` method, has been implemented. Thus, by using the `resolveClass()` method of the `MAsObjectInputStream` class, the MAL thread instructs the customised MACL to load the latest version of the incoming MA definition.

The MACL design also caters for the improbable case that the definition of a visiting MA object is not found in the local disk space (potentially because the local device was not originally specified by the administrator among the ones to be monitored by the MA). In this case a `java.lang.ClassNotFoundException` is thrown, the MACL contacts (through RMI) the MCR thread of the manager application and 'downloads' the requested class file.

4.5. QUANTITATIVE EVALUATION & ASSESSMENT

A critical omission frequently noticed in research papers introducing new MA-based frameworks for NSM is the analytical evaluation of the performance issues arising when implementations of the corresponding models are used in real networking environments. With very few exceptions (e.g. [LIO99] that proposes detailed mathematical models for comparative analysis of various MA-based management solutions depending on the MA organisation and deployment patterns), most of these works claim improved performance over alternative approaches, without providing any proofs. There are also several works (e.g. [SAH98, ELD99, ZAP99]) that present simplistic quantitative evaluations, oriented to their respective presented prototypes.

Reference [BAL98] comprises an interesting theoretical investigation of the three Mobile Code paradigms (COD, REV and MAs). A general performance comparison among these approaches is also provided, that may serve as a reference point for related quantitative evaluations. In this section, we undertake a preliminary evaluation, in terms of bandwidth usage and response time, comparing the performance of SNMP against that of our framework. The results of this theoretical investigation will be contrasted to the empirical results presented in Section 4.6.

4.5.1. Response Time Evaluation

In this section, we model the response time measured when performing SNMP or MA-based polling operations. First, let us consider SNMP-based polling (see Figure 4.20a). Assuming that SNMP requests are broadcasted to all the polled devices with the responses collected in parallel (*asynchronous* polling), the overall time for centralised polling, will be:

$$T_{SNMP,a} = max\left(2\left(t_{del,i} + t_p\right) + t_a\right) + N * t_{comp}\text{, for } i = 1..N \qquad (4\text{-}1)$$

where N accounts for the number of agents being polled, $t_{del,i}$ is the average network latency between the manager and the i^{th} agent, t_p the processing time required for handling a single request/response message, t_a the time needed to access the system resources, and t_{comp} the computation time needed to process the collected data. The overall response time is determined by the maximum individual time needed to poll any of the polled devices (hence the *max* operator). Clearly, t_p and t_a depend on the number v of MIB values obtained from each host:

$$t_p = t'_p + (v-1)\Delta t_p \quad and \quad t_a = t'_a + (v-1)\Delta t_a \qquad (4\text{-}2)$$

where t'_p and t'_a are the time needed to process the packet or contact the SNMP agent respectively, when there is a single OID in the request packet varbind list; Δt_p and Δt_a are the extra time needed to process and retrieve each extra requested object, respectively.

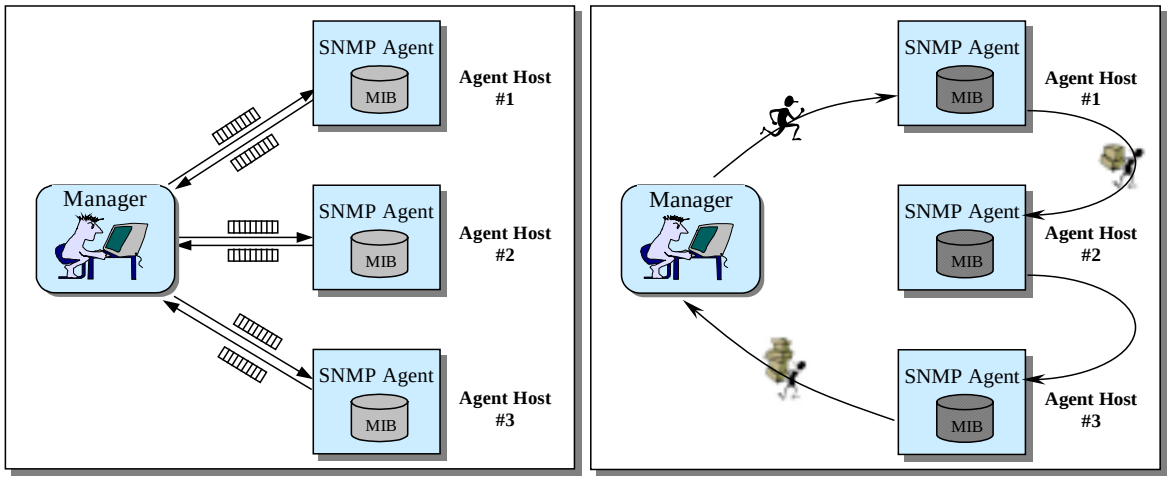


Figure 4.20: Centralised (SNMP-based) vs. MA-based polling

In case that the manager polls the agents sequentially, waiting for the response before sending the request to the next agent (*synchronous* polling), the overall response time is given by:

$$T_{SNMP,s} = \sum_{i=1}^{N}\left(2\left(t_{del,i} + t_p\right) + t_a + t_{comp}\right) \qquad (4\text{-}3)$$

Regarding MA-based polling, we consider the simplest case where a multi-hop MA sequentially visits every network device (Figure 4.20b). In this case, the response time is modelled as follows:

$$T_{MAs} = t_{cr} + (N+1) * (t_{del} + t_{s/d}) + N * (t_a + t_{comp}) \quad (4\text{-}4)$$

where t_{cr} is the time needed to create and instantiate an MA object, t_{del} the average network latency between the manager and the agent or any pair of agents, and $t_{s/d}$ the time taken to serialise/de-serialise an MA. T_{MAs} is defined as the time needed to create an MA object, plus the total transition time to visit each device and return to the manager (hence the factor N+1), plus the time needed to access and process the management information locally. Appendix B proposes programming techniques to accelerate MA migrations, thereby reducing MA-based polling response time.

Clearly, although useful, the modelling of response time for centralised and distributed polling is not enough to provide a clear understanding on how their corresponding response times scale. It is essential to conduct experiments incorporating response time measurements to evaluate the weight of the time factors mentioned above. The results of such experiments are presented in Section 4.6.

4.5.2. Network Overhead Evaluation

As evidenced in [FUG98], a key factor affecting performance is the overhead induced by transport layer protocols. In particular, TCP is more reliable, yet, traffic intensive resulting from its connection-oriented nature. In contrast, connectionless UDP sacrifices reliability in favour of a lightweight communication mechanism [TAN96]. In our implementation, the choice of the transport protocol is left to the network operator.

Examining SNMP-based polling, if S_{req} and S_{res} are the sizes of the request and response packet respectively and O_T is the overhead imposed by the transport protocol (UDP in the case of SNMP), the polling of N devices for p Polling Intervals (PI) would result in wasted bandwidth of:

$$B_{SNMP} = (S_{req} + S_{res} + 2 * O_T) * p * N \quad (4\text{-}5)$$

Given that the sizes of the request and the response packets are almost identical (see Section 2.4.1) and assuming that v MIB variables are obtained from each host, the previous equation becomes:

$$B_{SNMP} = ((2 * S_{req}) + (v-1) * \Delta S_{req} + 2 * O_T) * p * N \quad (4\text{-}6)$$

where every extra value included in the SNMP response packet's varbind list represents an additional overhead of ΔS_{req} bytes, on average. We also assume that the number *v* of retrieved MIB variables is such that allows the varbind list to be packaged within a single datagram without exceeding the SNMP packet's length limit.

It should be emphasised that the topological structure of the managed network is not taken into account. In other words, it is assumed that the cost of transmitting a certain amount of information from a device to another is independent of their location within the network. The effect of this parameter on the evaluation of network overhead will be investigated in Chapter 6.

Regarding MA-based polling, in overall MA state is transferred *N*+1 times in every PI, including its return back to the manager station. At each point of contact, the MA retrieves a certain amount of management information *b*, subsequently processed by applying a filtering operation. The polling cost highly depends on the increment rate of the MAs state size, which in turn is a function of "*selectivity*" σ ($0 \le \sigma \le 1$), a metric defined in [LIO98] as the proportion of data *delivered* to that *acquired* from each host. For high selectivity values (the major part of the obtained data being filtered at the source) the MAs state size practically remains constant, otherwise the state rapidly grows. Thus, an MA's state size at its i^{th} hop is given by:

$$ST_i = ST_0 + (\sigma * b) * i \tag{4-7}$$

where ST_i represents the compressed state size of an MA when migrating from the i^{th} host (ST_0 is the initial state size). The network overhead imposed by MA-based polling is quantified by:

$$B_{MAs} = \underbrace{N * (C + O_B)}_{Bytecode\ transfers} + \underbrace{p * \sum_{i=0}^{N} (ST_i + O_T)}_{State\ transfers} \tag{4-8}$$

The first term of Eqn. (4-8) describes the overhead imposed when multicasting the MA code to all active MAS servers, whilst the second represents the bandwidth consumed by the MA state transfers between the manager and the polled devices. Post initialisation only the MA state is transferred as the corresponding bytecode is multicasted only once, at the MA's construction time; hence, MA state transfers dominate on the overall overhead when considering long-term monitoring tasks. It is therefore important to apply optimisations so as to minimise the persistent state of MAs, thereby reducing the associated management cost. Such optimisations are proposed in Appendix B.

MAs state size increases exponentially as a function of the hop count (number of visited polled devices). This is demonstrated in Figure 4.21 that shows how the management cost per

PI varies as a function of the network size for various selectivity values (it is assumed that the initial MA state size is 400 bytes, the amount of retrieved data is 1000 bytes and that *no* MA state compression is performed). On the contrary, management cost increases linearly with selectivity (see Figure 4.22). The bytecode transfer and the transport protocol overhead have not been taken into account when drawing these figures, which basically illustrate the dependence of state transfers overhead ($\sum_{i=0}^{N} ST_i$) on the network size N and selectivity σ respectively.

Interestingly, following the deployment of MA bytecode, MA-based approach results in a number of message network transfers per polling interval which is almost the half of the ones generated by SNMP (the ratio is $(N+1):2N$). This observation is of particular importance when many managed devices are located on network segments remote to the manager station. In such case, SNMP will create considerable traffic through the inter-connecting links, whereas an MA object following an 'intelligent' itinerary plan would minimise their utilisation through consecutively visiting the hosts residing on individual remote management domains.

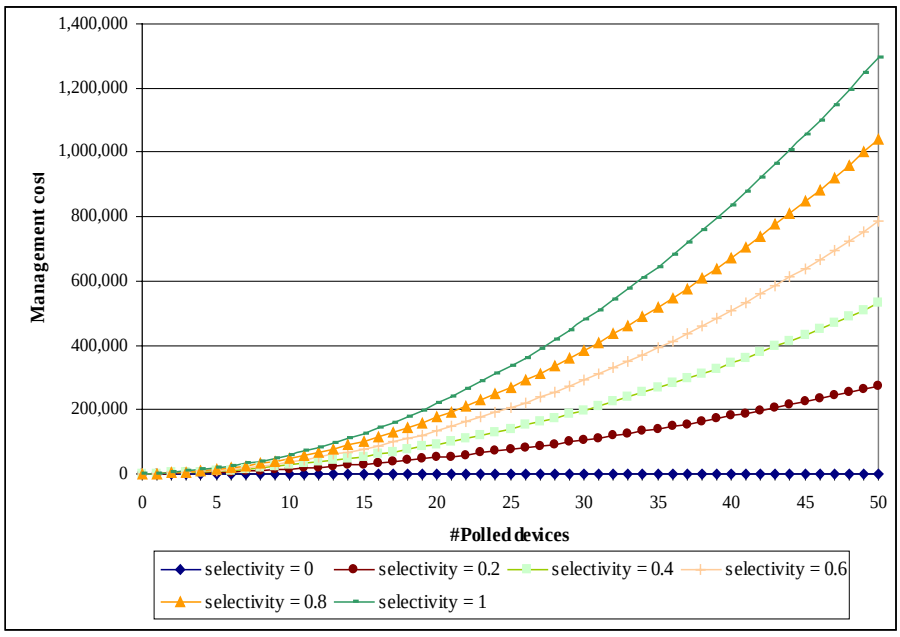


Figure 4.21. Management cost of MA-based polling as a function of the network size for various selectivity values.

The findings of this simple evaluation suggest that MA-based polling performs better than its centralised counterpart only in case that: (a) the volume of retrieved management data is low, (b) the selectivity values are small, (c) the number of polled devices visited by a single MA object is limited. As expected, in any other case MA-based polling does not scale well and may even perform worse than the centralised model. However, the most important conclusion

deduced from this evaluation is the exponential growth of MA state size with the number of visited NEs.

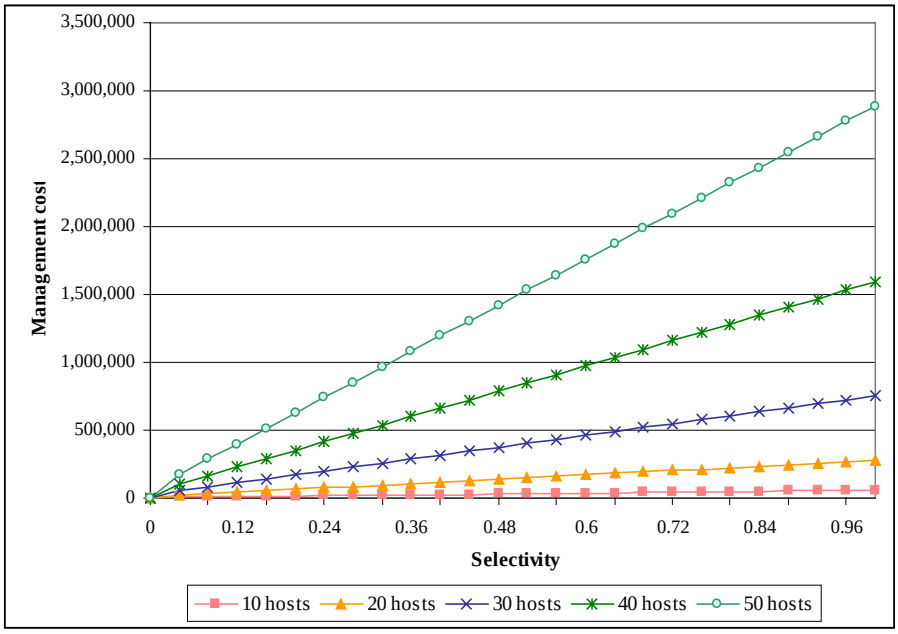


Figure 4.22. Management cost of MA-based polling as a function of the selectivity for various network sizes.

4.6. EXPERIMENTAL WORK

To thoroughly evaluate the proposed framework's performance, a number of experiments have been conducted. In particular, the MA-based solution is compared against both SNMP and a static distributed objects approach based on the Java RMI. This comparison follows two directions: measuring the response time and the network overhead related to management operations.

Response time: The time needed to complete a number of network performance management operations is of critical importance when considering real-time management operations. It is therefore important to experimentally evaluate the response time of MA-based applications and compare it against distributed object approaches. The effect of a number of polling parameters on the overall time also needs to be elucidated. `Utilities.Timer` class has been created to count the timestamps; it includes a number of methods that allow an application to instantiate, start and stop a timer, obtain statistics such as average, standard deviations (SD), etc, print these statistics in text files, etc. The `Timer` class uses the `currentTimeMillis()` method of `java.lang.System` class, which returns the current time in the precision of milliseconds.

The response time experiments presented in this section, start with a performance comparison of MAs against a representative distributed objects technology (DOT), Java RMI.

In particular, a simple experiment is conducted where a string of variable length is transferred from a remote device to the manager station either through an MA object or through an RMI call. A second set of measurements deals with the overall time needed for a multi-hop MA to visit a group of nodes and obtain a certain amount of information from each of them. Additional timing experiments are described in Appendix C. These experiments aim at providing a better understanding of the response time measured for MA migrations; the objective is to investigate how the overall time is distributed among the individual phases of an MA object migration and what is the effect of a number of factors such as the transport protocol used and MA state size.

Network overhead: The traffic generated by the proposed management applications needs also to be measured and compared against alternative approaches. The *WinDump* network analyser [Windump], developed at the Politecnico di Torino, has been used for the network overhead experiments. This program, which represents a graphical version of the well-known *tcpdump* analyser, is capable of capturing packets transferred over a LAN and decoding their contents, providing a useful insight of ongoing network interactions.

In particular, the traffic incurred through MA-based operations is measured and compared against the traffic generated by Java-RMI invocations. The effect that data compression and the transport protocol used for MA transfers have on the measured traffic volume is also weighted.

Regarding the *experimental testbed*, all these measurements have been taken over a lightly loaded Ethernet in the role of the management network, using ten different machines with the following specification: Windows NT operating system, Pentium III (450 MHz) processor and 128MB of memory. The response time experiments were conducted during off-peak period hours, in order to ensure minimum network traffic fluctuations.

4.6.1. Response Time Experiments

4.6.1.1. MAs vs. RMI

This part of the experimental work focuses on comparing MAs against Java RMI, in terms of response time. As far as the MA-based approach is concerned, two different scenarios are examined:

a) An MA object is instantiated by the manager application and sent to a remote device where it obtains and encapsulates an amount of data (a string of characters) and returns back to the manager to deliver the collected data (Figure 4.23a). This type of MA is termed ‘ping-pong’ MA. In the next PI, a new MA is launched and repeats the same procedure.

b) An MA object is instantiated by the manager application and sent to a remote device where it remains permanently. In every PI, this MA (termed the ‘Master’) creates a clone of itself

(termed the 'Slave'). The Slave encapsulates the obtained data and returns back to the manager to deliver the collected information (Figure 4.23b). In the next PI, the same procedure is followed with the creation of a new Slave agent.

The state of migrating MA objects can optionally be compressed before their transmission.

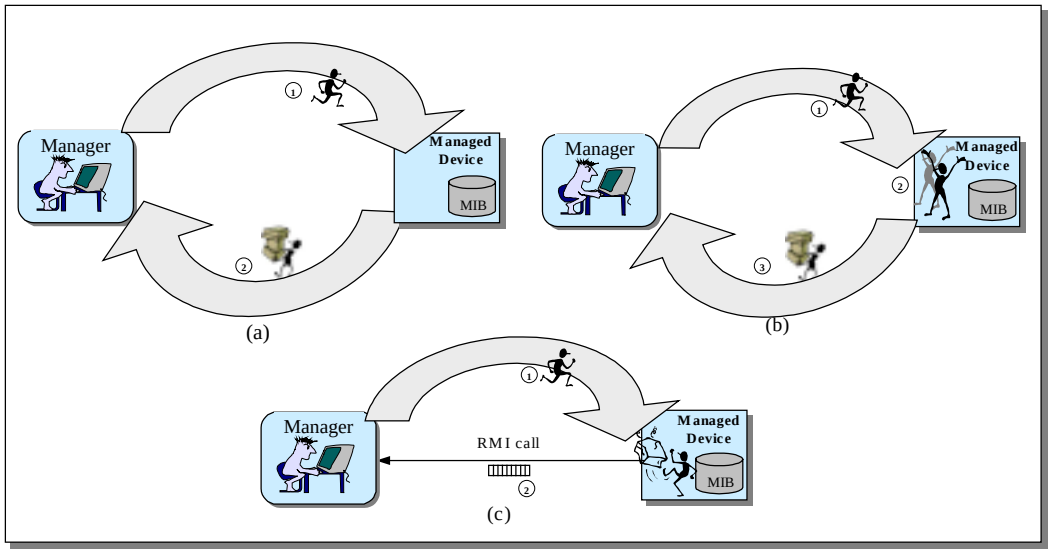


Figure 4.23. Strategies for obtaining information from remote devices: (a) 'ping-pong' MA, (b) MA cloning ('Master-Slave' scheme), and (c) RMI call.

These two schemes, are compared against a third scheme where an MA object is dispatched by the manager application and remains permanently at a remote site, communicating the obtained results to the manager through RMI calls (Figure 4.23c).

String length (bytes)	MAs (non-compressed, round trip)		MAs (non-compressed, cloning)		MAs (compressed, round trip)		MAs (compressed, cloning)		RMI	
	Average	SD	Average	SD	Average	SD	Average	SD	Average	SD
10	84.3	20.2	33.8	6.6	79.9	8.2	31.3	6.0	24.7	14.8
50	84.5	19.0	35.5	5.4	78.7	5.9	26.6	6.9	25.4	14.5
100	83.6	11.0	32.5	8.3	78.1	5.7	29.1	6.4	23.6	7.2
200	86.4	18.0	35.2	10.6	82.9	6.8	26.0	5.3	33.6	96.1
500	88.5	6.5	35.0	5.4	86.9	7.0	32.9	10.8	22.7	6.3
1000	103.9	18.1	42.6	4.6	97.1	13.2	29.3	5.8	24.9	11.3
2000	113.1	17.0	45.6	5.6	108.2	12.7	26.8	7.4	27.7	12.8
5000	147.8	28.6	46.1	7.3	147.7	24.0	35.5	5.0	34.1	11.8
10000	249.2	35.6	62.9	7.5	203.9	20.0	40.4	8.0	43.6	13.5
20000	379.3	33.3	93.1	30.0	306.8	34.0	65.9	20.5	70.8	32.4

Table 4.2. Comparison of MAs vs. RMI-based approaches in terms of response time as a function of the transferred data

The average response times and the corresponding SDs for the three schemes are presented in Table 4.2. The effect of the amount of transferred information on the overall response time is also investigated, hence the length of the string returned to the manager varies from 10 up to 20,000 characters. It is noted that MAs are transferred using the TCP protocol, since it is not

feasible to time transfers over UDP. That is because such a measurement would pre-suppose that the clocks of the sending and receiving hosts are perfectly synchronised, which cannot be easily achieved (a way to get around this problem would be to 'echo' the UDP message back to the origin host and measure the overall round-trip delay, yet, that measurement would not be very accurate). In contrast, when considering MA transfers over TCP, the sender is able to measure the migration latency, as the completion of the transfer is signified by the receiver's request to release the connection. In order to perform a direct comparison between MA cloning and RMI-based approaches, the corresponding measurements have been taken at steady state, that is after the code had been shipped to the remote elements, as the latter is performed only once. Code deployment time is not included.

The results displayed in Table 4.2, are also graphically presented in Figure 4.24. As expected, the approach that incorporates a round-trip travel of an MA object scales worse than the scheme that involves remote cloning, as in the latter the MA travels only in one direction. Interestingly, the response times of the MA-based remote cloning approach and the RMI method invocations almost coincide, with RMI exhibiting a slightly worse scaling. Also, in agreement with the conclusions of the previous section, the compression of MA state becomes more attractive as the amount of encapsulated data, i.e. the MA state size, increases.

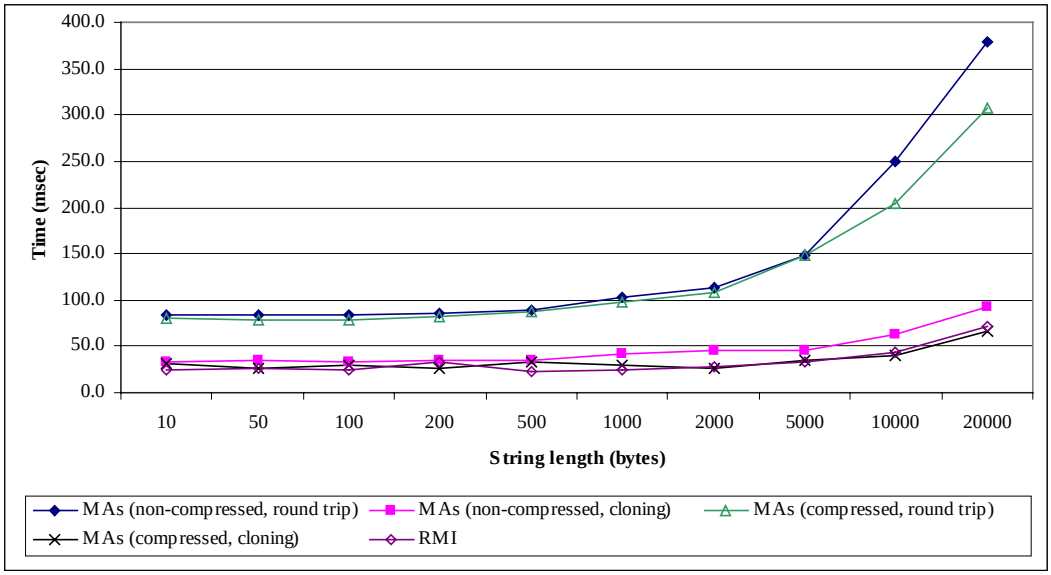


Figure 4.24. Graphical representation of the MAs vs. RMI-based approaches comparison in terms of response time

4.6.1.2. Multi-hop MAs

This set of experiments aims at investigating the effect of several factors on the overall response time of multi-hop MAs, i.e. mobile objects that sequentially visit a number of polled devices, obtain a certain amount of data from each one and return to the manager to deliver their collected data. The evaluated factors are the transport protocol, the network size and the

volume of data obtained from each host. The network size varies from 1 to 10 hosts, while the amount of encapsulated data varies from 10 to 100, 500, 1000 and 2000 bytes. No data compression is used in this case to ensure that the increment on the amount of encapsulated data coincides with the MA's state size increment. The results are presented in Table 4.3 and Table 4.4, corresponding to time measurements where MAs are transferred over TCP and UDP, respectively. The same results are graphically illustrated in Figure 4.25 and Figure 4.26.

 # Collected bytes (from each host)

# Polled Devices	**10**		**100**		**500**		**1000**		**2000**	
	Average	**SD**	**Average**	**SD**	**Average**	**SD**	**Average**	**SD**	**Average**	**SD**
1	84.6	10.3	84.4	18.8	88.8	7.1	89.5	19.4	148.9	328.9
2	85.6	12.2	83.1	9.3	91.4	13.7	100.3	19.1	119.6	20.8
3	89.3	12.4	89.0	16.1	98.5	14.4	113.6	20.2	195.5	41.1
4	108.3	22.1	95.4	15.0	124.8	16.9	159.9	30.3	274.5	29.3
5	108.6	14.9	114.9	20.3	158.8	21.9	220.0	33.5	374.5	31.3
6	138.7	20.5	138.4	13.5	193.8	19.0	301.6	23.6	503.6	38.9
7	149.9	9.5	161.9	18.4	243.5	24.6	383.3	35.4	680.5	74.6
8	173.8	17.4	175.1	20.2	298.3	38.0	475.6	43.2	850.5	67.0
9	185.5	14.5	196.5	14.9	357.5	39.2	620.1	487.3	1098.9	113.5
10	199.3	11.0	230.9	28.4	413.7	41.5	792.8	69.7	1390.1	115.2

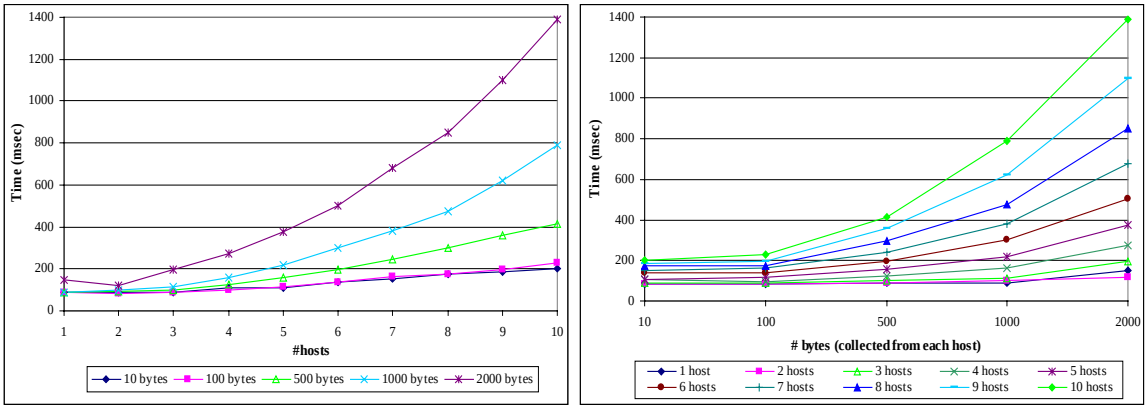
Table 4.3. Average response time and standard deviation for multi-hop MAs for various network sizes and volumes of collected data (TCP protocol used for MA transfers)

 # Collected bytes (from each host)

# Polled Devices	**10**		**100**		**500**		**1000**		**2000**	
	Average	**SD**	**Average**	**SD**	**Average**	**SD**	**Average**	**SD**	**Average**	**SD**
1	77.6	6.3	78.1	6.5	88.2	14.5	87.3	19.0	94.7	10.8
2	81.2	12.5	86.5	10.8	88.9	9.6	95.7	10.1	113.6	20.9
3	81.1	6.4	84.9	14.3	100.1	17.3	117.6	12.8	185.2	29.7
4	87.3	11.9	95.9	13.8	136.6	15.8	170.0	23.7	300.4	28.0
5	104.7	9.2	117.7	10.7	161.0	12.8	227.0	22.5	399.0	30.0
6	123.2	12.8	137.7	22.4	201.6	19.5	313.9	31.9	531.9	31.9
7	143.1	17.3	162.5	19.1	243.3	21.5	403.1	30.1	701.3	40.2
8	158.7	19.2	183.7	19.1	297.8	27.5	497.9	35.3	858.1	41.0
9	180.2	23.5	210.2	15.9	357.6	30.7	602.7	42.3	1101.1	83.6
10	199.3	24.4	235.6	14.9	414.3	34.6	760.9	63.2	1384.3	108.4

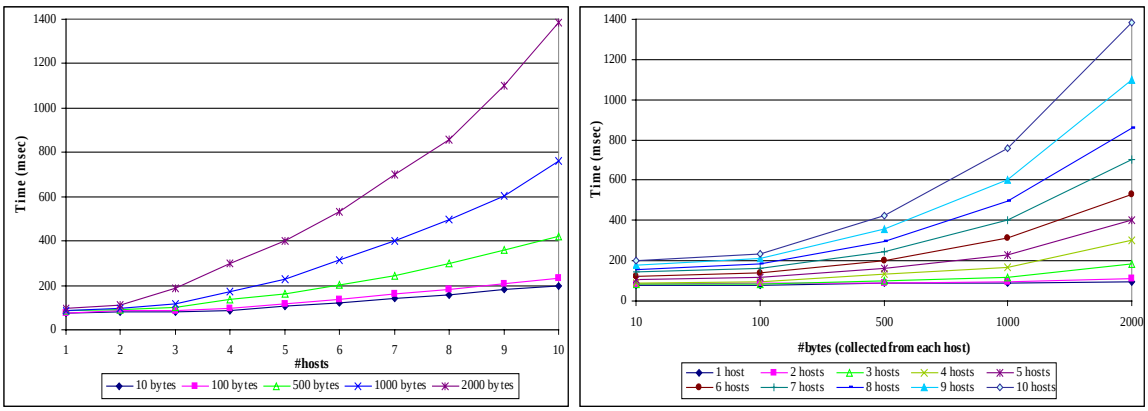
Table 4.4. Average response time and standard deviation for multi-hop MAs for various network sizes and volumes of collected data (UDP protocol used for MA transfers)

The results indicate an exponential growth of the overall response time as the network size increases. Regarding the effect of transport protocol, UDP performs marginally better for small volumes of encapsulated data while the performance of the two protocols converges for higher volumes of data and larger network sizes. The experimental results agree with the conclusions of the quantitative evaluation presented in Section 4.5.2, according to which it is necessary to limit the itinerary length of multi-hop MAs in case that their selectivity is low, i.e. when the volume of collected data at each visited host is high.



(a) (b)

Figure 4.25. Response time for multi-hop MAs as a function of (a) amount of encapsulated information and (b) network size (TCP protocol used for MA transfers)



(a) (b)

Figure 4.26. Response time for multi-hop MAs as a function of (a) amount of encapsulated information and (b) network size (UDP protocol used for MA transfers)

4.6.2. Network Overhead Measurements

An important scalability parameter that signifies the appropriateness of a model on management applications is the volume of data transferred through the network when implementing this model. Although scalability is suggested as the main argument that favours the development of management frameworks employing MAs, very few works (e.g. [BOH00a]) supplement design ideas and implementations with quantitative evaluations or real

traffic measurement data. In addition, the decision to adopt MA-based solutions for management applications instead of static DOT-based approaches is rarely justified through systematic performance comparisons.

This section attempts to correct this insufficiency by presenting the results of an experiment that compares the performance of the proposed MA-based framework against Java-RMI, in terms of the network traffic they incur. The experimental testbed is very simple in this case: it comprises a pair of PCs, with the first playing the role of the manager and the second the role of the managed device.

	RMI (random sequence)	MA (TCP - repetitive sequence)	MA (TCP - random sequence)	MA (UDP - repetitive sequence)	MA (UDP - random sequence)
	# Captured frames				
String length (chars)					
0	8	9	9	1	1
10	8	9	9	1	1
50	8	9	9	1	1
100	8	9	9	1	1
500	8	9	9	1	1
1000	11	9	9	1	2
2000	13	9	12	1	3
5000	19	9	17	1	5
10000	32	9	25	1	9
	Transferred data on the Transport layer (bytes)				
String length (chars)					
0	699	842	842	503	503
10	713	846	858	506	518
50	776	848	932	508	587
100	855	849	986	510	664
500	1527	854	1515	515	1185
1000	2327	859	2220	520	1838
2000	3901	865	3490	525	3079
5000	8609	876	7115	535	6803
10000	16677	889	13613	550	12957
	Transferred data on the MAC layer (bytes)				
String length (chars)					
0	843	1004	1004	521	521
10	857	1008	1020	524	536
50	920	1010	1094	526	605
100	999	1011	1148	528	682
500	1671	1016	1677	533	1203
1000	2525	1021	2382	538	1874
2000	4135	1027	3706	543	3133
5000	8951	1038	7421	553	6893
10000	17253	1051	14063	568	13119

Table 4.5. Comparison of MAs vs. RMI-based approaches in terms of network overhead: The total number of captured frames and the volume of the transferred data on the transport and MAC layers

First, the manager instantiates and launches an MA object to the managed device. The MA then creates a string (`java.lang.String`), which is either directly returned (passed as a parameter) to the manager through an RMI call (first scenario) or encapsulated into the MA's state and returned to the manager application through a second MA transfer (second scenario). The experiment has been repeated for string lengths that very between 0, 10, 50, 100, 500, 1000, 2000, 5000 and 10000 characters. Focusing our attention on the second scenario, the MA transfers may be performed over TCP or UDP protocol. In addition, in order to assess the effect of data compression, the encapsulated string might either be a repetition of a 10-characters-long substring or composed by randomly generated characters (through using the `random()` method of the `java.lang.Math` class). The use of data compression in the RMI-based approach was not possible, as that would require modification of the code that determines the way that RMI handles input and output streams.

By running the *Windump* network analyser and using appropriate filters, the packets exchanged between the two devices have been captured and analysed. The results of this experiment are shown in Table 4.5, that presents the total number of packets captured during a single transaction, the volume of data transferred through the network on the OSI *transport* layer (including the TCP/UDP headers) as well as on the *MAC* (Medium Access Control) layer (including the Ethernet headers). The latter is also graphically presented in Figure 4.27. Unlike response time experiments, there was *no* need to repeat the measurements since the network overhead could be accurately measured by discriminating it from the background traffic[8].

Clearly, using TCP for transport protocol results in exchanging a much higher number of packets, especially when the size of the actual 'useful' data is small; most of these packets relate to the connections establishment/release phases. On the other hand, UDP, operating on connectionless fashion, does not require the exchange of any control packets and thus results in lower network overhead at the expense of more unreliable data transfers.

Notably, the effect of compression is much more prominent when the encapsulated string represents a repetitive sequence of characters; in such case, there is a high degree of redundancy, which is effectively exploited by the *gzip* compressor to achieve higher compression ratio and diminish network overhead. Compression becomes more effective as the volume of transferred information increases, for instance the compression ratio equals 90% for 50-characters-long string and 99.5% for 10000-characters-long-string. High compression ratios are not, however, feasible when the string is randomly generated. Yet, the latter represents a

[8] *Windump* offers a variety of 'filters' that control the capturing process so that the captured frames are those that conform to certain restrictions, specified by the user.

more realistic scenario, as collected management information is unlikely to exhibit high redundancy.

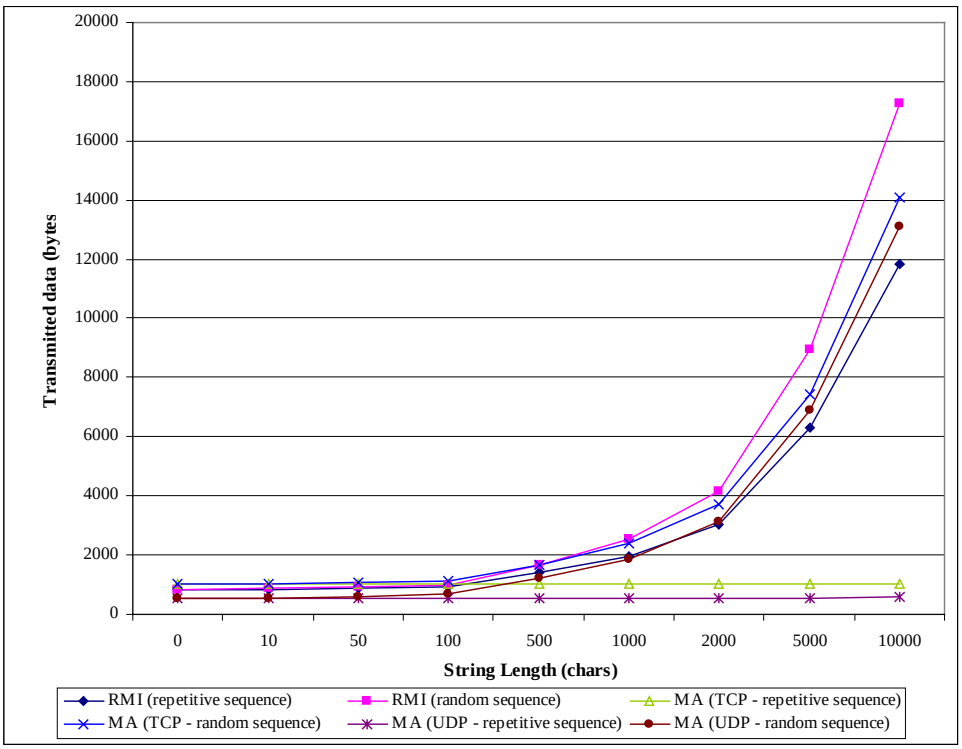


Figure 4.27. Graphical representation of Mobile Agents vs. RMI comparison in terms of network overhead

The most significant outcome of this experiment is that even when encapsulated strings are randomly generated, the performance of MA-based solutions is slightly superior to that of the RMI-based approach, especially as the volume of transferred data increases. The experiment would be more complete if our framework was also compared against CORBA. However, a similar comparison reported in [BOH00a] indicated that CORBA and RMI performances are in the same order of magnitude, with CORBA exhibiting slightly improved scalability. Still, these conclusions may be reconsidered should different implementations of RMI and CORBA are evaluated.

Although the relation of this experiment to management applications might not be evident, it sufficiently demonstrates the competence of the proposed framework against static distributed objects technologies. The results of additional experiments, presented in Chapter 7, will prove the precedence of MA-based solutions over traditional centralised (SNMP-based) models, in terms of network overhead, when considering realistic management scenarios. As a final remark, it should be mentioned that the network overhead incurred in multi-hop MA-based operations is not shown here, but will be examined in Chapter 5 (Section 5.4.2).

4.7. SUMMARY - CONCLUDING REMARKS

This chapter comprised a description of the design and implementation of a Java-based MAP intended for use in NSM applications. In order to achieve management orientation the framework has been implemented from scratch, incorporating a variety of features that meet the special requirements of distributed management applications. Hence, the presented infrastructure exploits the benefits associated with MA technology, providing a lightweight design with minimal impact in managed devices, moderate use of network resources, prompt execution of time-sensitive tasks and ease in the introduction of new management operations. The key features of the described framework are summarised in the following:

- Acknowledging the huge installation basis of SNMP within TCP/IP networks, the introduced framework is integrated with an SNMP stack developed in Java, allowing incoming MA objects to locally interact with SNMP agents through the SF component.
- A variety of GUIs offering MIB browsing, data visualisation, performance statistics and high-level management operations have been designed.
- The manager application allows performing both centralised (SNMP) and distributed, MA-based operations upon managed devices.
- MAS servers have been designed so as to have lightweight footprint on system resources and therefore provide execution environments easily installable even on devices with limited storage capacity and processing capability.
- Client/Server interactions between the manager and the managed devices is supported through an RMI server controlled by the manager application and another optionally installed and integrated within MAS modules.
- MAS entities keep an active control over locally executing MA threads through registering and maintaining a reference of the corresponding MA objects. That also allows the administrator to visually profile and remotely control (stop, suspend, resume, etc.) MAs execution through RMI calls.
- A set of security features has been incorporated into the introduced framework. Travelling MA objects are authenticated at each visited host ensuring that they have been originated from a trusted host, their actions are authorised, whilst the data encapsulated into their state can optionally be encrypted. Although not answering all the security concerns raised against MA technology, these features ensure that NEs are reasonably safe against MA attacks and MA objects shielded against malicious hosts.

- In order to improve flexibility and maximise code reuse, MAs are implemented following a flexible hierarchical approach whereby MA classes created by the MAG tool inherit the properties of an MA's 'superclass', which defines the root attributes required to employ mobility characteristics.
- The administrator is given the choice between TCP and UDP as a transport protocol with respect to MA migrations. As shown by the experimental results, the choice of transport protocol may affect both the network overhead and latency of MA-based operations.
- The itineraries of travelling MAs may either by automatically built or manually assigned.
- A graphical tool (MAG) that automates the generation of MA code has been designed facilitating the introduction of new management tasks conforming to generic service patterns.
- A number of fault-tolerance features have been integrated into the framework. These cover the cases where a host included in an MA's itinerary has failed, an interconnecting link has broken or the manager station itself is down.
- A customised MA ClassLoader has been created, which in addition to receiving new instances of management tasks at runtime, is able to distinguish different versions of the same MA class, thus enabling modifications over existing tasks to take instant effect.
- A number of transport protocol parameters have been configured so as to accelerate MA migrations and enable the timely execution of real-time management operations (see Appendix B). Along the same line, MA objects are assigned the highest priority among the processes executing on their hosting devices.
- Based on the observation that the network overhead associated with MA transfers is mainly due to the MA code rather than the MA state information, a lightweight code distribution scheme has been implemented. In particular, MA bytecode is transferred to managed devices at the MAs' creation time, with only MAs state transferred following that.
- To further reduce the impact on network resources, MAs state is compressed (*gziped*) prior to its transfer.

In addition to presenting implementation details, this chapter also included a quantitative evaluation of the introduced framework, providing simple mathematical formulations that quantify the response time and network traffic load associated with centralised and MA-based management. Finally, a variety of experiments have been conducted in order to assess the performance of the introduced framework in practice. The experimental work has been

twofold, namely the framework's performance is assessed both in terms of the response time and the generated network traffic involved in management operations.

Experimental results revealed that our framework marginally outperforms RMI both in terms of network overhead and response time when considering simple *data transfers between a pair of hosts* (manager - managed device). Nevertheless, although reducing the number of messages communicated through the network compared to centralised models, the use of MAs for managing multiple NEs should be carefully evaluated. As evidenced in Section 4.6.1.2, the use of a single multi-hop MA that visits all managed devices obtaining a specific amount of data from each of them, represents a *non-scalable solution*, especially for tasks where selectivity is low (i.e. only a small portion of the obtained information is filtered at the source). That conclusion dictates that MAs itinerary length should be limited.

It should also be emphasised that the topological structure of the introduced framework and the distribution of managed devices within the network have not been taken into account in the quantitative evaluation of Section 4.5 or the experimental results, assuming fixed MA migration cost, regardless of the location of sending and receiving devices within the network.

These issues will be addressed in Chapters 5 and 6, where more efficient models that improve the framework's scalability, flexibility and adaptation on changing networking environments are identified.