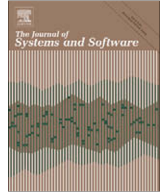
The Journal of Systems and Software 82 (2009) 355–371

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss



A mobile agent platform for distributed network and systems management

Damianos Gavalas*, George E. Tsekouras, Christos Anagnostopoulos

Department of Cultural Technology and Communication, University of the Aegean, Arionos and Sapfous Street, 81100 Mytilene, Lesvos Island, Greece

ARTICLE INFO

Article history:
Received 13 March 2008
Received in revised form 23 June 2008
Accepted 24 June 2008
Available online 3 July 2008

Keywords:
Network and systems management
Mobile code
Mobile agents platform
Custom class loader
Optimal itinerary design
Agent generator

ABSTRACT

The mobile agent (MA) technology has been proposed for the management of networks and distributed systems as an answer to the scalability problems of the centralized paradigm. Management tasks may be assigned to an agent, which delegates and executes management logic in a distributed and autonomous fashion. MA-based management has been a subject of intense research in the past few years, reflected on the proliferation of MA platforms (MAPs) expressly oriented to distributed management. However, most of these platforms impose considerable burden on network and system resources and also lack of essential functionality, such as security mechanisms, fault tolerance, strategies for building network-aware MA itineraries and support for user-friendly customization of MA-based management tasks. In this paper, we discuss the design considerations and implementation details of a complete MAP research prototype that sufficiently addresses all the aforementioned issues. Our MAP has been implemented in Java and optimized for network and systems management applications. The paper also presents the evaluation results of our prototype in real and simulated networking environments.

© 2008 Elsevier Inc. All rights reserved.

1. Introduction

Current IP network management systems are typically based on centralized client–server architectures, with the simple network management protocol (SNMP) (Stallings, 1999) being the core management protocol. Within SNMP, the *manager* application serves the role of the client and the static SNMP *agents*[1] (installed on managed devices) serve the role of distributed servers. The functionality of both the manager and the SNMP agents is rigidly defined at design time. Physical resources are represented by managed objects. Collections of managed objects are grouped into tree-structured management information bases (MIB). Despite its wide deployment base, SNMP presents serious scalability problems when the size and complexity of the network increases (Liotta et al., 2002; Stephan et al., 2004). Thus, the need for decentralized and distributed network management architectures is more important and necessary than ever before (To et al., 2005).

The emerging mobile agent (MA) technology can play an important role in distributed network and systems management (NSM) (Eid et al., 2005). The term agent is used to denote a (usually static) software entity with a well-defined role, typically acting on behalf of a human or another software component, which may be used in a variety of applications (e.g. Manvi and Venkataram, 2006). In contrast, the term MA refers to a software unit that travels between network nodes following either a pre-defined or a context-dependent itinerary (Fuggeta et al., 1998). Post its creation, an MA can carry its *persistent state* and *code* to another node, where its execution can be restarted or resumed. Through interacting with a node, an agent can perform complex processing and filtering operations upon retrieved data, directly control equipment and dynamically deploy software to the nodes. That is, the agent can carry the application logic where it is needed and only accumulate filtered data rather than the entire data set retrieved from nodes (To et al., 2005). Several researchers have proposed the application of MA technology in the area of NSM (Bohoris et al., 2000; Du et al., 2003; Liotta et al., 2002; Puliafito et al., 2000; Rubinstein et al., 2003; Satoh, 2006; Stephan et al., 2004). Typical scenarios involve: (a) multi-hop MAs that sequentially visit a set of nodes to retrieve and filter management data and then deliver high-level results to the manager host, or (b) single-hop agents uploaded to remote devices where they monitor systems and notify the manager in the event of a 'system's health parameter' crossing a pre-determined threshold. Despite its potential though, MA technology has not yet achieved the anticipated commercial success, mainly due to security concerns[2] (Fuggeta et al., 1998; Jiang et al., 2004); however, several software developers have lately incorporated agent technology

* Corresponding author. Tel.: +30 22510 36643.
E-mail addresses: dgavalas@aegean.gr (D. Gavalas), gtsek@ct.aegean.gr (G.E. Tsekouras), canag@ct.aegean.gr (C. Anagnostopoulos).

[1] SNMP agents serve as an interface between the manager application and the legacy systems and should not to be confused with mobile agents.

[2] The most critical security concerns related to MAs comprise: (a) protecting mobile hosts from malicious agents, (b) protecting agents from malicious hosts, and (c) protecting sensitive information carried by agents from eavesdropping. The first is addressed through implementing authentication and authorization features which ensure that only trusted agents may be executed, in a restricted authority domain. The second may be achieved through protecting agents against tampering, while the third is sufficiently addressed through encrypting sensitive data.

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2008.06.034

into their software products, including management product solutions (Present Technologies, JAMES project; SysteMATech; Whitestein Technologies). In particular, JAMES (Present Technologies, JAMES project) is a Java-based mobile agent platform used for the management of telecommunication and data networks, while SysteMATech is tailored to decentralized systems management applications. Whitestein Technologies is a software firm that makes use of the advanced capabilities of software agents within its products, solutions and services in application domains that span transportation, production and supply networks.

A large number of commercial and research initiatives have led to the development of mobile agent platforms (MAPs). While many amongst them are referred to as 'general-purpose' platforms (i.e. they can be effectively used in any distributed computing application), others have been optimized having management applications in mind. However, most of these platforms impose considerable burden on network and system resources and also lack of essential functionality, such as security mechanisms, fault tolerance, strategies for building network-aware MA itineraries and support for user-friendly customization of MA-based management tasks.

In this paper, we discuss the design considerations and implementation details of a complete MAP research prototype that sufficiently addresses all the aforementioned issues. It is noted that an earlier version of this work has been published in Gavalas (2006) which has been extensively revised and extended so as to discuss all the design issues, implementation details and experimental evaluation of our proposed platform.

The remainder of the paper is organized as follows: Section 2 reviews works related to our research. Section 3 discusses the design considerations for our MAP prototype, while Section 4 describes the implementation details of the individual MAP software components. Experimental evaluation results on real networking environments are presented in Section 5 and simulation results in Section 6. Section 7 concludes the paper and draws direction for future research on the field.

2. Related work

The phenomenal popularity of MAs is reflected on several industrial initiatives that led to the development of numerous MAPs (Eid et al., 2005). Most of them represent commercial, general-purpose MAPs, e.g. IBM Aglets (Mobile Agent Platform (IBM)), Toshiba Bee-gent, Present Technologies JAMES (Present Technologies, JAMES project). Among other areas, these platforms have been employed in distributed management applications (Du et al., 2003; Stephan et al., 2004). From the management viewpoint though, general-purpose MAPs share several weaknesses: (a) rich, but unnecessary functionality: some of their features are unnecessary for management applications, yet, they are 'hard-coded' and result in large MA sizes that affect the usage of system and network resources; (b) lack of essential features: some features considered as essential for flexible management systems are not supported, for instance, the ability of servers that receive incoming MAs to distinguish between different versions of the same MA class, which may reflect the update/modification of an existing management task; (c) questionable support: companies shipping commercial MAPs, often suspend their support (for instance GMD FOKUS and IKV++ ceased their support for the -once popular- Grasshopper platform (Bäumer and Magedanz, 1999)).

The aforementioned weaknesses of general-purpose MAPs led to the development of numerous MAP research prototypes with network management orientation, aiming at optimizing functionality, flexibility and performance aspects, e.g. Codeshell (Bohoris et al., 2000), MAP (Puliafito et al., 2000) and MobileSpaces (Satoh, 2006). However, a number of limitations have been identified on these MAPs:

- *Heavyweight migration schemes*: Most existing MAPs involve the transfer of both state *and* code at each MA migration (Satoh, 2006). However, the transfer of code is necessary only when the MA visits a device for a first time (the Java class loader stores every loaded class on a local code cache). That inefficient scheme may result in serious scalability problems both in terms of latency and migration overhead. This problem is partially addressed through the migration strategy proposed in Puliafito et al. (2000), where *only* the MA state is transferred and should the corresponding code is not present at a visited device, the code is downloaded from a remote code server. However, this approach increases the latency (the MA's execution cannot start until its code is downloaded) and involves more complex migration mechanisms, which are not necessary when MAs itinerary is known in advance.
- *MA services customization*: The development and customization of MA-enabled management tasks is not effortless with available MAPs (Bohoris et al., 2000; Puliafito et al., 2000; Satoh, 2006), as it requires programming skills and detailed knowledge of the MAPs' design.
- *Class loading*: Most MAPs include a class loader component, able to receive and load at runtime visiting MAs bytecode. Yet, to the best of our knowledge, there is not any MAP which allows to overwrite a cached MA class, i.e. to dynamically upgrade MA-enabled management tasks (this problem is related to a limitation of the default Java class loading mechanism). Aglets framework (Mobile Agent Platform (IBM)) represent an exception to this rule, yet, using a resource-demanding method detailed in Section 4.3.8.
- *Security*: Not all management-oriented MAPs sufficiently address security issues related to MAs, e.g. authentication (Bohoris et al., 2000), authorization (Bohoris et al., 2000; Puliafito et al., 2000; Satoh, 2006) and encryption (Mobile Agent Platform (IBM); Toshiba Bee-gent; Bohoris et al., 2000).
- *Fault tolerance*: MAPs should be able to survive situations where link or node failures disrupt the normal migration process of roaming MAs or the communication with the manager station. However, only few platforms (e.g. Present Technologies, JAMES project) have addressed fault tolerance issues.
- *Lack of itinerary optimization*strategy: When multi-hop MA-based management tasks are concerned, the order in which MAs visit managed devices (i.e. the MAs' itinerary) is a crucial factor seriously affecting the overall delay and the network overhead imposed by the MA transfers (Fuggeta et al., 1998; Gavalas, 2004). Hence, an efficient MA itinerary design approach is needed, which is not addressed by any of the available MAPs.

In this paper, we introduce a Java-based MAP research prototype that addresses all the aforementioned limitations of existing general-purpose and management-oriented MAPs. The main strengths of our MAP are:

- Network management applications orientation: Unlike publicly available general-purpose MAPs, the introduced MAP takes into account the special characteristics and requirements of management applications and therefore provides a flexible and scalable environment tailored to distributed management operations.
- Lightweight memory and storage footprint (only the necessary MAP functionality for management operations is included).
- Support for existing management standards: Our architecture encompasses the dominant management framework of the Internet world, i.e. the SNMP. Due to its huge installation base,

integration with SNMP was considered of vital importance to maintain compliance with legacy management systems (Pagurek et al., 2000).
- Adoption of modular MAP architecture that eases the addition of new services or the modification of existing ones.
- Implementation of a novel lightweight migration scheme wherein the MA code is deployed at startup and only the MA state transferred thereafter, resulting in minimal usage of network resources (bytecode size is typically much larger than state size (Fuggeta et al., 1998) and faster class loading). The performance trade-off related to our code deployment strategy is discussed in Section 3.10.
- Incorporation of various migration optimization (programming) techniques suggested in Gavalas (2004).
- Ability of managed devices to distinguish between different versions of the same MA class, which may reflect the update/modification of an existing management task; to achieve that, a novel class loading mechanism has been implemented as an extension to the standard Java virtual machine (JVM).
- Easy addition of new MA-based management tasks through a novel tool (mobile agent generator) which removes the requirement for programming skills by the network administrator.
- Support for essential security aspects (authentication, authorization and encryption).
- Fault tolerance: Our MAP has been designed so as to enable a reasonably fault-tolerant system, able of recovering from some basic software faults that disrupt the normal migration process of roaming MAs.
- Implementation of a software module that, depending on the underlying network topology and the set of devices involved in a management task, constructs near-optimal MA itineraries thereby optimizing the usage of network resources.

At this point, we should make a distinction among mobile agents and agents that derive from artificial intelligence (AI) formalisms, termed as "intelligent agents" (IAs). Basically, IA ideas and technologies have been influenced by a wide variety of disciplines and practices. However, the origins of this term are in the field of AI, in particular distributed artificial intelligence (DAI). It is used since the beginning of the 1980's to reflect the idea of creating autonomous software entities that behave based on self-contained intelligence (knowledge and inference) (Magedanz et al., 1996). However, the following attributes are typical for intelligent agents: An agent is a self-contained software element responsible for performing part of a programmatic process. Therefore, it contains some level of intelligence, ranging from simple predefined rules to self-learning AI inference machines. It acts typically on behalf of a user or a process enabling task automation. Agents operate rather autonomously (they are often event or time triggered) and may communicate with the user, system resources and other agents as required to perform their task. Moreover, more advanced agents may cooperate with other agents to carry out tasks beyond the capability of a single agent. Finally, as transportable objects, they may move from one system to another to access remote resources or to meet or cooperate with other agents. In principle, intelligent agents have been agreed upon to share the following properties (Franklin and Graesser, 1997; Kendall et al., 1998; Krause and Magedanz, 1996): acting on behalf of someone; reactiveness; autonomy; proactiveness; communication and collaboration ability; learnability and rationality; flexibility; personality; mobility.

Evidently, the MAs involved in our MAP prototype only have a small fraction of the above-listed properties, that is acting on behalf of someone, autonomy and mobility. As such, they cannot be classified as IAs, hence, they cannot be defined neither described using the AI domain's formalisms.

3. Design considerations for the MAP prototype

This section identifies the design space and justifies the most important design choices regarding our MAP research prototype.

3.1. Modular design

One of our basic design objectives has been the adoption of modular MAP architecture that facilitates code reusability and eases the addition of new services or the modification of existing ones. Hence, in addition to identifying the main software modules of the MAP framework, each one of these modules, follows a modular design and has been built on the basis of several independent and interrelated sub-components.

Our MAP prototype consists of the following major building blocks (see Fig. 1): (a) the Manager, responsible for launching and controlling MAs and displaying results; (b) the MAs, capable of migrating between the managed entities to collect information based on pre-defined policies; (c) the mobile agent server (MAS), capable of receiving MAs and providing an interface to the local physical resources; (d) the mobile agent generator (MAG), a tool that automates the creation and deployment of service-oriented agents.

It is noted that two of the aforementioned components (MA and MAS) are common place in almost all known MAPs (e.g. (Mobile Agent Platform (IBM); Toshiba Bee-gent; Bohoris et al., 2000; Puliafito et al., 2000; Satoh, 2006)), while the manager component is essential in management applications. However, the MAG tool represents a key feature of our MAP, not available on existing platforms; the motivation for its development is explained in Section 3.8.

In a typical scenario, the network administrator creates a service-specialized MA through the MAG tool (described in Section 4.4). The MA is then stored in the MA code repository and retrieved by the manager application when a monitoring task is about to start. The instantiated MA object is then supplied with an itinerary and sequentially visits a group of nodes. On each node, the MA is received by the local MAS, which acts as an interface between incoming MAs and the legacy system. The MA retrieves management data from the local SNMP agent and proceeds to the next destination host, ultimately returning back to the manager station to deliver collected data.

3.2. Implementation platform and database technology

A major design concern for our MAP has been to guaranty operation on any network device, regardless of the underlying hardware platform or operating system. To address this issue, we have chosen Java as implementation platform (the MAP has been entirely developed using JDK 1.4.1 (Java 2 Platform)), known to offer platform-independence.

Another serious concern has been the choice of an appropriate database technology. The manager processes deliver data in real-time to identify potential alarm levels of monitored variables and stores these data in a database for future statistical analysis. The relational database technology chosen both for storing the MA code and management data is MySQL, because of its low cost and effortless connection with the Java-based manager application, through a Java database connectivity (JDBC) MySQL driver.

3.3. Choosing intelligent agent itineraries

In most distributed applications, using a single MA that sequentially visits a set of devices set may result in serious scalability problems. That is in large networks the *round-trip delay* of the MA

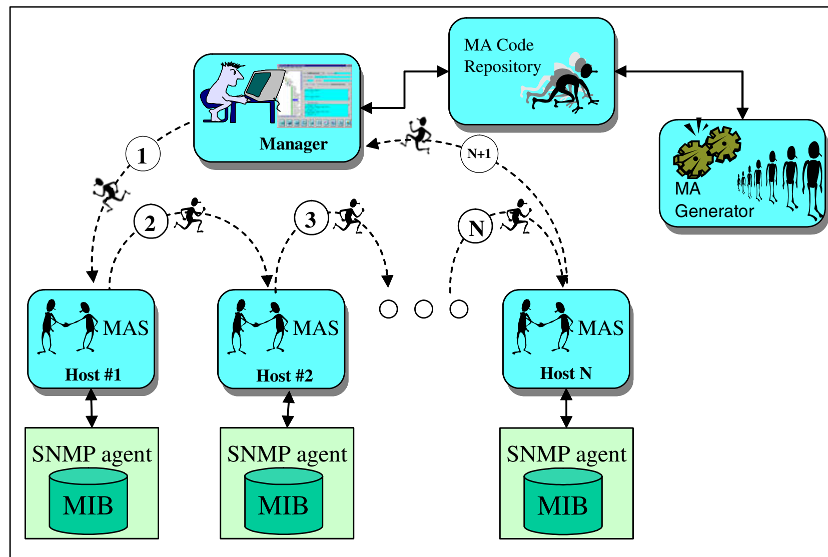


Fig. 1. The mobile agents-based infrastructure.

greatly increases as the overall travel time depends on the number of hops executed by the MA. More importantly, the *network overhead* imposed by the MA transfers grows exponentially with the network size, since MAs accumulate more data from every visited node and become heavier on the next hop (Fuggeta et al., 1998). Hence, a more efficient MA itinerary design approach is needed, which should take into account the underlying managed network topology and the volume of accumulated management data; in such approach, multiple MAs (each assigned a limited-hop itinerary) will be employed in parallel to carry out given management tasks, thereby minimizing the overall response time and optimizing the use of network resources. To this end, we have designed and implemented the Heuristic algorithm for itinerary planning (HIP), which has been introduced and evaluated in Gavalas and Politi (2006).

The main motivation for designing optimal agent itineraries is to minimize the overall network overhead associated with MA transfers and also to maintain low latency for completing their task. In network monitoring applications for instance, using a single MA object that sequentially visits all managed devices, regardless of the underlying topology may actually lead to performance worse than the conventional SNMP-based approach. The performance further declines, when the monitoring MA collects data from multiple subnets, often interconnected by low-bandwidth links. In such cases, the traffic associated with management tasks typically traverses several network segments and, when summed up, results in increased bandwidth waste (Gavalas and Politi, 2006), hence an itinerary optimization process would minimize the number of "expensive" link traversals. HIP algorithm represents an efficient approach for addressing the itinerary optimization problem and has been shown to provide better performance results in comparison to alternative existing approaches (Gavalas and Politi, 2006).

3.4. Optimizing MA migrations

A crucial design objective for a MAP is to minimize the network overhead and latency associated with MA migrations. Following the suggestions reported in Gavalas (2004), we have implemented a series of design optimizations: (a) use of a lightweight MA code transfer scheme (described in Section 4.4), (b) reduced state size through minimal use of non-transient objects and use of primitive instead of complex data types for non-transient objects (e.g. use of arrays instead of Java Vector objects), (c) compression of MA state and use of stream buffering for MA transfers.

3.5. Security considerations

MA technology has not yet achieved the anticipated commercial success, mainly due to security concerns (Fuggeta et al., 1998; Jiang et al., 2004). Hence, it is essential for a MAP to provide a reasonably safe environment against malicious hosts and malicious agent attacks. Our objective is to provide the following security features:

- authentication, to ensure that only trusted agents (dispatched by authorized hosts) are instantiated;
- authorization, to restrict the authority domain of visiting MAs upon legacy systems, e.g. prevent MAs from directly accessing files, creating sub-processes, shutting down the MAS application, etc;
- encryption, to protect collected management data from eavesdropping;
- protection of MAs against tampering (to prevent malicious hosts from modifying sensitive MA properties).

With these security precautions described in this section, we consider the network devices to be reasonably safe from malicious attacks. In general though, the implementation of an impenetrable security shield has not been a principal aim during the development of our architecture. Besides, agent security is a particularly complex subject and represents an active and evolving research area (Fuggeta et al., 1998; Jiang et al., 2004).

3.6. Fault-tolerance features

Our architecture has also been designed to provide a limited range of 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:

1. An MA should adapt to unexpected situations, such as the failure of a host's listener daemon, as shown in Fig. 2. In this case, the TCP connection establishment fails (step 1). The MA records the unreachable host's name into the MA's 'problem folder'

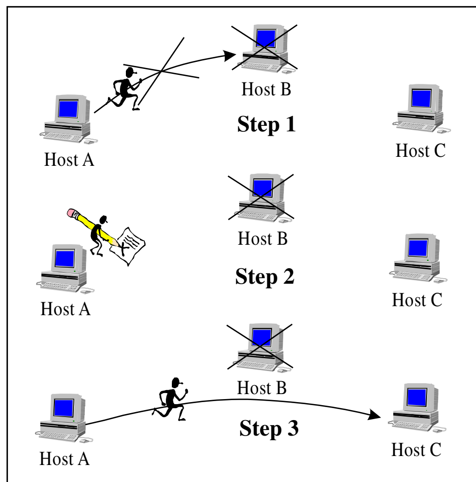


Fig. 2. Tolerating node failures: (a) MA reaction to the detection of a failed MAL thread.

(step 2) and retrieves the next destination host from the itinerary folder. The MA will then migrate to this host (step 3). When returning to the manager host, the MA reports the failed devices to the manager application, which in turn will take any necessary fault recovery actions.

2. A second scenario involves a fault in the listener daemon of the manager host itself (Fig. 3), which represents a special case. Certainly, loss of management data carried by an MA should be avoided. Hence, upon detecting a fault (step 1) the MA 'sleeps' for a given interval (step 2) 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 (step 3), otherwise it is disposed of.

If – in the process of an MA's migration – a node or a link fails, the persistent state of the MA (the data already collected from previously visited hosts and transient variable values) is *not* lost. That is, the persistent state of departing MAs is disposed of only in the event of a successful agent migration. In the failure event of node hosting MA objects, the manager application re-dispatches the monitoring MA, after a pre-defined time period (500 ms) elapses; this MA will skip the crashed node, based on the first scenario described above. An alternative fault tolerance approach would be to use a redundant number of agents and MAS instances.

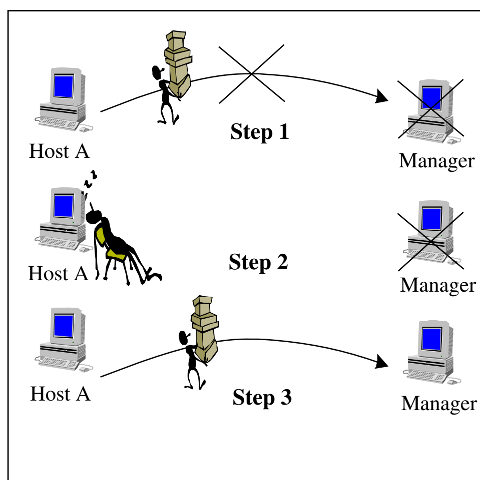


Fig. 3. The specific case of MAL thread failure on the manager host.

It should be noted that fault-tolerance has not been a primary design goal for our framework. Fault tolerance mechanisms must be based on the provision of useful redundancy, both for error detection and error recovery. In software the redundancy required is not simple replication of programs but redundancy of design (Randell, 1975). Our objective has only been to support some fault-tolerance features so as to enable a system able of recovering from some basic software faults. A weakness of the above presented MAP's fault-tolerance mechanisms is that the manager application represents a single point of failure. Namely, in the event of manager host's failure travelling MAs cannot deliver their collected monitoring data. This can be easily addressed by hosting a backup manager application, preferably on a separate station. Also, in the first fault scenario presented above, our platform basically can detect only TCP connection establishment fails. Certainly, such failures do not necessarily translate to host failures, as they may be due to network disconnections, firewall settings or some host's listener daemon failure.

3.7. Dealing with the versioning problem

In a typical MAP, when an MA arrives at a host, its definition is retrieved from the network stream, and cashed within the default JVM's class loader (CL). This CL works fine as long as MAs' definitions remain unchanged; it is not able though to distinguish different versions of the same class, due to a limitation of the default JVM's CL[3] (Venners, 1998). That is, the older (cashed) version of a class is still loaded even if an updated version of the same class has been uploaded. This problem, known as 'versioning' problem, represents a major issue for MA-based management applications wherein management tasks are often modified at runtime and expected to take immediate effect.

To the best of our knowledge, none of the existing MAPs copes with this MAs 'versioning' problem and, as a result, the administrator 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 an adequate solution for networks comprising hundreds of managed nodes. To this end, we have extended the standard JVM, designing and implementing a custom CL, the mobile agent class loader (MACL), which is able to identify different versions of the same MA class and enable the execution of the most recent version while disposing the old one.

The MACL 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 network stream rather than from the cache.

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 'overwritten' 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 minimize the time needed to load these MAs when they first visit the node (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 Fig. 4.

[3] To the best of our knowledge, the newer versions of Java (e.g. Java 6) do not eliminate or ameliorate the consecutive class loading problem.

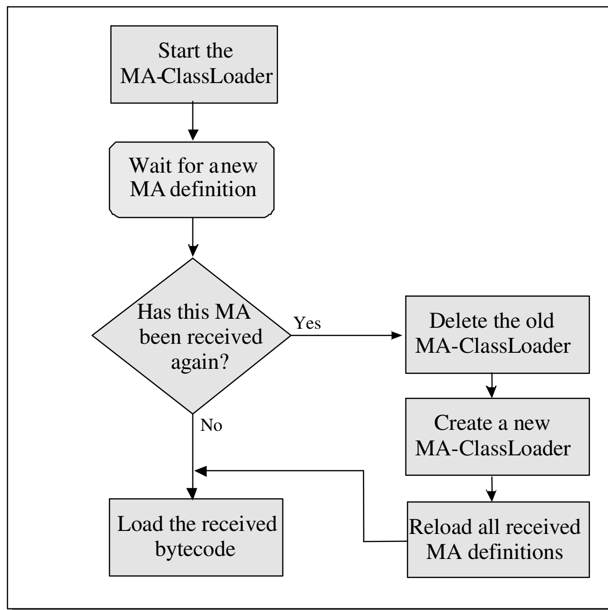


Fig. 4. Class loading mechanism block diagram.

The implementation details of the MACL component are provided in Section 4.3.8.

3.8. Automating the creation and deployment of service-oriented agents

The focus of current research on MA-based management is on the development of mobile agent platforms (Present Technologies, JAMES project; Puliafito et al., 2000) and applications for MAs (Du et al., 2003; Stephan et al., 2004). 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. Therefore, we have designed and implemented the mobile agent generator (MAG) tool that satisfies a number of design objectives:

- The MAG ensures that the architecture 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.

The functionality of MAs created by the MAG may only be an extension of limited generic service types, corresponding to specific service 'templates'. The MAG uses a skeleton (template) Java source code file with empty slots filled with the user-specified MA's properties. The Java code created is then programmatically

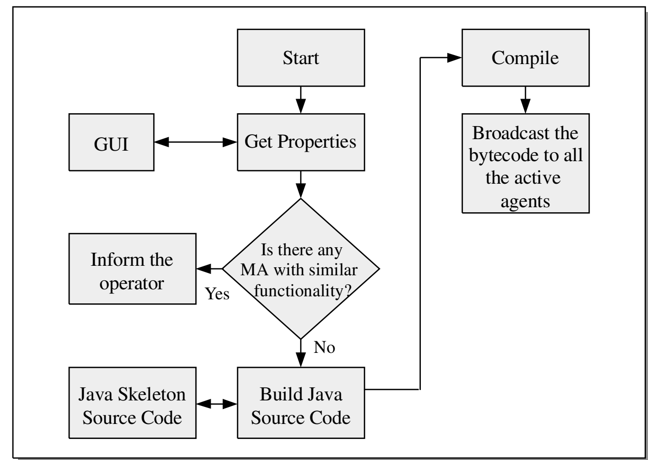


Fig. 5. Mobile agents generator functional diagram.

compiled and the generated Java bytecode transferred to selected managed devices (see Fig. 5).

The implementation details behind the MAG tool are given in Section 4.4.

3.9. Minimal usage of network resources for code deployment

A primary design objective of our framework is to employ policies that guarantee minimal usage of network resources for MA migrations. To this end we have implemented a lightweight migration scheme wherein the transfer of the MA bytecode is performed only *once*, at the MA construction time. From that point onwards the transfer of persistent state is sufficient for MAS entities to recognize the incoming MA and recover its state. In contrast, contemporary MAPs (with the exception of Puliafito et al., 2000) 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 (code size):(state size) typically lies in the range 10:1–15:1.

We expect the proposed MA code transfer approach to perform well in large-scale network environments, although whenever updating the configuration of a monitoring task, the corresponding MA class should be transferred to all monitored devices. This is because: (a) unlike existing MAPs, the transfer of the MA code will not be repeated; thereafter the transfer of the MA state will be enough for executing the monitoring task; (b) in many cases, updating the monitoring task's definition does not necessarily translate to code update, rather an update of the MA's state (i.e. modification of some non-transient variables) may be sufficient.

3.10. Reduced demand upon resource-constrained managed devices

The code deployment scheme described in Section 3.9, although relieving the network from unnecessary MA code transfers, it certainly implies increased demand upon the execution environments of distributed systems for MA code storage. Admittedly, some of the MA code definitions and their corresponding functionalities may be rarely used in some of the hosting systems. Although typical MA class definitions are of small size (1–2 kbytes), permanently storing these classes in resource-constrained managed systems might raise serious storage capacity issues.

To alleviate this drawback we have implemented two strategies: (a) the MA code is exclusively multicasted to those devices where monitoring tasks are to be performed; (b) to release the resources engaged to rarely used MA-based monitoring tasks, we

employ a LUFD (last used-first delete) strategy, wherein the last executed MA class is deleted from the local MA code repository to free some storage space (this 'garbage collection' is performed when the total space bound to MA definitions exceeds a device-specific threshold).

In case that an MA corresponding to a deleted class definition arrives, an approach similar to the one proposed in Puliafito et al. (2000) has been implemented: the code is downloaded on-the-fly from a remote code repository, residing at the manager station.

4. Infrastructure overview – implementation details

The following sections elaborate on the design and implementation details behind each one of the platform's components.

4.1. Manager application

The manager application performs monitoring and control operations through interacting with devices running agent processes. It comprises a multithreading environment where a main thread instantiates, controls and co-ordinates the operation of a number of specialized software components. These components are illustrated in Fig. 6 and analyzed in the following subsections.

4.1.1. Network discovery thread

At startup, the manager's network discovery thread (NDT) executes a topology discovery algorithm proposed in Lin et al. (1999) to populate a list of 'discovered' active MASs. First, the manager interacts with the DNS server to acquire all IP addresses of nodes hosted on the local domain and then 'pings' the corresponding nodes checking whether there is an active MAS thereon. It is noted that active servers are discovered either at manager initialization or whenever a new MAS starts operation. In the latter case, the manager application is notified and the host name is appended to that manager's 'discovered' list. The MAS finds out the ID of the manager host either through parsing a property file or through 'pinging' (at a specified TCP port) all hosts attached to its LAN. A managed host name may be also added manually by the administrator.

4.1.2. Itinerary scheduler module

The itinerary scheduler module (ISM) executes an algorithm that suggests the number *N* of MAs should be created to carry out this task and what their itinerary should be.

The ISM executes a Java implementation of the Heuristic algorithm for itinerary planning (HIP), which has been incorporated into the manager's ISM component. HIP is executed whenever a new multi-hop agent-based management task is about to start, suggesting not only the optimum number of MAs that should be dispatched but also their exact itinerary. It is noted that HIP's output is re-calculated whenever the managed network topology is altered, e.g. upon the event of a node's failure or the discovery of a new active agent server.

4.1.3. Polling threads

Polling threads (PTs) are started and controlled by the manager application; each of them corresponds to a single monitoring task. They properties of PTs (i.e. the configuration of monitoring tasks) are defined through the MAG tool (described in Section 4.4). The role of each PT is to instantiate a number *N* of MAs (suggested by the ISM) and assign them their itinerary, also provided by the ISM.

In the event of a pending MA-based task, the class definition of the MA corresponding to the management task is retrieved from the mobile code repository (MCR), a MySQL database where MA class files (bytecode) are stored in binary format.

Whenever the manager application shuts down, the PT properties are serialized and stored as polling thread configuration (PTC) objects ('configuration files'). Upon startup, those objects are deserialized and the associated PTs are resumed.

4.1.4. Migration facility component, mobile agent listener and security component

Upon starting/resuming a monitoring task the state information of the involved MA objects is compressed (using the Java gzip utility) and transferred through the migration facility component (MFC) to their first destination host.

When the MAs return back, they are received by the mobile agent listener (MAL) thread. The MAs are authenticated by the security component (SC) and their collected results are subsequently retrieved. The results are then presented to the user and optionally stored in a database. The MFC, MAL and SC components are basically identical to the ones used by the MAS entities; their implementation details are discussed in Section 4.3.

The results delivered by monitoring MAs may be displayed in real-time line graphs. Several classes of the Java-based AdvnetNet SNMP package (Adventnet) have been utilized for the implementation of these graphs. Fig. 7 presents screenshots of real-time graphs displaying the fluctuation of a MIB scalar variable value retrieved from a specific managed device and notification messages (alarms) issued upon the event of management parameters crossing predetermined thresholds.

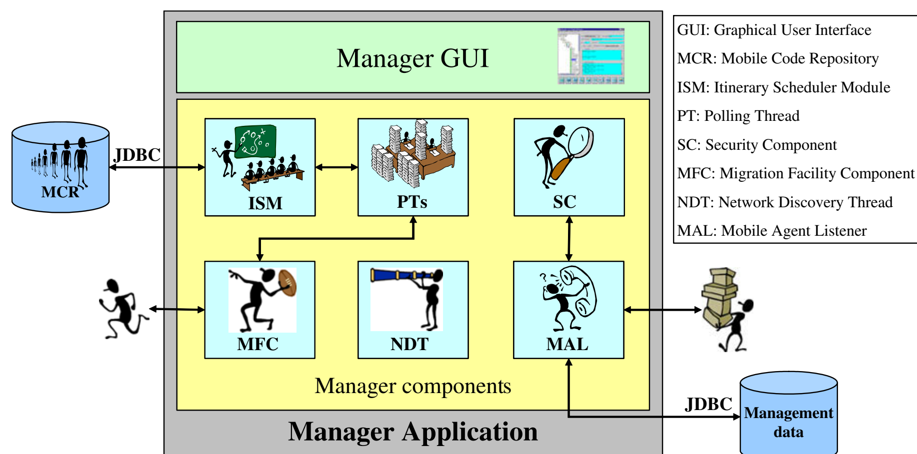


Fig. 6. Break-down of the components that compose the manager application.

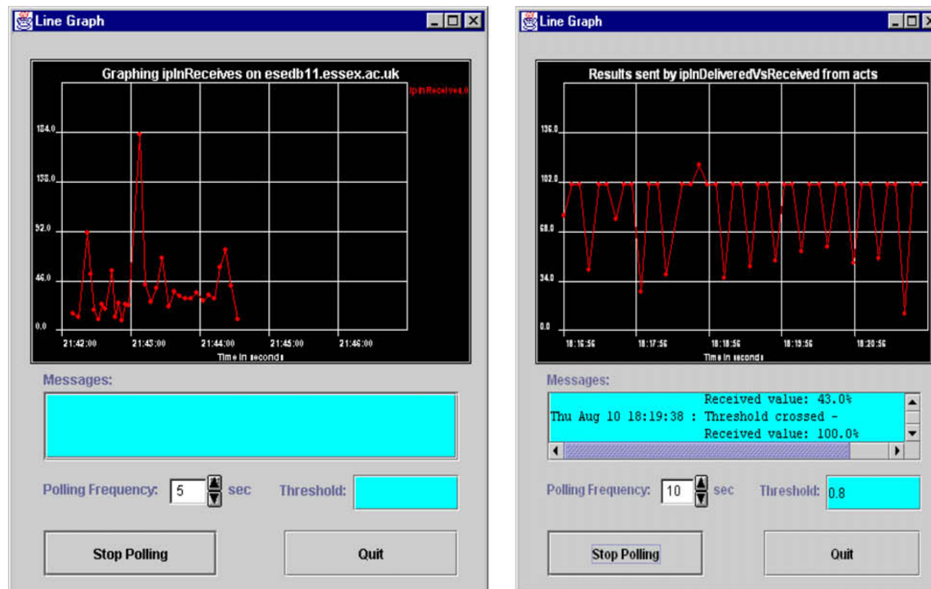


Fig. 7. (a) Polling of a scalar MIB variable (ipInReceives variable of MIB-II (McCloghrie and Rose, 1991)); (b) real-time notifications for management parameters (result of ipInDelivered/ ipInReceives) crossing pre-determined thresholds.

4.2. Mobile agent implementation

An agent is identified by its code (description of its behaviour) and persistent state information (modifiable variables). In the context of our MAP, MAs are Java classes supplied with: a unique ID, an 'itinerary folder' (supplied by the ISM), a 'data folder' (used to store collected data), a 'problems folder' (used to report faults), a flag indicating whether retrieved data should be encrypted, a byte array containing the MA's 'signature', a number of methods that facilitate the interaction with managed devices.

MA transfers are performed by the MFC components. In particular, through the process of *serialization*,[4] the state of an MA can be saved, transferred through the network and reconstructed (de-serialized) at the receiving node. 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_Authorized_To_Initialize_Exception exception is thrown.[5] Admittedly though, our platform offers only basic protection against malicious hosts; our research prototype design has not given emphasis on that particular security aspect. 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. A last set of callback methods are invoked when the MA is instantiated, arrives or moves from a host, fails to migrate, or when its execution is started, stopped, resumed or suspended.

Regarding its fault-tolerance features, our platform records as 'failure event' the failure to establish a connection among migration facility component of the source host and mobile agent listener of the destination host. This may occur as a result of a MAS failure and not failure of the managed device itself. In such case, the crashed MAS may be remotely recovered through the manager interface.

The fault-tolerance feature of our platform has been extensively tested through controlled fault injection experiments, e.g. by shutting down nodes' MASs, managed nodes and the manager station itself.

4.3. The mobile agent server (MAS)

The interface between visiting MAs and legacy systems is implemented through MAS modules.[6] Functionally, the MASs reside above standard SNMP agents, defining an efficient runtime environment for receiving, instantiating, executing, and dispatching incoming MAs, whilst protecting the system against malicious agent attacks. The SNMP agent process is started automatically at MAS initialization, if not already active.

Similarly to the manager application, the MAS class is designed as a multi-threaded entity. Apart from increasing the architecture's modularity, this scheme allows independent components to work concurrently and therefore decrease the overall delay when handling MAs, especially when the rate of incoming MAs is relatively high or bursty.[7] The main building blocks of a MAS are illustrated in Fig. 8 with their implementation detailed in the following subsections.

4.3.1. Mobile agent listener (MAL)

The MAL is a daemon, which listens for incoming MAs on a well-known port. Upon the arrival of an MA, its state is decompressed and de-serialized. 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 a certain priority level which equals a 'priority value' attribute carried

[4] The MA superclass (extended by all MA class definitions) implements the java.io.Serializable interface A Java class cannot be serialized unless it implements this interface, or it extends another class implementing the java.io.Serializable interface.

[5] The setParams() method of the MA super-class is invoked by the manager application that originally creates an MA object to set the MA's basic parameters (e.g. ID, itinerary, etc). This information cannot be later altered by any process. The same applies for the MA's signature, used for authentication purposes. In addition, a number of MA methods have been declared as 'protected', hence, they cannot be invoked by any other process.

[6] The installation of a MAS on every managed device since SNMP interactions between MAs hosted by a MAS and remote SNMP agents are also possible.

[7] In fact, the use of threads and concurrent servers for distributing the processing load and reducing the overall delay has been standard practice in the Internet for several years.

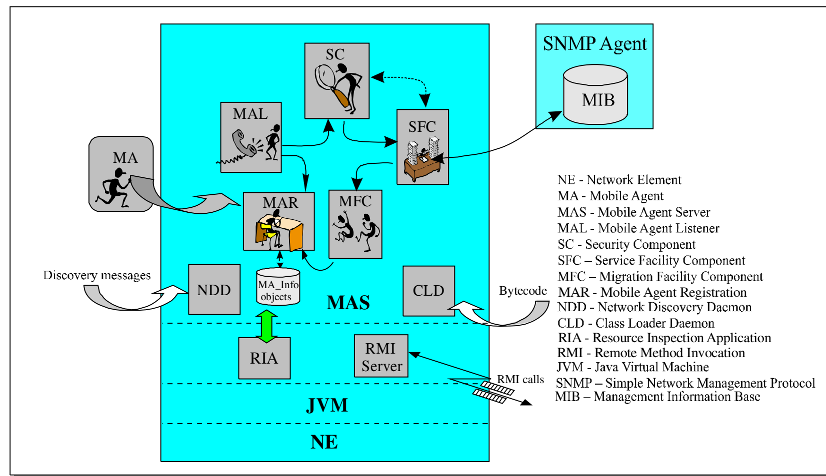


Fig. 8. The mobile agent server.

by the MA. That way, critical management tasks may be prioritized against other pending executing threads on the local device, to attain their timely execution.

The MAL then binds the MA to the local MAS 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. From that point onwards, the execution of MAL and MA threads is detached, with the first returning back to listening mode.

Of course the simultaneous arrival of many MAs cannot be prevented. However, no special programming techniques need to be used to synchronize the activity of several agents received simultaneously by a single MAL. The default operation of the MAL enables decompressing and de-serializing arriving MAs in a first-in, first-out order. If an MA arrives while the MAL is bound on decompressing another MA, the compressed state of the former MA will be kept within the temporary data buffer of the input stream.

4.3.2. Security component (SC)

This component acts as the system's protective barrier. First, the SC verifies the authenticity of the received MA through the use of security keys, ensuring that only trusted agents (dispatched by authorized hosts) are instantiated. In addition, it authorizes the actions performed by the locally executing MAs and ensures the privacy of sensitive management data returned to the manager station. The RSA (Rivest–Shamir–Adleman) algorithm (Rivest et al., 1978), based on the 'public–private pair of keys' paradigm, has been implemented providing both *authentication* and *encryption* features. It is noted that data encryption is performed only if requested by the administrator at the MA's creation time.

The security of MAS entities has been strengthened by introducing *authorization* features that restrict the authority domain of visiting MAs upon legacy systems. Specifically, the JVM's standard *security manager* (SM) has been extended to prevent MAs from directly accessing crucial system resources. 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.

Summarizing, our platform offers data encryption, as well as some basic protection measures against malicious hosts (tampering) and malicious agents: (a) a malicious agent will fail to pass the signature (authentication) test; (b) malicious MAs will be prevented to access legacy systems by the SM.

4.3.3. Mobile agent register (MAR)

MAS entities keep active control of agents executing on their local devices. In particular, the MAR component maintains a hashtable, using MA IDs as a primary key and including a list of "*MA_Info*" objects, each mapped to an MA running on the local host. MA_Info objects are basically records that include a reference to the MA they correspond to (allowing the MAS 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.

The information maintained by the MAR can be remotely obtained by the manager application through a remote method invocation (RMI) call and presented in the graphical user interface (GUI) (see Fig. 9). The administrator may then select any of the MA instances and perform a number of actions upon them, e.g. suspend, resume or activate an MA, modify the management task's polling frequency, etc.

4.3.4. 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 MAs to perform their decentralized management tasks.

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 MAs 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.

In a typical scenario, an MA passes an arbitrary number of object MIB OID strings to an SF method, which performs the SNMP query and returns the requested values. These values are subsequently processed, if necessary, by the MA. The value acquired,

Report on MAs running on esedb12.essex.ac.uk

Suspend | Resume | Deactivate | Activate | Dispose | Dispose All | Itinerary

ID	Agent Class Name	Pol. Mode	Pol. Frequency	Authentication	Encryption	Status
esedb6:4	tcpConnTableFiltering	GnS	15	☑	☐	active
esedb6:10	ifTablePolling	GnS	20	☑	☑	suspended
esedb6:36	ipDiscardedVsSent	GnS	30	☑	☐	active
esedb6:35	ifInErrorRate	GnS	25	☑	☑	active

Refresh | Close

Fig. 9. Online visual profiling of MAs executing on a network device.

either directly by the system or as a result of computation, is passed to the SC sub-system, encrypted and stored within the MA's data folder.

4.3.5. Migration facility component (MFC)

The role of the MFC is to dispatch upon request an MA to a specific network device. An MA transfer may be requested either by the MA itself or any other thread that holds a reference to the MA and the MFC, through calling the `move()` method of the MFC. The MFC is part of both the MAS servers and the manager application. In principle, its operation is the inverse of the MAL's operation. Namely, it establishes a network connection, compresses and serializes the MA state, with the resulted byte stream directed into the connection's output stream. Following the MA's migration, the MFC deletes the MA_Info object associated with to the dispatched MA by invoking the `delete()` method of the MAR component. It is noted that, in case of link or node failure in the process of an agent's migration, the fault tolerance mechanisms described in Section 4.2 take effect.

A diagram that illustrates the sequence of interactions among the MA and the various MAS components is shown in Fig. 10.

4.3.6. 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.

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 agents currently executing on them. It also allows the administrator to perform a number of actions upon the MAs (see Fig. 9).

4.3.7. Resource inspection application (RIA)

The RIA is an application developed in C programming language, which runs outside the boundary of the MAS. 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 applica-

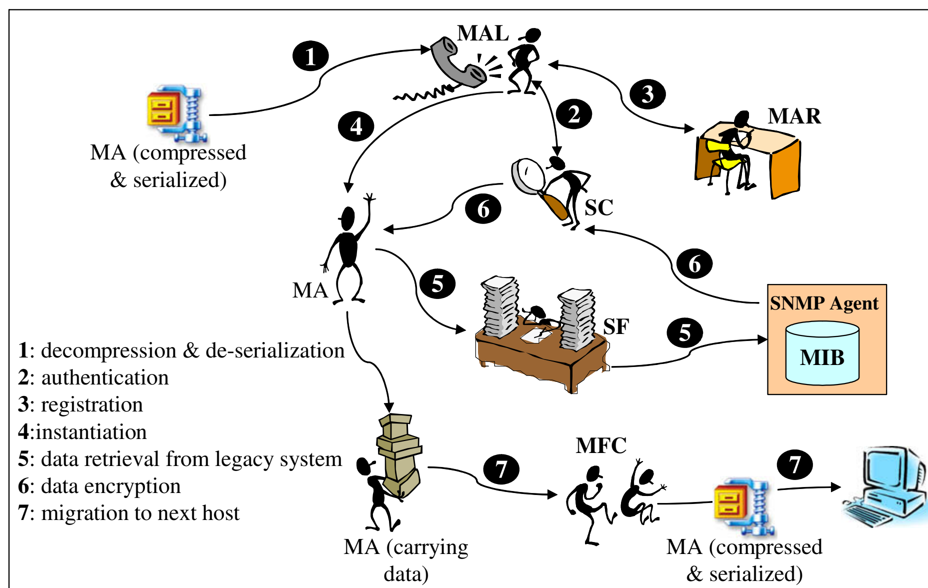


Fig. 10. Sequence of interactions among MAs and the components of MAS.

tions with programs written in other languages. The motivation behind RIA's development has been to allow the administrator and incoming MAs to obtain information regarding network devices load. Similarly to the RMI server, the installation and instantiation of the RIA is optional.

It is noted that the role of the RIA component is not to control important resources of underlying operating systems (e.g. threads, sockets creation, memory allocation, etc) but only to inform the manager application on the CPU and memory load of distributed systems; this information is given to the ISM component which prioritizes the inclusion of low-loaded systems in MA itineraries and relieves the heavily loaded systems from the execution of additional MA threads.

4.3.8. Network discovery daemon (NDD) and class loader daemon (CLD)

The NDD is a 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.

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.

Given the motivation to efficiently deal with the MA 'versioning' problem (see Section 3.7), we have extended the standard JVM implementing a custom CL, the mobile agent class loader (MACL). Unlike the CLs of existing MAPs,[8] MACL is able to identify different versions of the same MA class and enable the execution of the most recent version while disposing the old one.

The MACL class extends the default (java.lang.ClassLoader) CL class and overrides its loadClass() method. In order to force the MAL thread to load the incoming MA object using the customized MACL (and not the JVM's default CL), a new class (`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 customized MACL to load the latest version of the incoming MA definition.

4.4. Mobile agent generator (MAG)

The MAG is essentially a factory for constructing customized, service-oriented MAs. In the context of this article, generated MAs are designed to poll static management (SNMP) 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 startup time for this particular management task;
- 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.

The programmatic compilation of the Java template code (discussed in Section 3.8) is implemented through an invocation of the `compile`() method of `sun.tools.javac.Main` class (this class will be replaced by `JavaCompilerTool` tool in future JDK releases). The generated Java bytecode is subsequently compressed and transferred through TCP connections to all operating agent hosts (see Fig. 5). On the agent side, the CLD receives and decompresses the transmitted bytecode, validates the included Java class and stores it in a designated space. Thereafter, only the MA state is transferred at each migration, resulting in minimal usage of network resources.

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-specialized MAs.

5. Experimental evaluation

A critical omission frequently noticed in research papers introducing new MA-based frameworks for distributed management (e.g. (Liotta et al., 1998; Puliafito et al., 2000; Satoh, 2006)) is the experimental evaluation of the performance issues arising when implementations of the corresponding models are used in real networking environments. To thoroughly evaluate the performance of our proposed MAP, response time and the network overhead experiments related to management operations have been conducted. Admittedly, our experimental work lacks comparison of our MAP against alternative solutions. This is because, first, a comparison against general-purpose MAPs would be unfair and pointless since those platforms lack of essential features for management applications support and, second, no NSM-oriented MAP is currently publicly available.

5.1. Response time experiments

The time needed to complete a number of network performance management operations is of critical importance when considering real-time management operations. This part of performance evaluation work involves a simple experiment wherein the round-trip delay of multi-hop MAs visiting a set of managed devices is measured. At each node, MAs accumulate management data of variable length finally delivered to the manager station. No data compression is performed to ensure that the increment on the amount of encapsulated data coincides with the MA's state size increment. We have examined two scenarios: the first involves a single MA that visits the entire set of managed devices in arbitrary order; in the second, we have enabled the execution of the HIP algorithm (by the manager's ISM component) to suggest an optimum number of MAs (and their corresponding itinerary) depending on the network size and the volume of accumulated data. Our experimental test bed includes 11 hosts (Windows 2000 operating system, CPU 1,2 GHz, 256 MB RAM), all included in the same 100 Mps Ethernet segment; the 10 of them act the role of the managed device and host a MAS while the last hosts the manager application.

[8] The only exception is the Aglets MAP, which allocates different CLs to different sets of classes, with a new CL created for each updated version of an MA class (Mobile Agent Platform (IBM)). The distinction between the updated and older versions of an MA is achieved by sending information about thenames and versions of classes along with the classes' bytecode. 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.

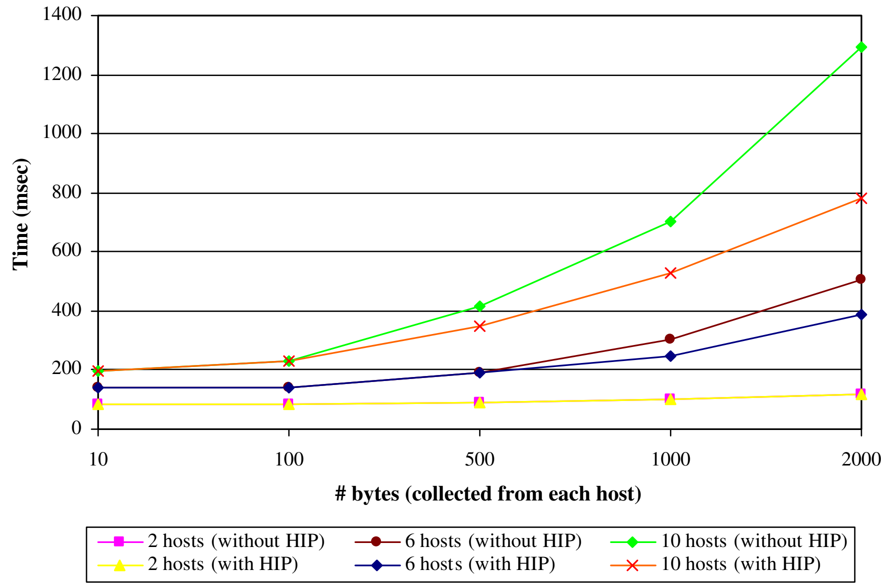


Fig. 11. Response time for multi-hop MAs as a function of the amount of encapsulated information.

The results shown in Fig. 11 indicates an exponential growth of the real overall response time measured when engaging a single MA to visit all our test bed devices (the overall network overhead increases exponentially with the amount of collected data, hence the transmission delay increases likewise). On the other hand, when executing the HIP algorithm, more than one MAs are typically launched when the network size and the amount of encapsulated data grows. As expected, the parallel employment of MAs decreases the overall latency. In general, mainly due to its lightweight MA code distribution scheme and the incorporated MA migration optimization techniques our MAP ensures the timely completion of monitoring tasks. The 90% confidence interval of the measurements shown in Fig. 11 are between $(\bar{x}-3.5;\bar{x}+3.5)$ for 2 hosts (without HIP, 2000 bytes) and $(\bar{x}-12;\bar{x}+12)$ for 10 hosts (without HIP, 2000 bytes).

The results drawn in Fig. 11 refer to response time measurements taken while evaluating our MAP prototype in our experimental test bed. Admittedly though, the latency measurements taken in a test bed comprising 11 hosts in a Gigabit LAN, cannot serve as a reliable 'hypothesis' for evaluating the performance of our framework in realistic large-scale environments. In such environments, the manager station typically retrieves data from dozens of managed devices spread in geographically dispersed locations; hence, agent migrations are associated with prolonged transmission delays affecting the overall latency.

To evaluate the performance of our MAP in such environments, we have conducted a second set of latency experiments emulating the network conditions among hosts (see Fig. 12), although still using the original – 11 hosts in a Gigabit LAN-test bed. In particular, travelling MAs are forced to delay their migrations for a random time period following a normal distribution (mean $\mu = 0.5$ s, variance $\sigma = 2$ s), emulating the prolonged transmission delays experienced on real networking environments. The curves appearing in Fig. 12 are no longer exponential since the transmission de-

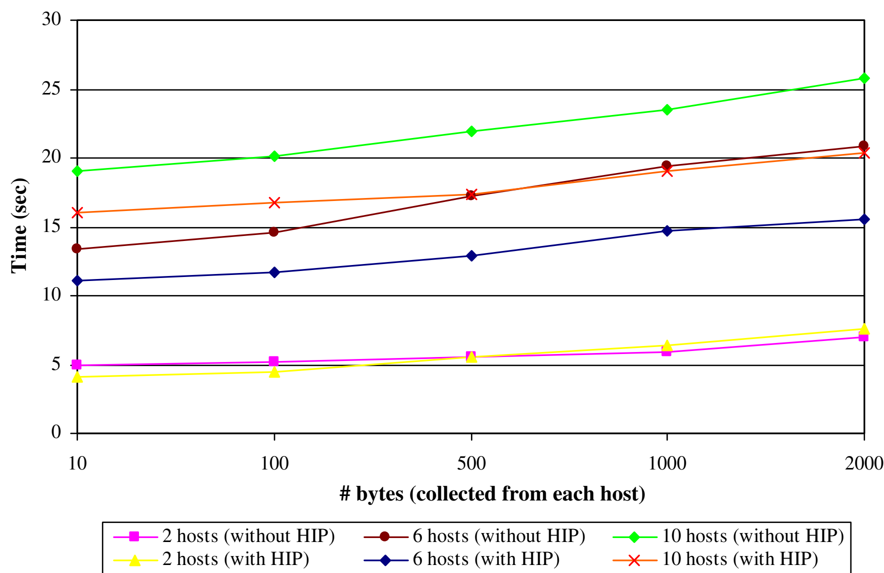


Fig. 12. Response time for multi-hop MAs in emulated realistic networking environments.

lay dominates over the other latency components. As expected, the overall latency is drastically increased and the prominence of the solutions involving the execution of the HIP algorithm is evident.

5.2. Network overhead measurements

An important scalability parameter that signifies the appropriateness of a model on NSM 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. (Bohoris et al., 2000)) supplement design ideas and implementations with real traffic measurement data.

Similarly to the response time experiments, our objective has been to measure the overall network overhead incurred when multi-hop MAs perform distributed monitoring tasks upon a group of nodes. The initial state size of the MA used in this experiment is 400 bytes. Similarly to the response time experiment, the aim is to investigate the effect of optimal itinerary design (execution of the HIP algorithm) on the management cost (in the context of this article, the term 'management cost' refers to network overhead). As shown in Fig. 13 the management cost increases exponentially with the amount of encapsulated information when employing a single MA to visit all devices. In contrast, the execution of the HIP algorithm dictates the use of a larger number of MAs as the volume of encapsulated data and the network size grows. For instance, when considering 10 managed devices, 5 MAs are employed in parallel when the amount of data becomes 2000 bytes (each MA will only visit two devices). Namely, the cost gain of using HIP results from the fact that multiple MA instances are activated in parallel, instead of sequentially, choosing a more intelligent path. It is noted that network overhead measurements have been performed with the Ethereal network protocol analyzer, using appropriate capturing filters.

The overall polling overhead highly depends on the increment rate of the MAs' state size, which in turn is a function of "*selectivity*" σ ($0 \leqslant \sigma \leqslant 1$), a metric defined in Liotta et al. (1998) as the proportion of data *maintained* to that *retrieved*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 con-

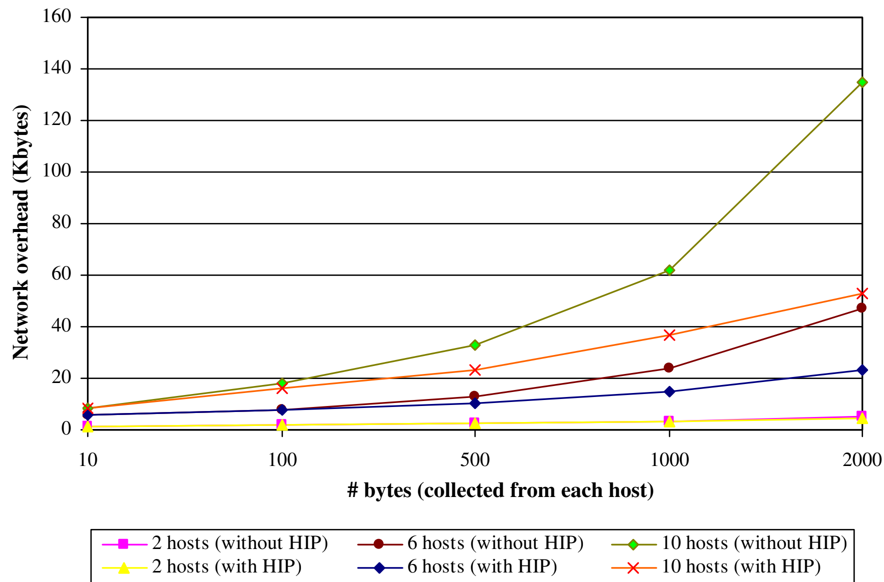


Fig. 13. Total network overhead (per polling interval) for multi-hop MAs as a function of the amount of encapsulated.

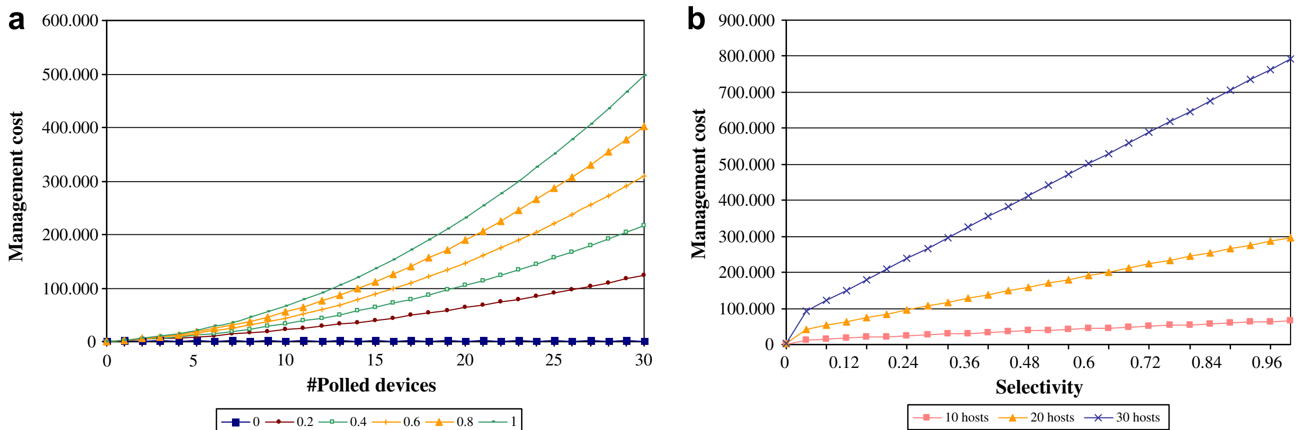


Fig. 14. Management cost (per polling interval) of MA-based polling as a function of: (a) the network size for various selectivity values; (b) selectivity for various network sizes.

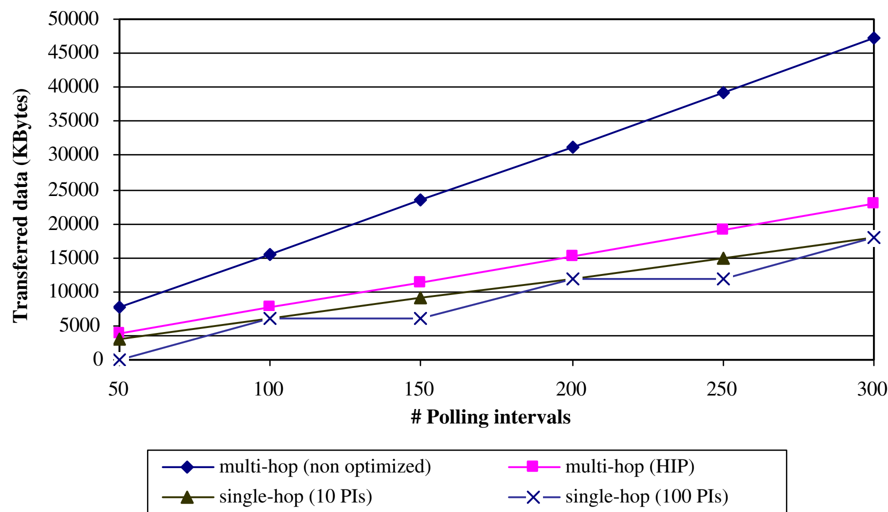


Fig. 15. Total network overhead for single-hop vs. multi-hop MAs as a function of polling intervals.

stant, otherwise the state rapidly grows. A second set of measurements aims at investigating the effect of selectivity and network size upon the total network overhead. As demonstrated in Fig. 14a, MAs state size increases exponentially as a function of the hop count (number of visited polled devices). In particular, it is shown how the management cost per polling interval varies as a function of the network size for various selectivity values (the initial MA state size is 400 bytes, the amount of retrieved data is 1000 bytes, *no* MA state compression is performed). On the contrary, management cost increases linearly with selectivity (see Fig. 14b). It is noted that the results corresponding to more than 10 managed devices have been extrapolated.

A last experiment comprises a performance comparison among multi-hop versus single-hop agent performing monitoring tasks (we consider 10 managed devices and data samples of 2000 bytes), where no processing of retrieved management information takes place. For multi-hop MA-based polling we have measured the overhead for both arbitrary and optimized (execution of HIP algorithm) itineraries. For single-hop MA-based polling, 10 MAs migrate to each individual host and remain there delivering retrieved data through RMI invocations, either every 10 or 100 polling intervals (i.e. 10 or 100 data samples respectively are returned to the manager with a single RMI call).

The results are presented in Fig. 15. As expected, the results demonstrate an advantage of single-hop scheme, since MA migrations occur only when the monitoring task is initiated. Multi-hop MAs with non-optimized itineraries clearly exhibit the worse performance amongst all alternative approaches. Extra savings on single-hop scheme's overhead should be expected in the case that management information is locally processed (e.g. when MAs monitor particular nodes' health parameters and notify the manager only when these parameters cross pre-determined thresholds). On the other hand, network monitoring based on multi-hop MAs is advantageous for short-term monitoring tasks (in such tasks it would be time consuming to employ single-hop mobility, i.e. to upload the agent code to all monitored devices) and also in cases where a global (domain)-level rather than a local (device)-level view of managed resources is required (Liotta et al., 2002).

6. Simulation results in large-scale networking environments

Our platform has also been evaluated in simulated large-scale network environments, using the network simulator (NS-2) tool (Simulator - NS-2), a discrete event simulator targeted at networking research. The application scenario involves polling managed elements for the contents of tcpConnTable MIB-II[9] table, which lists information about all TCP connections of a host. In the SNMP-based implementation, individual MIB tables are remotely retrieved through exchanging request/response messages, in particular by issuing successive get-next requests (each retrieving a table row). Every tcpConnTable row contains five values (columns) (McCloghrie and Rose, 1991), while managed elements store information about *40* TCP connections on average, i.e. included 40 table rows (40×5 values in total).

We have implemented SNMP and mobile agent modules. The SNMP packets and MA size parameters measured though the Ethereal packet sniffer (Ethereal network protocol analyzer) have been incorporated into the implemented modules to enhance simulation results validity and accuracy. In particular, the MA initial size is 1.08 KB and the MA state size increment per table sample is 98 bytes; the SNMP request/response packet size is 90 bytes and the average SNMP packet increment for each additional requested value is 17 bytes; the average SNMP-based and MA-based table retrieval response times are 102.1 ms and 71 ms, respectively.

Three different transit-stub topologies created by the topology generator GT-ITM (Zegura et al., 1997) are used (the GT-ITM output has been first converted to NS-2 format). The topologies comprise 272 nodes with interconnecting links of 2 Mbps bandwidth and latency of few milliseconds; 100 Mbps links are assumed within stub boundaries (see Fig. 16). The management station controls 15 stub domains each hosting 16 network elements (240 managed elements in total). Several simulations tests (SNMP-based and three variations of MA-based network monitoring scenarios) have been performed and several measurements have been taken and evaluated.

In the SNMP implementation, monitoring is performed in the following way: first, all elements of a stub are accessed; then, the next stub is managed until all the 16 stubs are accessed. In

[9] A management information base (MIB) is a formal description of the set of network objects that can be managed using SNMP. MIB-II (Magedanz et al., 1996) Magedanz, Rothermel, Krause, "Intelligent agents: an emerging technology for next generation telecommunications ?", Proceedings of IEEE INFOCOM'96, pp. 464–472, March 1996. McCloghrie and Rose (1991) is the standard MIB in the IP world, supported by virtually all SNMP-compliant network devices.

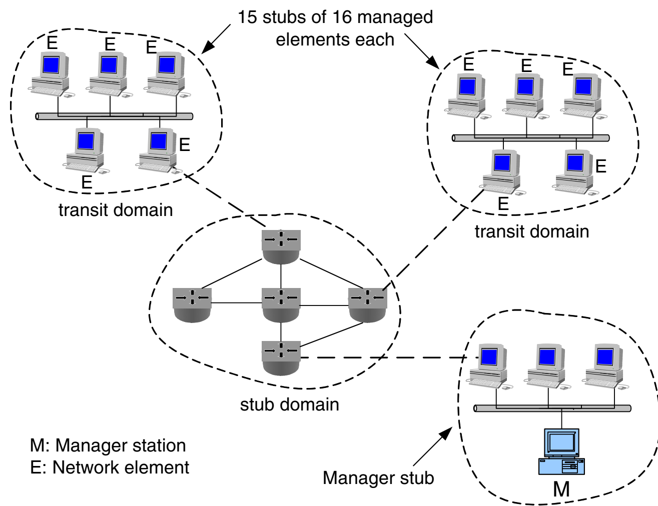


Fig. 16. Network monitoring application on a transit-stub topology.

the MA-based implementation we simulated three different alternatives:

- monitoring 'with HIP', which involves the parallel execution of multiple MAs, each supplied with a 'near-optimal' itinerary; the number of monitoring MAs depends on the specific topological characteristics (this is the approach taken in our MAP),
- monitoring 'without HIP', wherein HIP algorithm is turned off, i.e. a unique MA object visits sequentially all monitored devices (carrying its state), interacts with local SNMP agents and stores collected management data within its state,
- monitoring 'without code upload', wherein a unique MA object visits sequentially all monitored devices, carrying *both* its code and state, i.e. the MA code is not uploaded to all monitored devices prior to executing the monitoring task, as suggested in Section 4.4 (in effect, this is the approach taken in all existing MAPs).

Simulation results are reported in Fig. 17, where various curves represent the mean response time or bandwidth consumption for the three simulated topologies. Fig. 17a presents mean response times for various managed network sizes. For a few sets of managed elements, SNMP performs better than HIP-based approach due to its lightweight nature and the fact that all the elements of a specific stub are polled in parallel. As the number of managed elements increases, SNMP response time grows proportionally since the time to manage a stub is approximately the same for all stubs. In the 'without HIP' approach, the response time increases faster when the number of managed elements grows due to the incremental rate of the MA's state size which implies increased transmission delays. HIP presents better scalability as monitored elements are separated in 'virtual' managed domains, each managed by separate MA objects collecting management data in parallel. The monitoring 'without code upload' performs worse since the unique MA carries its code in addition to its state on every migration. Admittedly though, latency measurements may differ when deploying our platform in real large-scale environments where different networking conditions may apply.

Fig. 17b shows that the network overhead generated by our MAP (with HIP) and SNMP is approximately proportional to managed network size, with our platform achieving significantly better results. On the contrary, when disabling HIP the scalability becomes very poor (as the number of elements visited by the unique object increases, the amount of collected data becomes much larger than the size of MA code itself).

A comparison of the real management cost associated with monitoring operations (the cost of using the relatively expensive 2 Mbps interconnecting links is taken into account) is given in Fig. 17c where the performance gain of HIP-based approach is amplified. The drawbacks of the SNMP centralized approach be-

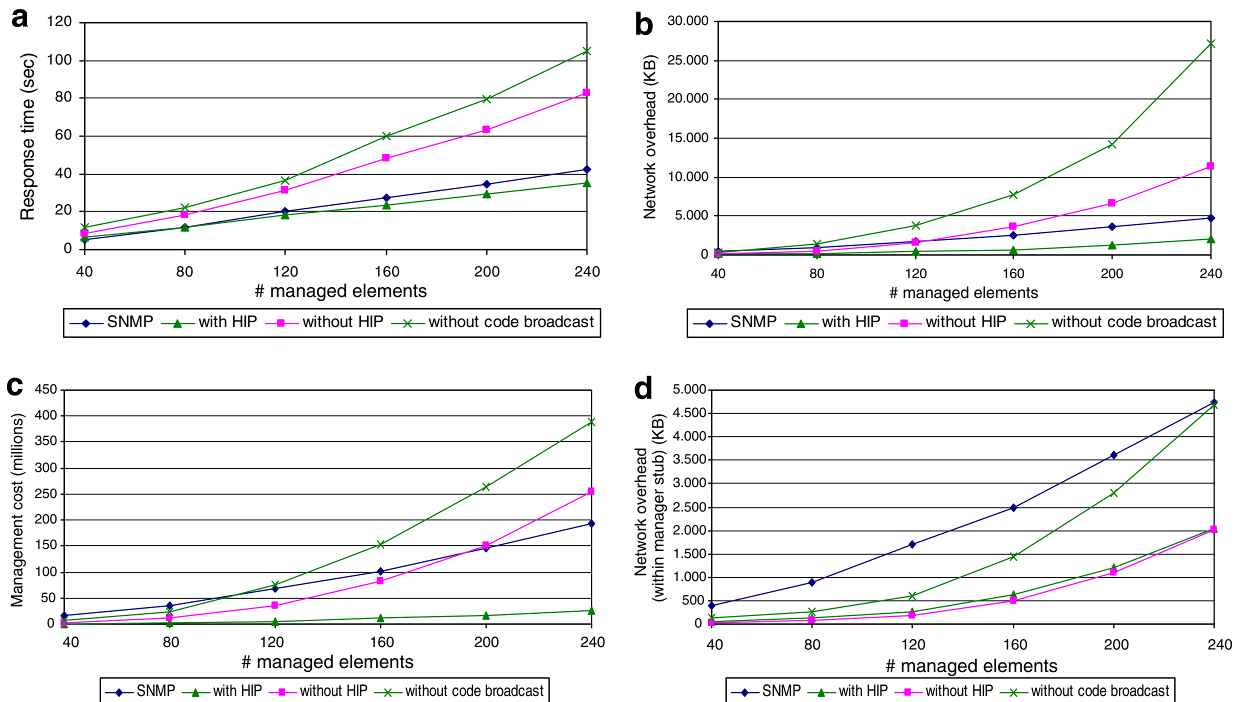


Fig. 17. Monitoring of `tcpConnTable` MIB-II variable: comparison of SNMP against the proposed MAP (with HIP, without HIP and without code upload) in terms of: (a) overall latency, (b) network overhead, (c) management cost, (d) network overhead within manager station stub.

comes evident as it indiscriminately makes heavy use of relatively expensive network resources.

Finally, Fig. 17d illustrates the usage of network resources within manager station stub. Interestingly, this metric represents the main asset of the 'without HIP' solution, since it suggests that the unique monitoring MA will use manager stub link only twice: on its first migration and at the end of its itinerary (when it returns to deliver collected data to the manager station). On the other hand, SNMP routes all monitoring traffic through the manager stub links, resulting in a congestion point.

7. Conclusions and future work

A critical issue that will affect the wider adoption of MA paradigm in management applications is the development of MAPs expressly oriented to distributed NSM. These platforms should not only provide an programmable environment for performing decentralized management tasks but also to satisfy a number of design requirements in terms of flexibility, functionality, security and demand on systems and network resources.

In this paper, we introduced the design decisions and implementation aspects of a complete MAP research prototype that sufficiently addresses all the aforementioned concerns. In comparison to existing approaches, our MAP consolidates several novel design features, dictated by our design choices and reflected upon our research prototype implementation: (a) a lightweight code distribution scheme, (b) a class loading mechanism that allows the modification of MA-based NSM tasks at runtime, (c) a tool that supports the user-friendly customization of service-oriented MAs, (d) a component that builds near-optimal network-dependent MA itineraries. In addition, it satisfies other important NSM-related requirements, such as lightweight footprint on systems resources, security (authorization, authentication and encryption), recovery from basic software faults, modularity, incorporation of agent migration optimization techniques, platform-independence, etc. Our MAP's performance has been evaluated in realistic management application scenarios.

The design and development team of our platform has gained many experiences and received valuable lessons the course of our research:

- several design decisions had to be made and the respective trade-offs to be carefully evaluated; for instance, the decisions related to the design of the itinerary optimizer tool, the agent migration and code deployment scheme, the class loader dealing with the 'versioning' problem, the method for reducing the demand upon resource-constrained managed devices, etc.
- the design of an impenetrable security mechanism that deals with any malicious host or agent attack is particularly complex, in fact it remains an open research area;
- the development of the core MAP functionality (MA entities, migration facility, agent server components, interface with legacy systems, fault tolerance, information storage in databases, etc) proved relatively effortless for an experienced Java programmer, as opposed to the implementation of the class loading mechanism and the optimal itinerary scheduling algorithm;
- the creation of new monitoring tasks through the MAG tool has been found an easy-to-do task, even by inexperienced users;
- professional network managers pointed out that a web interface for the manager application would very much assist their job.

Future research will address the following issues: (a) performance comparison in terms of latency and network overhead between our proposed MAP, general-purpose MAPs and NSM-oriented MAP (when they become publicly available); (b) extensive testing and practical evaluation of our MAP in real-world monitoring applications by research engineers in order to identify potential deficiencies and derive methods for improving the MAP's performance; (c) extension of the manager platform with the implementation of a web interface that will allow human administrators to view updated management statistics through a typical web browser and remotely control their network; (d) implementation of a light manager application that will be hosted to resource-constrained mobile devices; the manager will interact with managed devices through a wireless application protocol (WAP) or i-mode interface.

Acknowledgements

The authors are indebted to Kristen Pol for his valuable tips and insightful ideas during the implementation of the custom class loader. Paolo Bellavista is acknowledged for his invaluable help throughout the development of the Resource Inspection Application module.

The authors also wish to thank the anonymous reviewers for their constructive comments which significantly improved the technical content and the presentation of the paper.

References

Adventnet, <http://www.adventnet.com>.
Aglets Mobile Agent Platform (IBM), <http://www.trl.ibm.com/aglets/>.
Bee-gent Multi-Agent Framework (Toshiba), <http://www2.toshiba.co.jp/rdc/beegent/index.htm>.
Bohoris, C., Liotta, A., Pavlou, G., 2000. Evaluation of constrained mobility for programmability in network management. In: Proceedings of the 11th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM'2000), December 2000, pp. 243–257.
Bäumer, C., Magedanz, T., 1999. Grasshopper – a mobile agent platform for active telecommunication. In: Proceedings of the 3rd International Workshop on Intelligent Agents for Telecommunication Applications (IATA'99), 1999, pp. 19–32.
Du, T., Li, E., Chang, A.P., 2003. Mobile agents in distributed network management. Communications of the ACM 46 (7). July.
Eid, M., Artail, H., Kayssi, A., Chehab, A., 2005. Trends in mobile agent applications. Journal of Research and Practice in Information Technology 37 (4), 323–351. November.
Ethereal network protocol analyzer, <http://www.ethereal.com/>.
Franklin, S., Graesser, A., 1997. Is it an agent, or just a program? a taxonomy for autonomous agents. In: Mueller, J. (Ed.), Intelligent Agents III; Agent Theories, Architectures, and Languages. Springer, Berlin.
Fuggeta, A., Picco, G.P., Vigna, G., 1998. Understanding code mobility. IEEE Transactions on Software Engineering 24 (5), 346–361.
Gavalas, D., 2004. Mobile agent platform design optimisations for minimising network overhead and latency in agent migrations. In: Proceedings of the 2004 IEEE Global Communications Conference (Globecom'2004), December 2004.
Gavalas, D., 2006. A lightweight and flexible mobile agent platform tailored to management applications. In: Proceedings of the 2006 Conference on Mobile Computing and Wireless Communications (MCWC'06), September 2006.
Gavalas, D., Politi, C., 2006. Low-cost itineraries for multi-hop agents designed for scalable monitoring of multiple subnets. Computer Networks, Elsevier Science 50 (16), 2937–2952. November.
Java 2 Platform, Standard Edition (J2SE), <http://java.sun.com/j2se/>.
Kendall, E.A., Krishna, P.V.M., Pathak, C.V., Suresh, C.B., 1998. Patterns of intelligent and mobile agents. In: Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98), May 1998, pp. 92–99.
Krause, S., Magedanz, T., 1996. Mobile service agents enabling intelligence on demand in telecommunications. Proceedings of IEEE GLOBCOM'96, 78–84. November.
Jiang, Y.C., Xia, Z.Y., Zhong, Y.P., Zhang, S.Y., 2004. Defend mobile agent against malicious hosts in migration itineraries. Microprocessors and Microsystems 28 (10), 531–546.
Lin, H.C., Lai, H.L., Lai, S.C., 1999. Automatic link layer topology discovery of IP networks. Proceedings of the IEEE International Conference on Communications (ICC'99). June.
Liotta, A., Knight, G., Pavlou, G., 1998. Modelling network and system monitoring over the internet with mobile agents. In: Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS'98), February 1998, pp. 303–312.
Liotta, A., Pavlou, G., Knight, G., 2002. Exploiting agent mobility for large scale network monitoring. IEEE Network 16 (3), 7–15. May/June.

Magedanz, T., Rothermel, K., Krause, S., 1996. Intelligent agents: an emerging technology for next generation telecommunications? In: Proceedings of IEEE INFOCOM'96, March 1996, pp. 464–472.
McCloghrie, K., Rose, M., 1991. Management information base for network management of TCP/IP-based internets: MIB-II. RFC 1213. March.
Manvi, S.S., Venkataram, P., 2006. Agent based subsystem for multimedia communications. IEE Proceedings Software Journal 153 (1), 38–48.
Network Simulator – NS-2, <http://www.isi.edu/nsnam/ns/>.
Pagurek, B., Wang, Y., White, T., 2000. Integration of mobile agents with SNMP: Why and how. In: Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS'2000), April 2000, pp. 609–622.
Present Technologies, JAMES project (Java Mobile Agent Platform for the Management of Telecommunication and Data Networks), <http://www.present-technologies.com/james.jsp>.
Puliafito, A., Tomarchio, O., Vita, L., 2000. MAP: design and implementation of a mobile agents platform. Journal of System Architecture 46 (2), 145–162. January.
Randell, B., 1975. System structure for software fault tolerance. IEEE Transactions on Software Engineering Se-1(2). June.
Rivest, R.L., Shamir, A., Adleman, L., 1978. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21 (2). February.
Rubinstein, M.G., Duarte, O.C., Pujolle, G., 2003. Scalability of a mobile agents based network management application. Journal of Communications and Networks 5 (3). September.
Satoh, I., 2006. Building and selecting mobile agents for network management. International Journal of Network and Systems Management 14 (1), 147–169. March.
Stallings, W., 1999. SNMP, SNMPv2, SNMPv3 and RMON 1 and 2, third ed. Addison Wesley.
Stephan, R., Ray, P., Paramesh, N., 2004. Network management platform based on mobile agents. International Journal of Network Management 14, 59–73.
SysteMATech – System Management based on Mobile Agent Technology, <http://www.eutist-ami.org/more_systematech.asp>.
To, H.H., Krishnaswamy, S., Srinivasan, B., 2005. Mobile agents for network management: when and when not!. In: 2005 ACM Symposium on Applied Computing (SAC'05), March 2005, pp. 47–53.
Venners, B., 1998. Designing with dynamic extension: how dynamic extension works in Java and how to use it in your designs, December 1998, <http://www.artima.com/designtechniques/dynaext.html>.
Whitestein Technologies, <http://www.whitestein.com/pages/index.html>.
Zegura, E.W., Calvert, K.L., Donahoo, M.J., 1997. A quantitative comparison of graph-based models for internet topology. IEEE/ACM Transactions on Networking 5 (6), 770–783. December.

Damianos Gavalas received his B.Sc. degree in Informatics (Computer Science) from the University of Athens, Greece, in 1995 and his M.Sc. and Ph.D. degrees in electronic engineering from University of Essex, UK in 1997 and 2001, respectively. Currently, he is an Assistant Professor in the Department of Cultural Technology and Communication, University of the Aegean, Greece. He has served as a TPC member in several leading conferences in the field of mobile and wireless communications. He has co-authored over 70 papers published in international journals, conference proceedings and book chapters. His research interests include distributed computing, mobile code, network and systems management, mobile computing & multimedia, m-commerce, mobile ad hoc & sensor networks.

George E. Tsekouras received the B.S. and Ph.D. degrees from the Department of Chemical Engineering-National Technical University of Athens in 1994 and 2000, respectively. For several years he worked as a control engineer in petrochemical industries. From 2002 until now he is with the department of Cultural Technology and Communication in the University of the Aegean, Greece. His research interests include computational intelligence and its applications in system modelling, statistical data analysis, image and signal processing and software engineering.

Christos-Nikolaos E. Anagnostopoulos was born in Athens, Greece in 1975. He received his Mechanical Engineering Diploma from the National Technical University of Athens (NTUA) in 1998, and the Ph.D. degree from the Electrical and Computer Engineering Department, NTUA in 2002. Currently, he serves as assistant professor in the Cultural Technology and Communication Department, University of the Aegean. Dr. Christos–Nikolaos E. Anagnostopoulos is a member of the Greek chamber of Engineers and member of IEEE. His research interests are image processing, computer vision, neural networks and artificial intelligence. He has published more than 60 papers in journals and conferences, in the above subjects.