CHAPTER 6

ADAPTIVE HIERARCHICAL MANAGEMENT

6.1. INTRODUCTION

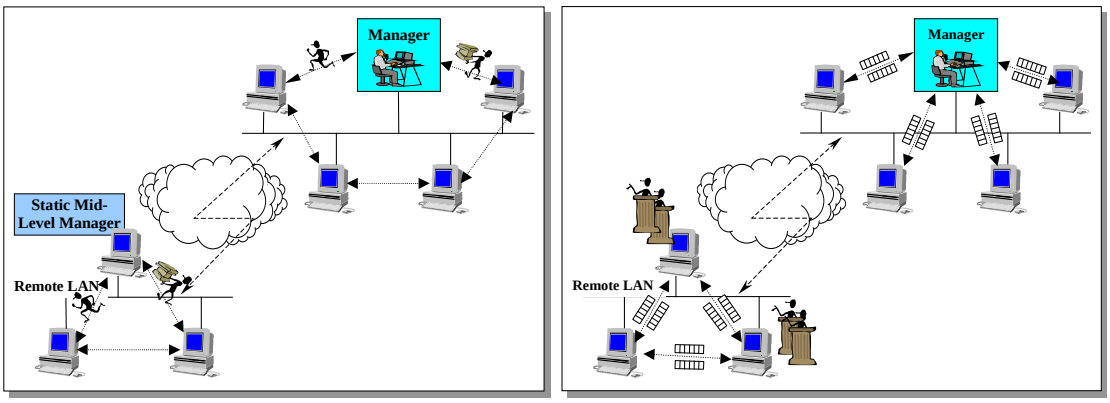
Contemporary large enterprise networks follow a hierarchical structure, spanning applications, organisational and geographical boundaries. In order to cope sufficiently with the unpredictable growth of the number of network devices, logical partitioning is being employed as a design and deployment principle. Any of the following or a combination of these partitioning criteria can be used: (a) physical distribution, i.e. partitioning based on the 'geographical' location of network elements (NE); (b) administrative subdivisions; (c) grouping based on different access privileges and security policies; (d) performance-driven network partitioning, e.g. aiming at evenly distributing traffic load among network segments. The design of such networks' management systems cannot escape that rule: the resultant network clusters may be better managed by hierarchically structured management models.

However, there is a notable inconsistency between the topology of hierarchically structured networks and the organisation models of emerging Mobile Agent (MA)-based management frameworks, the majority of which insists on 'flat' architectures. That approach fails to solve the scalability limitations of centralised architectures as it has been shown to result in increased response time and network overhead (see Section 4.5). The situation seriously deteriorates when considering management of remote LANs, connected to the backbone network through low-bandwidth, expensive WAN links. In this case, frequent MA transfers are likely to create bottlenecks and affect the overall management cost.

The response time part of flat management scalability problem is addressed by the 'segmentation' approach, i.e. Get 'n' Go (GnG) polling, introduced in Section 5.2.1; this scheme enables the parallel and therefore more time efficient data collection. The network overhead problem is more efficiently addressed by the 'broadcast' approach, i.e. Go 'n' Stay

(GnS) polling, described in Section 5.2.2; this approach is not however suitable for real-time operations or monitoring tasks whose execution is planned for short-time periods.

Geographically dispersed enterprise networks are more efficiently managed by hierarchical management systems (presented in Sections 2.10 and 2.11) which obviate the need for centralised Simple Network Management Protocol (SNMP) polling and localise management traffic through mid-level manager (MLM) entities acting on behalf of the central manager. However, existing hierarchical management solutions imply an inflexible and static definition regarding the physical location where MLMs execute and the assignment of the managed devices under their supervision (see Section 2.12). In other words, they are only suitable for static network topologies but are not in step with the dynamically evolving topological and traffic characteristics of large-scale enterprise networks. MA technology could provide the means for enriching the functionality and improving the flexibility of hierarchical management [MOU98a].



(a) (b)

Figure 6.1. Approaches to hierarchical MA-based management

The concept of hierarchical MA-based management is not entirely new. In particular, two approaches have been presented so far: (a) Use of static MLMs relying on MAs for the network monitoring process [LIO98, SAH98] (see Figure 6.1a); (b) Use of mobile MLMs performing decentralised SNMP management [ZAP99] (see Figure 6.1b). The first approach does not adequately address the flexibility limitations of proprietary hierarchical management. Still, even the latter approach lacks mechanisms for achieving automatic adaptation of the management system to changing network configurations, i.e. MLMs do not normally change the location where they execute. This model also involves the deployment of a new MA for each introduced monitoring task; these MAs work independently and do not necessarily execute at the same host. However, this approach is not in line with the concept of a *compact* MLM entity responsible for *all* the decentralised operations performed within its domain, which in our opinion offers better grouping, organisation, manageability and control over distributed Network & Systems Management (NSM) tasks.

The aforementioned approaches have certainly realised a step forward, yet, their direct application to distributed management is not straightforward. Critical issues such as well-defined criteria for segmenting the network into management domains, explicit determination of the domain boundaries or strategies for assigning mid-level managers to these domains, are not elucidated. Although the problems that need to be solved have been identified, the rules that define the mid-level entities' deployment strategy, i.e. questions concerning *when* and *where* to deploy, remove or change the location of mobile mid-level managers still remain open. Furthermore, it is important to use MAs both as MLMs and for the actual monitoring process in order to take advantage of their ability to filter management information at the source. The development of a highly adaptive hierarchically structured MA-based management model seems a rational approach to address these issues as well as to overcome the limitations of statically configured NSM frameworks.

This chapter addresses these issues through introducing a highly scalable and adaptive hierarchical management model. Such a model presupposes the presence of a novel management element, termed the *Mobile Distributed Manager* (MDM), operating at an intermediary level between the manager and the stationary agents. MDMs are essentially MAs, which take full control of managing a specific network domain and localise the associated management traffic. Apart from the fact that management functionality may be added/configured at runtime, this architecture can also dynamically adapt to fluctuating networking conditions. The system's scalability is further improved by assigning monitoring tasks to additional MAs (launched and controlled by the MDM) capable of filtering collected data locally. MDMs are deployed to remote subnets according to policies defined by the administrator.

The ideas introduced in this chapter have been originally published in the proceedings of the *7th International Conference on Intelligence in Services and Networks* (IS&N'2000), with further extensions and implementation details published in the proceedings of the *IEEE Global Communications Conference* (Globecom'2000). A brief overview is also included in a paper to appear as an invited contribution in the *Microprocessors and Microsystems* special issue on "Mobile Agent Technology: from first proposals to current evolutions". An extended version of the two conference papers has been submitted to *Computer Networks* journal. Full references are given in Appendix A.

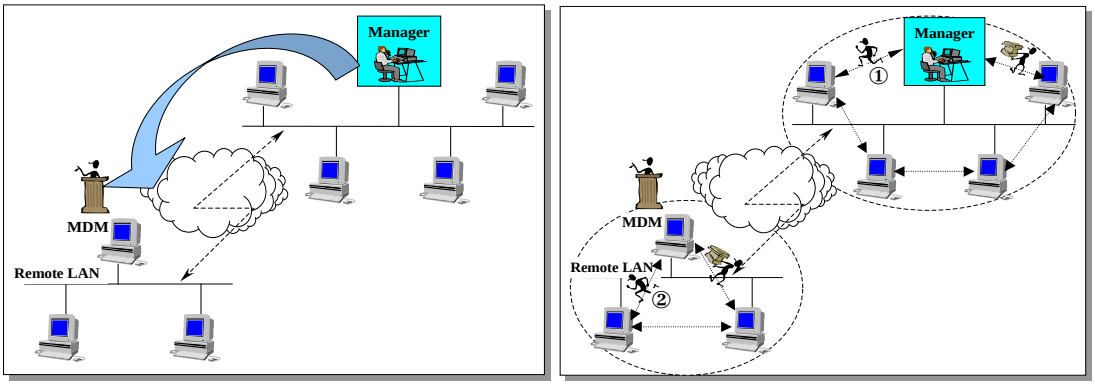
The remainder of the chapter is organised as follows: Section 6.2 describes and explains the rationale behind our chosen hierarchical approach. Section 6.3 comprises the core of the chapter, discussing the implementation details of the introduced architecture. The advantage of the proposed model over non-hierarchical models is verified by a quantitative evaluation of the

bandwidth usage, given in Section 6.4. The evaluation is complemented by experimental results reported in Section 6.5. Finally, the chapter is summarised in Section 6.6.

6.2. HIERARCHICAL, MOBILE AGENT-BASED NETWORK MANAGEMENT

In search of more flexible solutions, this work aspires to push the concept of MA-based hierarchical/distributed management much further. Specifically, we introduce the novel concept of MDM, referring to a management component that operates at a level between the manager and management agent end points. MDM entities are mobile objects that undertake the full responsibility of managing a network domain, when certain criteria (determined by the administrator) are satisfied. Upon being assigned to a domain, the MDM migrates to a host running in that domain (Figure 6.2a) and takes over the management of local NEs from the central manager.

As a result, the traffic related to the management of that domain becomes localised, as the MDM is able to dispatch and receive MAs to collect NSM data from the local hosts (Figure 6.2b), or even execute centralised management operations upon them. The MDM continues to perform its tasks without the manager's intervention, even if the interconnecting link fails. A first-line response can also be given to tackle trivial faults/alarms, with the manager being notified only in case of a complex problem or emergency situation. In performance management applications, only aggregated values and statistics are sent to the manager at regular intervals, thereby diminishing the amount of data transferred through the WAN link. The duration of these intervals is application-dependent and determined by the administrator.



(a) (b)

Figure 6.2. Hierarchical MA-based management

The decision concerning the selection of the host where the MDM will carry out its management tasks from, will be discussed in Section 6.3.6. It is noted that the management domain assigned to an MDM entity may be confined to a single network segment or expand to a larger set of hosts.

The mobility feature of MDMs allows the network management system (NMS) adaptation to network dynamics, optimising the use of network resources. Management functionality can be downloaded at runtime, while this architecture can also dynamically adapt to changing networking conditions. Namely, an MDM entity can be deployed to / removed from a network segment in response to a change in network traffic distribution, or move to the least loaded host to minimise the usage of local resources.

Notably, should the management domain assigned to an MDM comprises a large set of nodes, scalability problems might arise if centralised or flat MA-based management are employed for the data collection process. It is therefore important to further improve NSM scalability by combining our hierarchical framework with GnG/GnS polling schemes. For instance, MDMs can be easily coupled with GnG scheme: after deploying the MDMs to their remote domains, these can launch sufficient number of MAs per polling interval (PI) to reduce the overall response time. Alternatively, coupling the hierarchical model with GnS polling would minimise MA transfers within individual domains, thereby reducing the localised traffic volume.

The adaptive hierarchical management model is particularly suited to dynamically evolving networking environments. Namely, when the administrators perform frequent changes on network configuration, e.g. transfer network devices from one site to another or divide network segments in two or more subdivisions by installing interconnecting devices (bridges, routers, etc). The adaptability feature is also useful in cases that the manager platform is not permanently connected to a specific network location. To illustrate, let us consider an enterprise network comprising two separate segments, managed by a single manager application, possibly running on a portable computer. Should the administrator disconnect that computer from its current location and connect it to another site, the manager application might consider (depending on the chosen management policies) re-organising the NMS, i.e. re-defining the domain boundaries, terminating remote MDMs and deploying new ones to different locations, etc. In the mean time, the management operation will not be affected.

Summarising, the proposed architecture meets the following design requirements:

- *Load balancing*: The total workload should be equally distributed among the various processors of the underlying subsystems. MAs can take full advantage of the increasing processing capability of network devices to achieve management intelligence distribution, however that should not lead to exhaustive consumption of local resources. Therefore, MDMs should be designed as lightweight as possible so as to have minimal footprint on their hosting devices, i.e. they should be equipped with basic management functionality.

- *Fault-tolerance and robustness*: MDM entities should be programmed in such a way so that when detecting a failure on the inter-connecting link between their local subnet and the manager's site or a failure on the manager platform itself, to continue performing their decentralised tasks as normal. As soon as the communication channel is restored, all management data collected in the meanwhile can be returned to the manager.
- *Minimal intrusiveness*: MDMs should be deployed at specific hosts so as to minimise their intrusiveness in terms of the effect of management-related traffic on other applications and the additional processing burden placed upon host processors.
- *Dynamic adaptation*: Topological and traffic characteristics of modern networks are rapidly changing. A hierarchical NSM system should therefore be flexible enough to adapt to those changes. Hence, the location where MDMs run is not fixed, neither is the set of hosts under their control. MDMs can be transparently sent to a domain when the associated cost savings are considerable or removed when their existence is no longer necessary. They can also autonomously decide to move within their domain when the host processor is overloaded and continue their operation on the least loaded node.

6.3. IMPLEMENTATION DETAILS

The hierarchical model introduced herein has been implemented as an extension of the core MAP presented in Chapter 4. The following sections discuss in detail the implementation aspects of the model.

6.3.1. Topology Tree and Topology Map of Active Devices

An important element of the hierarchical framework is the *topology tree* (implemented by the `Manager.TopologyTree` class), a tree structure that comprises a representation of the managed network, made available to the manager application. In particular, the topology tree represents the underlying network topology, namely the individual subnets, the devices physically connected to these subnets and the way the latter are interconnected. It also provides information about the devices hosting active Mobile Agent Servers (MAS). Each of the topology tree nodes corresponds to a specific subnet, with the node representing the manager platform's location being the root of the tree (see Figure 6.3).

The topology tree nodes are implemented by the `Manager.TopologyTreeNode` class and provide the following information:

- the subnet's name;
- the names of hosts and routers physically connected to this subnet;

- a flag indicating the presence of an active MDM on this subnet;
- the number n_l of local active hosts on this subnet;
- the number n_s of active hosts on the subnet's 'subtree' (the term subtree here denotes the set of subnets located in hierarchically lower levels in the topology tree, including the present subnet itself), hence $n_s \geq n_l$;
- pointers to the upper level tree nodes[1];
- pointers to the next level tree nodes;
- a list of graphical components, each corresponding to a specific host, that will be made visible on the topology map upon discovering an active MAS entity on that host.

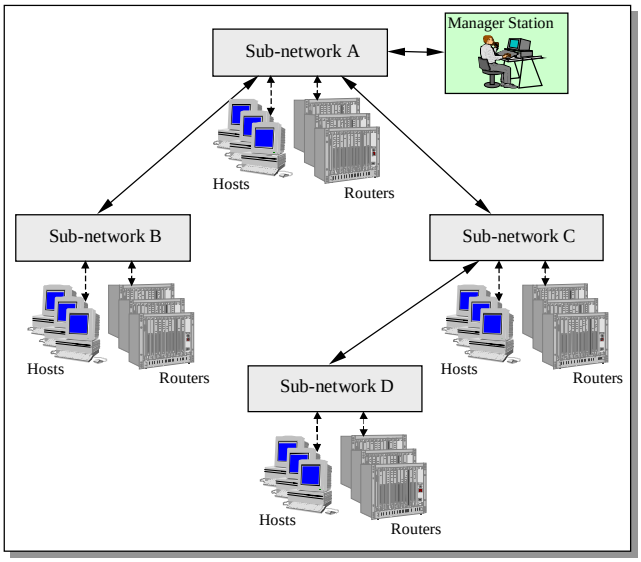


Figure 6.3. The topology tree structure

For instance, the number of active hosts in the subtree of Subnet A (in Figure 6.3) will be:

$$n_{s,subA} = n_{l,subA} + n_{s,subB} + n_{s,subC} = n_{l,subA} + n_{l,subB} + n_{l,subC} + n_{l,subD} \quad (6\text{-}1)$$

All the information related to the managed network described above, is given to the manager application upon its initialisation, through parsing the "*network configuration*" file, described in Section 4.4.1.1. That file does not include activity status information, which is automatically discovered. For each file entry, a new `TopologyTreeNode` is created and inserted into the topology tree. In particular, its 'parent' (upper-level) subnets are located and then the next-level pointer of the parent nodes as well as the upper and next-level pointers of the inserted node are updated.

[1] A `TopologyTreeNode` may have more than one 'parent' nodes. This is certainly inconsistent with the definition of tree structures (as found in graph theory textbooks), hence, the use of the term 'tree' herein is abusive.

As discussed in the following sections, the topology tree plays a crucial role when the manager application needs to make a decision on which subnets require the deployment of an MDM entity.

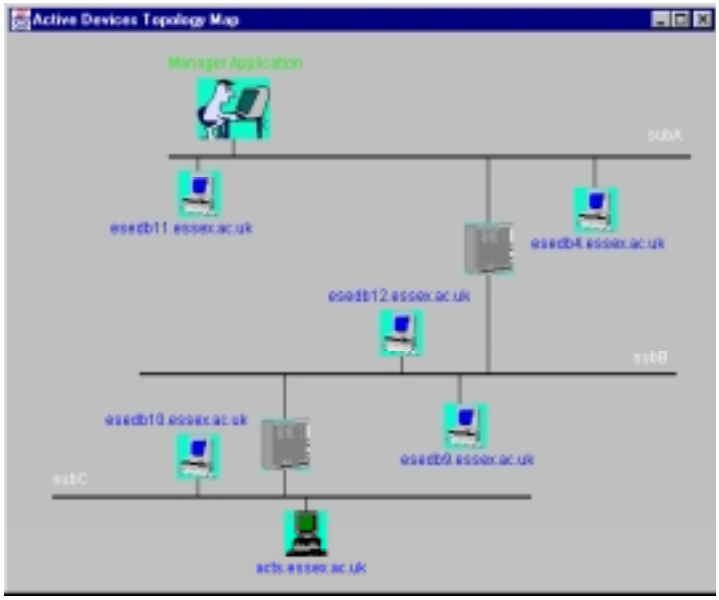


Figure 6.4. Topology map GUI

The topology tree is visually represented by the *topology map* Graphical User Interface (GUI), implemented by the `GUIs.TopologyMap` class; that is a graphical component of the manager application used to view the devices with currently active MAS servers. Managed devices are graphically illustrated with different icons depending on their hardware platform (PCs, workstations, etc). The topology map is updated at real-time when a new MAS is initialised or an existing one terminates. Such events are detected by the manager's Network Discovery thread, as discussed in Section 4.4.1.1. A snapshot of the topology map is shown in Figure 6.4.

6.3.2. MDMs implementation

The functionality of MDM entities is defined in the `MCode.MDM` class. As already mentioned, MDMs are essentially *mobile objects*. Namely, the MDM class extends the `MCode.MA` 'superclass', which defines the basic properties required to implement mobility characteristics (see Section 4.4.1.2). Yet, MDMs also act as remote *managers*. However, due to a limitation of Java programming language which does not support multiple inheritance, i.e. it does not allow a Java class to subclass two separate 'parent' classes, `MDM` class cannot extend the `Manager.Manager` class. For that reason, the `Manager.ManagerInterface` has

been implemented. As its name suggests, `ManagerInterface` is a Java *interface*[2] including the definitions of a number of *abstract*[3] methods. Both `Manager` and `MDM` classes implement the `ManagerInterface` in addition to the `java.util.Observer` interface, which allows them to receive events notifications from the MAG tool[4], when an existing Polling Thread (PT) configuration is modified by the administrator.

MDMs deployment involves the transfer of the `MCode.MDM` class and also a number of classes that assist the MDM in performing its decentralised NSM operations. These include:

- the `Manager.PT` class, which controls the execution of specialised monitoring tasks already defined by the administrator and the `Manager.PTC` class, whose instances comprise descriptions (configurations) of the PTs;
- the `MCode.DomainMonitoring` and `MCode.RI` class which, as described in Section 6.3.6, are responsible for checking the utilisation of the hosts located within the MDM's domain and ensuring that the MDM executes at the least loaded host;
- the `RMI.MdmRmiServer` class, implementing the MDM's RMI server which enables the communication between the manager and the MDM.

All these *six* classes are packaged in a *jar*[5] file whose size is 23.4 KB. The jar file is sent along with an MDM object carrying information about its assigned domain (e.g. the names of the managed devices located therein) and descriptions of the management operations it will perform. Upon arriving to a remote host, the MDM first starts its RMI server and then *registers* itself to the manager application (the `registerMDMToMAG()` method of the `ManagerRmiServer` class is invoked). The registration process is necessary in order to let the manager know the MDM's current location and also to subscribe to events generated by the MAG tool whenever a PT configuration is modified. MDM's registration process comprises two phases: first, the MDM passes its current location IP address to the

---

[2] *Interfaces* are defined similarly to classes, but include only declarations of methods. The designer of an interface declares the methods that should be supported by classes that *implement* the interface and what the functionality of these methods should be. Interfaces represent a powerful mechanism that helps to achieve many of the advantages of multiple inheritance, without its problems [ARN96].

[3] Methods declared as *abstract* can be defined either in abstract classes or interfaces. Their implementation is not supplied; the classes extending the abstract class or implementing the interface are enforced to provide specific implementations of the abstract methods [ARN96].

[4] The `Manager.MAG` class extends the `java.util.Observable` class. An *observable* object can have one or more *observers* (objects implementing the `java.util.Observer` interface). When an observable instance changes, the `Observable`'s `notifyObservers()` method is called, causing all its observers to be notified of the change by invoking their `update()` method [ARN96].

[5] The *Java Archive* (JAR) file format enables bundling multiple files into a single archive file. Typically, a JAR file contains the class files and auxiliary resources associated with applets and applications. The JAR file format provides many benefits, namely security, decreased download time, compression, portability, etc [JAR].

`ManagerRmiServer` object. The latter will then use that address to obtain (again through RMI) a *reference* of the MDM object, which in turn is added to the MAG's list of observers[6]. When an event is generated (a PT configuration is updated), the MAG will automatically propagate the event to all the observers (i.e. the manager application and the remote MDMs), which in turn will adjust the operation of the updated PT. The event propagation is completely transparent to the user, who does not have to be aware of the remote location where MDMs execute.

Upon successful registration, the MDM instantiates the `DomainMonitoring` object and starts the execution of the PT threads. Likewise, before migrating to another host, the MDM first enforces executing PTs to return their collected data to the manager, suspends the execution of the PTs and unregisters itself from receiving MAG's events (through invoking the `removeMDMFromMAG()` method of the `ManagerRmiServer` class).

6.3.3. MDMs Deployment Policies

A key characteristic of our hierarchical framework is its dynamic adaptation to changes on the managed network. The structure of the proposed model is not rigidly configured, as MDMs may be dynamically deployed to specific network domains, given that certain requirements are met. Specifically, the administrator may explicitly set (through the GUI shown in Figure 6.5) the policies that define the hierarchical NMS operation, i.e. specify the criteria that should be satisfied for deploying an MDM to a network segment. The settings customising the operation of the hierarchical management model are stored in an instance of the `Manager.HierarchicalSettings` class.

In general, the deployment of MDMs may conform to either of the two following policies:

- **Policy 1**: the population of remotely active managed devices.
- **Policy 2**: the overall cost involved with the management of a remote set of devices.

When applying *Policy 1*, the administrator specifies the number of remote managed NEs that will justify the deployment of an MDM to a particular network segment. This number may either denote n_l or n_s (see Section 6.3.1). If, for instance, the specified number N denotes the

[6] Dividing the registration process in two phases was necessary. Namely, it would not be possible to simply pass an instance of an MDM object and append it to the observers' list. That would presuppose that the PT objects controlled by the MDM would be serialised (when a method passes an object's instance through RMI, the object is actually serialised and so are the objects referenced by the serialised object). However, that would not be feasible as Java threads cannot be serialised. To get around this problem, the `registerMDMToMAG()` method obtains a reference of the MDM object. It is the MDM's reference that will be added to the MAG's observers list. Thus, when a MAG event is generated, the MDM's `update()` method will be basically executed at the remote host where the MDM is running and the PTs vector is maintained. Through this programming trick, i.e. by registering a *reference* and not an *instance* of the remote MDM, we obviate the need for serialising the MDM object.

population of the examined subnet's local devices n_l, an MDM will be deployed to every network segment S with $n_{l,S} \geq N$; the boundaries of the domain assigned to the MDM will then be limited to that segment. If, on the other hand, N denotes the active hosts on the subnet's 'subtree' n_s, the domain assigned to the MDM will include all the active hosts located within the examined subnet's subtree, excluding the hosts already assigned to another MDM. The MDM will initially migrate to the least loaded host included into its assigned domain (this issue is discussed in detail in Section 6.3.6).

When the population of NEs directly managed by an MDM exceeds a certain limit, that domain will be divided to two independent domains, with another MDM undertaking the management of the second domain. In particular, the remote MDM will be instructed to create a clone of itself (through invoking the `clone()` method of the `java.lang.Object` class). The original MDM will pass to its clone information regarding its assigned domain, before dispatching it to a new location. The cloning approach is preferable than deploying a new MDM from a central location, as it reduces the deployment time in addition to saving the network overhead associated with the deployment process.

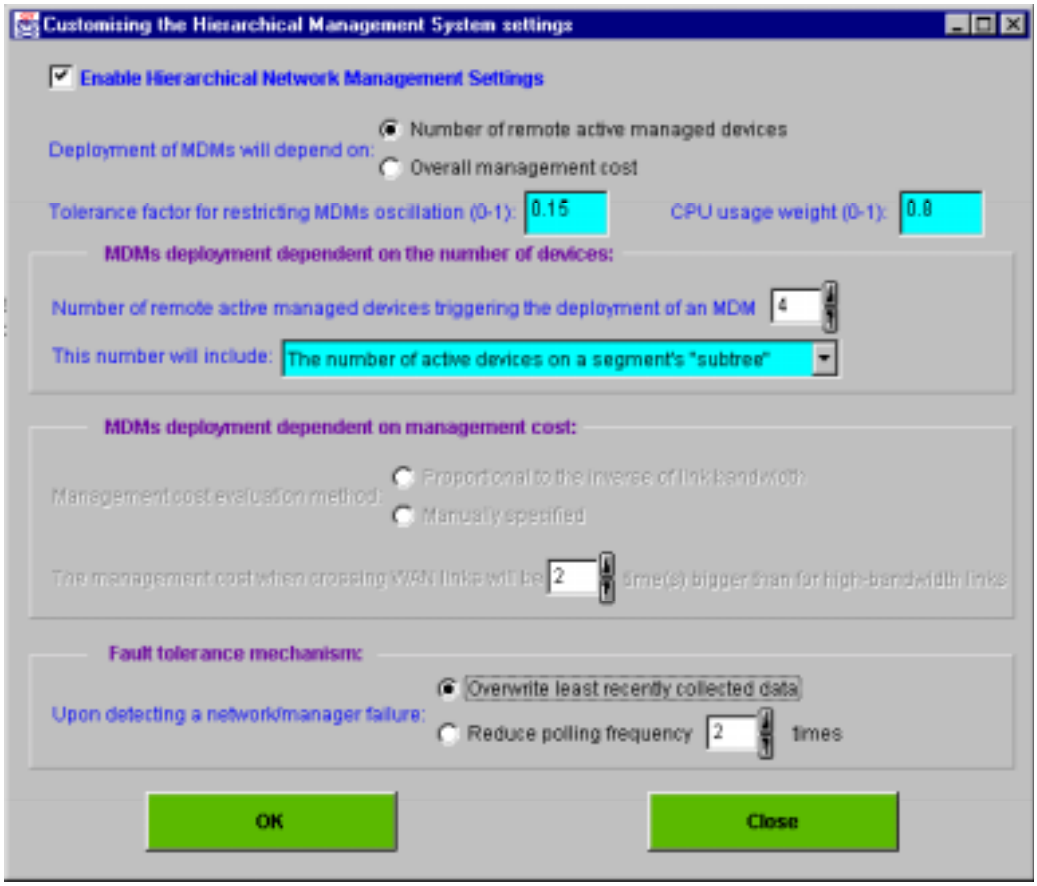


Figure 6.5. GUI for customising the hierarchical NMS policies

Accordingly, when the number of NEs managed by an MDM is reduced so that the requirements that originally triggered the MDM's deployment are no longer met, the manager

application might consider of *merging* two or more management domains. Specifically, it will request an MDM to assign its managed devices to a neighbour MDM and subsequently terminate its execution.

When applying *Policy 2*, the management cost may either be: (a) proportional to the inverse of link bandwidth (a link with low bandwidth implies higher cost than a high-speed link), or (b) manually specified. It should be noted that Policy 2 is not implemented in the current prototype.

6.3.4. MDMs Deployment Implementation

Upon discovering an active MAS module, the topology tree is scanned to locate the corresponding host and the subnet where the host resides; the host's icon is also instantly made visible on the topology map. Following that, the number n_l of active hosts on that subnet is increased by one and subsequently, through following the pointer to the upper-level nodes, all the topology tree nodes up to the root are traversed and their number n_s of subtree nodes is also updated (increased by one). A similar procedure is followed when a MAS server is being shut down.

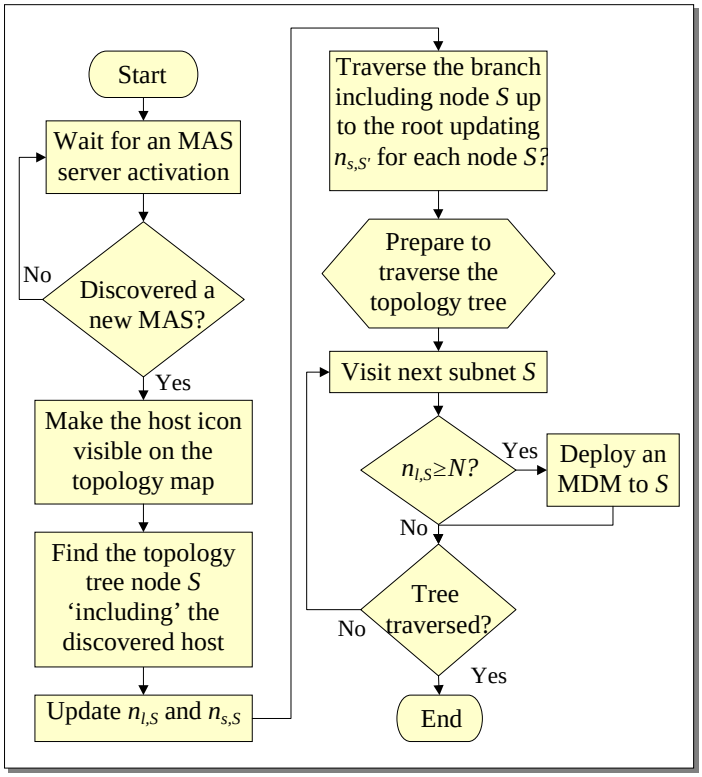


Figure 6.6. MDMs deployment algorithm (an MDM is deployed to each subnet including at least *N* active MAS servers).

The discovery or termination of a MAS server triggers an event at the manager host. The topology tree is then scanned with the subnets that meet certain requirements (determined by the `HierarchicalSettings` object) added to a list. In case that 'Policy 1' is employed, referring to the policies listed in the previous section, that list will include the subnets with n_l or n_s (depending on whether the MDMs deployment is a function of the active devices running locally or in the whole subtree) greater or equal to the specified constant N. If 'Policy 2' was employed, the cost corresponding to the management of each subnet would be evaluated and the list of subnets created accordingly. Ultimately, an MDM will be deployed to each of the subnets included in the list. The MDM deployment algorithm is illustrated in the block diagram of Figure 6.6, for the simple case where an MDM is deployed to each subnet including at least N active MAS servers.

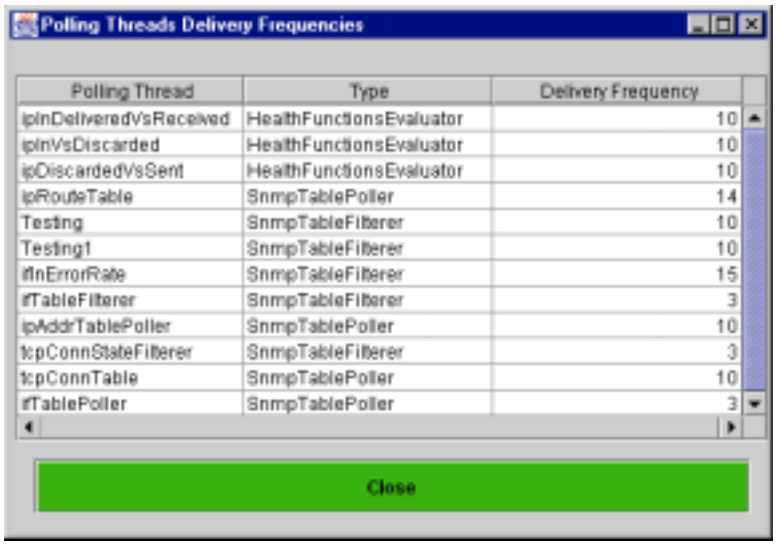


Figure 6.7. Adjusting PTs delivery frequency

Certainly, the set of management tasks already performed by the manager on these subnets will need to be conveyed to the MDM deployed therein. This is achieved through sending existing Polling Thread Configurations (PTC) along with the MDM. PTCs functionality has been described in Section 5.2.1.1. The reason for uploading the PTCs rather than the actual PTs is that the latter cannot be transparently transferred along with the MDM retaining their execution state, as Java does not support threads serialisation/deserialisation[7]. On the other hand, PTC objects are serialisable. Upon its arrival at the remote subnet, the MDM instantiates the PTs using their corresponding configurations, encapsulated into the MDM's state. PTs will thereafter start performing their tasks without any further disruption of the management process. Individual PTs are responsible for periodically delivering their collected data to the manager. The delivery frequency of monitoring tasks can be remotely adjusted by the

[7] The `java.lang.Thread` class does not implement the `java.io.Serializable` interface.

administrator, through the GUI shown in Figure 6.7; changes in delivery frequencies are communicated to the PTs through invoking the `update()` method of the `MdmRmiServer` class. It is noted that PTs are not aware of whether they operate under the control of the manager application or an MDM, as both `Manager` and `MDM` classes implement the `ManagerInterface`. That enables a flexible design and maximises code reuse.

6.3.5. MA Code Distribution Scheme

A key feature of MA-based NSM frameworks is their ability to dynamically customise management services. Our hierarchical model should therefore ensure that new NSM services (implemented by specialised MAs) can be introduced/updated at runtime, even post MDMs deployment. However, that should not affect the framework's performance, which primarily depends on the size of travelling MA entities. In a typical Java-based MAP, both the MA *code* and *state* are required at the destination to instantiate the received MA objects. Nicklisch et al. [NIC98] identified three alternative agent code transfer schemes: (a) "*push*", (b) "*pull*" and (c) "*migrate*" (see Section 3.6.1.3).

The majority of MAPs proposed for management applications [KU97, ZAP97, COR98a, BEL99, SAH98, SUS98], with the exception of [SOA99] and [PUL00a], involve transfer of both the MA's code and state on each migration, namely they employ the "migrate" approach. In Section 4.4.1.4.1, we have described a code distribution scheme implementing the "push" scheme; i.e. bytecode is distributed at the MA's construction time with only the state information transferred thereafter, resulting in a much lower demand on network resources. The "migrate" code distribution scheme offers a better starting point in terms of the associated network overhead, since the bootstrapping procedure described above is not required. However, it is outperformed by the "push" approach, after a small number of PIs elapses.

The introduction of the hierarchical management model reduces the code distribution cost even further by adopting a "*tree multicasting*" approach. In particular, MA bytecode distribution takes place in two successive phases. In the first phase the MAs bytecode is no longer multicasted to all managed devices as in the flat model (see Section 4.4.1.4.1.), but instead distributed to the MAS entities local to the manager segment and also to all active MDMs. The latter will then forward the received bytecode to their supervised NEs (see Figure 6.8). Should a remote domain including *N* hosts connects to the manager site through a low-bandwidth link, the tree multicasting approach will considerably decrease management cost as bytecode is transferred only once (instead of *N* times) through the interconnecting link. The execution of the monitoring task implemented by the uploaded MA class is started immediately by a dedicated PT, instantiated by the MDM for this purpose.

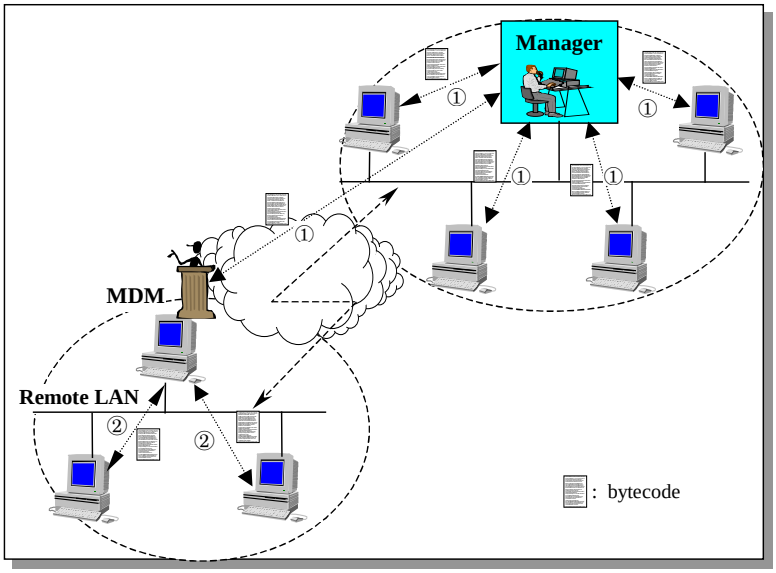


Figure 6.8. MA bytecode through "tree multicasting"

6.3.6. Processing Load Balancing

Although MDMs have been designed to be as lightweight as possible, they cannot avoid consuming memory and processing resources on the NE where they execute. The framework should therefore be sufficiently flexible to allow MDMs to autonomously move to another host, when their current hosting device is overloaded, to provide more balanced distribution of the overall processing load.

This is accomplished through the regular inspection of the domain's NEs, in terms of their memory and CPU utilisation: an MA object, termed *Resource Inspector* (RI), implemented by the `MCode.RI` class, is periodically dispatched (by the `DomainMonitoring` thread) and visits all the local devices obtaining utilisation figures before delivering the results to the MDM. Host load figures represent their average load over relatively long time windows to avoid sensitivity to temporal utilisation peaks. If the hosting system is seriously overloaded, compared to its neighbouring devices, the MDM will transparently move to the least loaded node. In the example depicted in Figure 6.9, a RI object sequentially visits all the managed devices in the MDM's local domain. At each host, the RI obtains the average CPU & memory load values during the last interval, keeping track of the least loaded device (in this example that will be Host *D*). Finally, the RI reports its results to the MDM, which in turn transparently migrates from Host *A* to Host *D* after informing the manager application about its decision. The correct and reliable operation of the MDMs transparent location change has been verified by overloading an MDM's hosting device (through launching several memory-consuming applications and running heavy background jobs), thereby enforcing the MDM to migrate to a least loaded host.

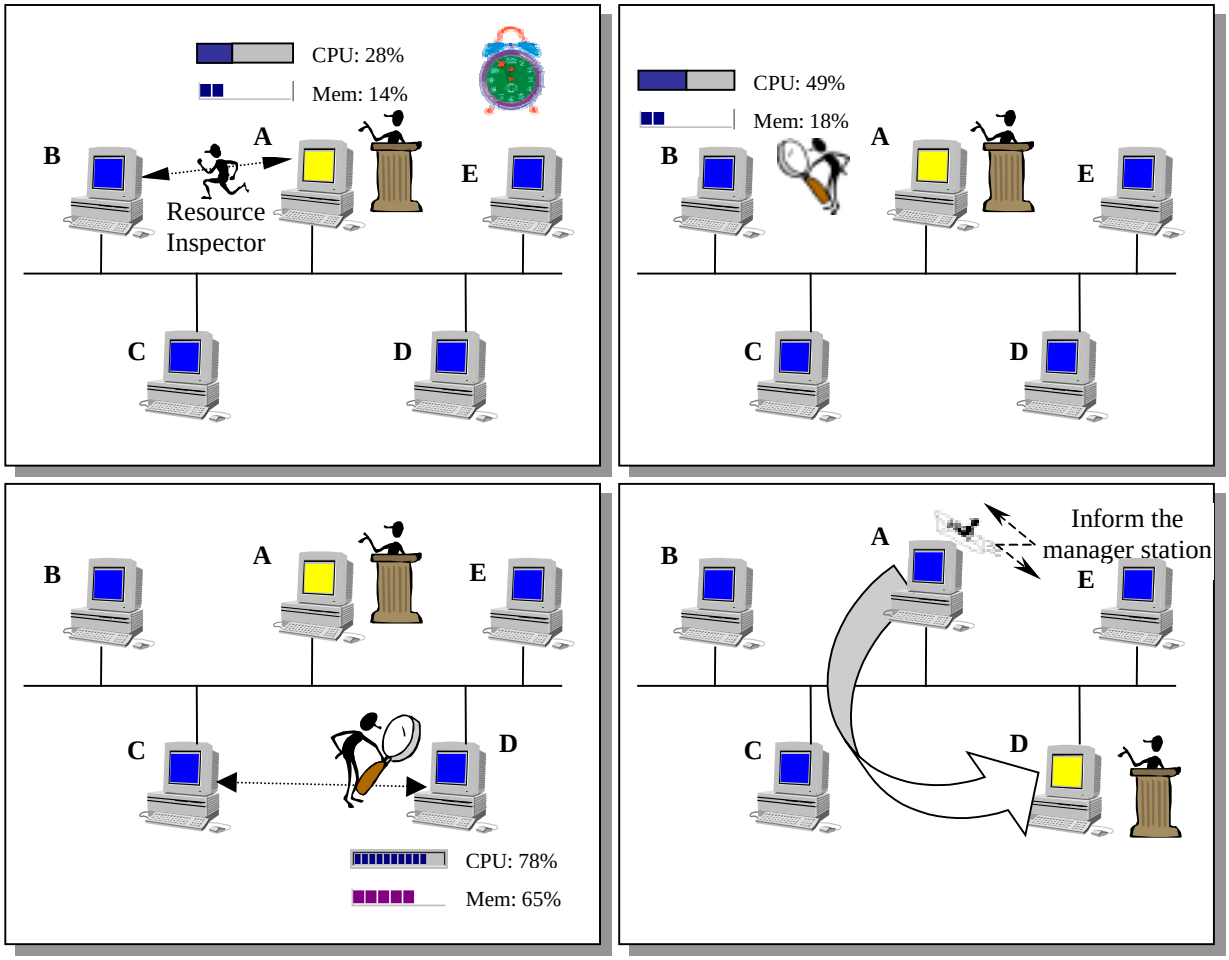


Figure 6.9. Obtaining host load profiles by a Resources Inspector MA object

MDMs are prevented from continuously *oscillating* between different hosts through adjusting the value of the *tolerance factor*, $0 < t_f < 1$. Assuming that an MDM executes on a host *x*, with average utilisation U_x, the MDM will consider migrating to another host *y* only if its utilisation is $U_y < (1 - t_f)\, U_x$. If for instance, $t_f = 0.2$, the MDM will not migrate to *y* unless its utilisation level is at least 20% lower than that of *x*.

A host's average utilisation U_x is a linear function of the CPU and memory usage, i.e. $U_x = a \times CPU_x + b \times M_x$. In particular, the administrator determines through the GUI shown in Figure 6.5 the weight *a* that CPU usage CPU_x will have on the evaluation of the aggregate host utilisation, where $0 < a < 1$. The weight of memory usage M_x will then be: $b = 1 - a$.

The MDM notifies the manager application about its new execution location before the actual migration occurs. Should the manager attempt to contact the MDM while the latter is still moving, an exception will be thrown and a new attempt to contact the MDM will take place after a specified interval; in the meanwhile, the MDM arrives at its new hosting device and is capable of receiving manager's messages. The location where the MDM executes is indicated on the manager's topology map through a label displayed next to the MDM's hosting

device icon. Location changes trigger real-time map updates. For instance, Figure 6.10 illustrates two snapshots of the topology map before and after the migration of an MDM from esedb9.essex.ac.uk to esedb12.essex.ac.uk host.

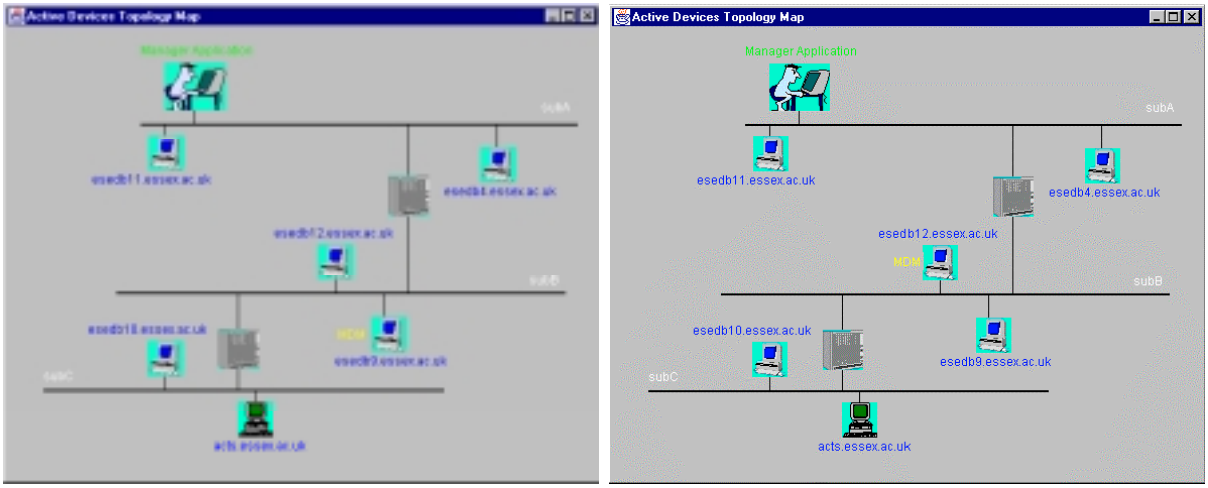


Figure 6.10. Illustration of an MDM location change on the manager's topology map

6.3.7. Resources Monitoring Tool Implementation

In order to obtain devices' load reports we have built a Resources Monitoring Tool (RMT), developed in C, able to accurately measure the CPU and memory usage. On Windows platforms, low-level functions of the Win32 API[8] [Win32] have been utilised, whereas standard Unix commands are executed under Solaris.

The integration of this tool with the MAS application, developed in Java, is achieved through the Java Native Interface (JNI), discussed in Section 2.7.1.1. The JNI allows Java code running within a JVM to inter-operate with applications and libraries written in other languages, such as C or C++. JNI is used to write native methods to handle situations where an application cannot be written entirely in Java. The steps required to achieve the integration of Java and native programs are described in [JNI]. The Java front-end (methods) accessed by incoming RIs provides them a uniform interface onto the local resources, whilst hiding the underlying architecture, i.e. the native methods implementation.

Snapshots of hosts' load profiles are taken in regular intervals. The duration of these intervals, termed *observation periods* O_p, should be carefully specified: O_p should be long enough to avoid sensitivity on sporadic load peaks and, at the same time, short enough so as not to omit potentially prolonged increments of processing load.

[8] The Microsoft Win32 API provides building blocks used by applications written for Microsoft Windows NT, Windows 2000, and Windows 95/98. Among others, an application using Win32 functions is able to manage system objects such as memory, disks, files, and processes.

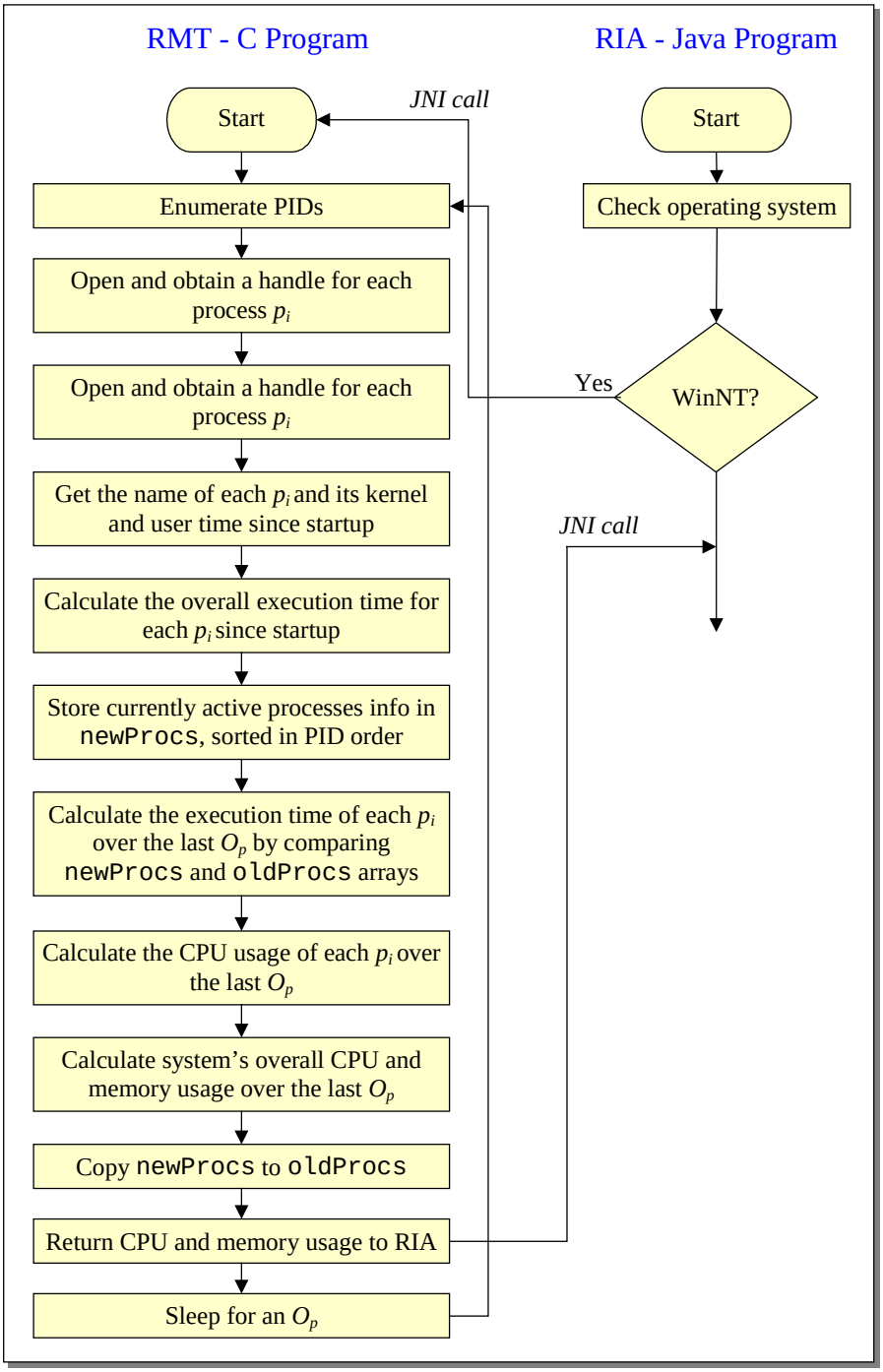


Figure 6.11. RMT and RIA operation

The operation of the RMT tool is only briefly described in the following (a detailed description would be beyond the scope of this thesis): RMT's execution is started and controlled by the Resource Inspection Application (RIA) of the MAS server (see Section 4.4.1.3.7), defined in the `MAS.RIA` class. A RIA thread is created and started upon the MAS

initialisation. First, RIA checks the identity of the underlying operating system[9]. Should RIA runs on a Windows platform, the RMT tool is initialised through a native call and starts monitoring the local resources usage (in time intervals defined by the O_p).

In particular, RMT periodically executes the `EnumProcesses()` function [Win32], which returns an enumeration of active system process identifiers (PID). Then, for each retrieved PID, the `OpenProcess()` is invoked, opening and returning a handle to the process object p_i. Handles are used as process references to obtain the process name (`GetModuleBaseName()` function) and also the process creation and execution time, through calling `GetProcessTimes()` function. In particular, when invoking the latter function at instant t_n, it returns the time period that process p_i has spent executing in kernel K_{p_i,t_n} and user mode U_{p_i,t_n} since system's startup.

The overall process execution time T_{p_i,t_n} is then calculated ($T_{p_i,t_n} = K_{p_i,t_n} + U_{p_i,t_n}$). Individual process memory usages M_{p_i} are also found through calling the `GetProcessMemoryInfo()` function. The information referring to currently active processes is stored in `newProcs` array[10], sorted in PID order. The execution time of each individual process over the last O_p is then calculated, by using the information stored in `newProcs` and `oldProcs`[11] arrays: $T_{p_i,O_p} = T_{p_i,t_n} - T_{p_i,t_{n-1}}$. The CPU usage of p_i is then easily found: $CPU_{p_i,O_p} = T_{p_i,O_p} / O_p$. The aggregate system and memory usage can be evaluated by summing up the individual process CPU and memory usages: $CPU_{O_p} = \sum_i CPU_{p_i,O_p}$, $M_{O_p} = \sum_i M_{p_i}$.

Finally, the contents of `newProcs` array are copied to `oldProcs` and RMT delivers a system resources report to RIA through a JNI call. RMT then 'sleeps' for a time period equal to O_p (the `Sleep()` function is called) before resuming execution and repeating the procedure described above. RMT and RIA operation is illustrated in the block diagram of Figure 6.11. An overview of the process enumeration procedure is given in [Win32Enum].

---

[9] This is achieved through simply examining the value of the local system's file separator (the `getProperty("file.separator")` method of the `java.lang.System` class is invoked). The file separator is '\' for Windows systems and '/' for Unix systems.

[10] `newProcs` array contains a set of records, each of which corresponds to a specific process p_i and includes its PID, its overall execution time since system's startup T_{p_i,t_n} and its memory usage M_{p_i}. The information stored in `newProcs` array refers to instant t_n.

[11] `oldProcs` array stores similar information to `newProcs`, but refers to instant t_{n-1}, where t_n-t_{n-1}=O_p.

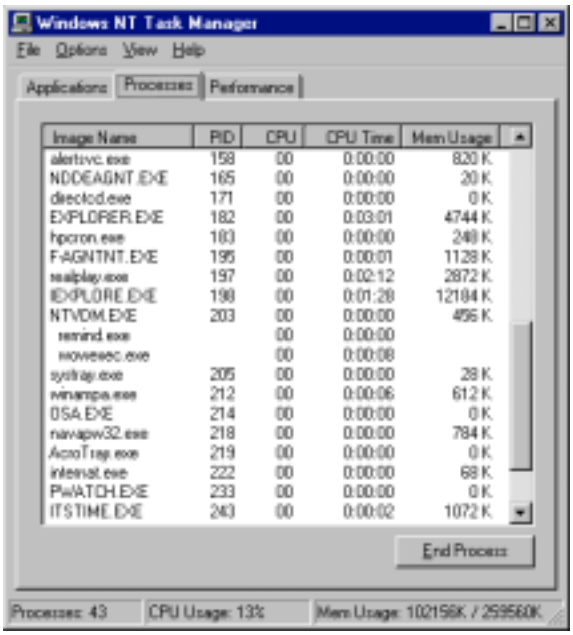


Figure 6.12. The Windows NT Task Manager

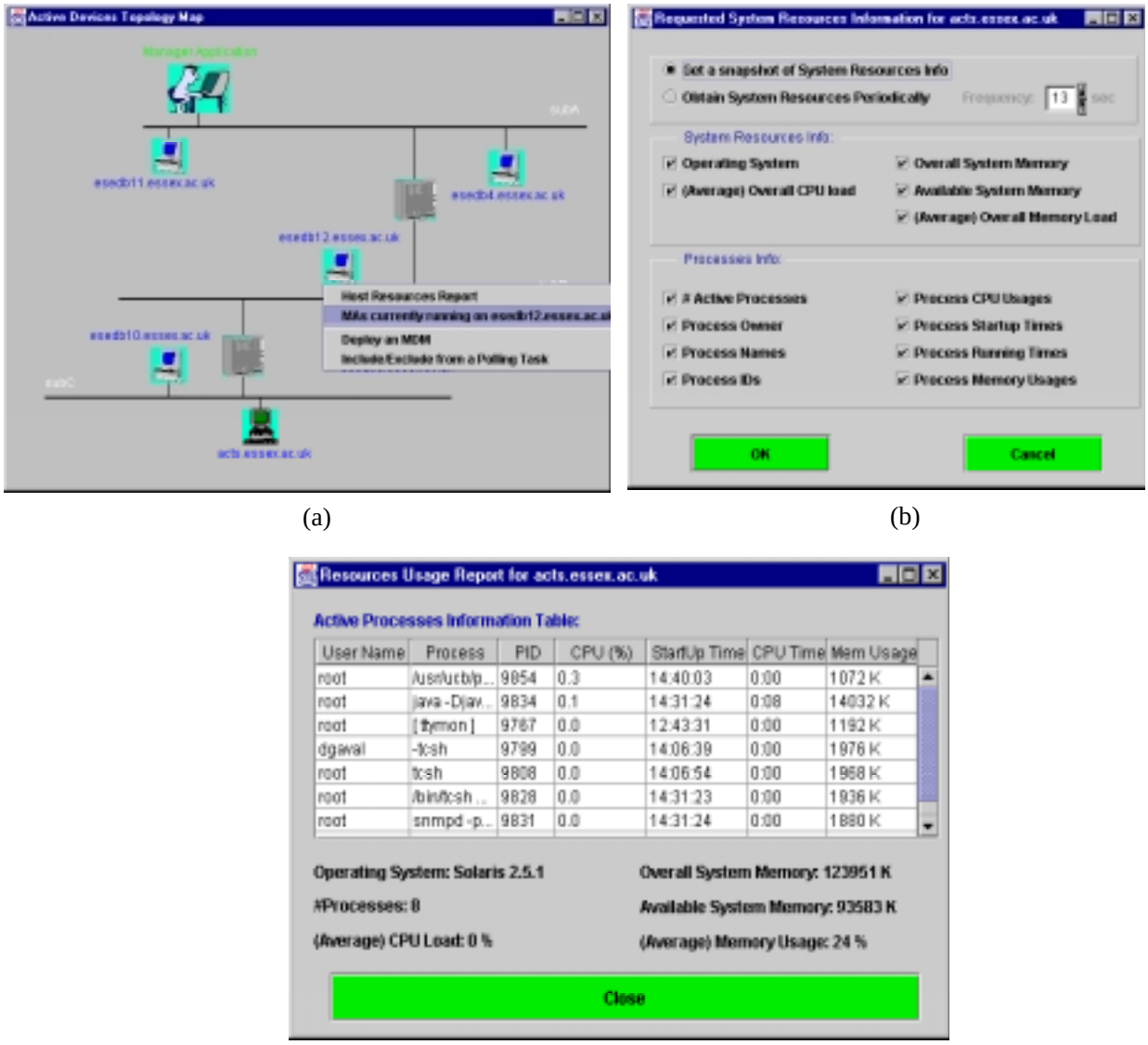
The C file implementing the procedure described above, along with the accompanying header files are compiled creating a library (DLL) file. That library, along with `psapi.dll` file which includes the implementations of the Win32 API functions comprise the RMT tool, representing an overall size of 100Kbytes. Among the existing Windows operating systems family, the current implementation of the RMT supports only Windows NT platforms; however, as claimed in [Win32Enum] only minor modifications are required to extend its support to Windows 95/98 and 2000 systems. In general, the RMT tool provides information very similar to that provided by the Windows NT Task Manager application (see Figure 6.12).

Regarding Solaris operating systems, the standard UNIX command *ps -au* is periodically executed[12]. That command prints the list of active system processes along with their PID, CPU and memory usage, startup time, etc. The command's output is captured and subsequently parsed in order to obtain the host's CPU and memory utilisation following a similar approach to the one described for Windows platforms.

In addition to assisting MDMs to locate the least loaded host within their management domain, RMT can also be utilised to provide user reports, describing the load profile of a remote host. This can be achieved through simply right-clicking on the host's icon on the topology map (Figure 6.13a). The administrator then selects the kind of system resources information he/she is interested in (Figure 6.13b). The user's request is subsequently sent to the

[12] Operating system-specific commands can be executed by a Java application through calling the `exec(<command>)` method of `java.lang.Runtime` class. `exec` method returns the process (`java.lang.Process`) handling the command's execution. The command's output can also be captured (through the `getInputStream()` method of `java.lang.Process`) and subsequently parsed.

remote MAS server (as a `Manager.ReportDescription` object). In particular, the `MasRmiServer`'s `getResourcesReport()` method is invoked (see Figure 4.14), with the returned information displayed in the GUI shown in Figure 6.13c.



(a) (b)

(c)

Figure 6.13. Obtaining on-line resource usage reports from remote devices

6.3.8. Manager-MDMs Communication

One of our framework's key advantages is that it considerably reduces the amount of information exchanged between the manager platform and the managed devices. This is due to the introduction of the intermediate management level, realised by dynamically deployed MDMs. However, that does not obviate the necessity for bi-directional communication between MDMs and the manager host. Java RMI [RMI] has been chosen for implementing the

communication bus between the distributed MDMs and the manager host, due to its inherent simplicity and the rapid prototype development that it offers.

In particular, the `RMI.ManagerRmiServer` class is used to pass information to the manager, while the `RMI.MdmRmiServer` class has been implemented to enable communication in the reverse direction (see Figure 6.14 for an overview of its methods). In the MDM-to-Manager direction, we have the following information flow:

- delivery of data reports generated through filtering raw data collected from local devices (these data have been obtained by MAs launched by the MDM's PTs);
- registration of the MDM to MAG's events;
- notification of the manager prior to an MDM's migration to another host.

```
public boolean terminate () {}  // Requests the MDM to terminate its execution
public boolean move (String host) {}   // Requests the MDM to migrate to another location
public boolean clone() {} /* Requests the MDM to create a clone and share the management
                              responsibility of its assigned domain with it */
public boolean deliverData () {}   // Requests the delivery of collected data from all running PTs
public boolean addPT (PTC configuration) {} // Uploads a new PT configuration
public boolean removePT (String PTname) {} // Requests the disposal of an existing PT
public boolean update (String PTname, PTC newConfiguration) {}  /* Updates an existing PT
                                                                    configuration */
public boolean addManagedDevice (String host) {}  /* Add a new host to the MDM's list of
                                                      managed devices */
public boolean removeManagedDevice (String host) {}   /* Removes a host from the MDM's
                                                          list of managed devices */
public Vector getManagedDevices () {}  // Returns the MDM's list of managed devices
public void returnDataFolder (String PTname, String MaSeqNum, Vector data) {}  /* Invoked by an
                              MA with sequence number MaSeqNum, which returns its collected data
                              to the PT that originally launched it */
```

Figure 6.14. The methods of the RMI.MdmRmiServer class

On the opposite direction, the manager may:

- request the MDM to terminate its execution, move to another domain or create a clone of itself to share the management responsibility of its assigned domain;
- request the MDM to enforce its controlled PTs to deliver all their collected data;
- upload at runtime additional management services (PTCs) or request the termination of an existing one;
- update an existing PT configuration;
- add/remove a NE from the MDM's list of managed devices;
- obtain the list of the MDM's supervised devices.

6.3.9. Fault Tolerance

A key motivation for the development of the introduced hierarchical model has been to minimise dependency on network resources and exploit MAs ability of acting autonomously, without the manager's intervention. Hence, MDM entities have been designed so as to tolerate failures on the interconnecting links between their local subnets and the manager's site or on the manager platform itself. Such failures are typically detected when MDMs attempt to return aggregated results back to the manager. Upon detecting a failure (an exception is thrown when the data transfer cannot be completed), they continue to perform their decentralised tasks as normal while periodically checking for the status of the link and/or the manager. As soon as the communication is restored, all management data collected in the meanwhile are returned to the manager.

Certainly, in case that a large number of monitoring tasks are controlled by the MDM and should the interruption of the normal communication flow between the MDM and the manager be prolonged, the MDM's size will significantly grow. That may in turn have a serious impact on the MDM's hosting device resources. Hence, in order to maintain control over the growth of the MDMs state, the administrator may choose (through the GUI shown in Figure 6.5) between the following models: As soon as an MDM detects the failure it will either (i) overwrite the least recently collected management data by keeping only the latest acquired values, or (ii) reduce the polling frequency of the individual monitoring tasks so that the management data accumulation rate will decrease. The reliable operation of the fault tolerance mechanism has been verified by creating 'artificial' failures on the manager host, i.e. by shutting down the manager application and restarting it after an MDM detected the failure.

Future extensions will also cater for faults on MDMs hosting devices: The manager will periodically check the status of the devices where MDMs currently execute. Should a hosting device fails, the manager will automatically deploy another MDM to a 'healthy' host to take over the management of the remote subnet.

6.4. QUANTITATIVE EVALUATION

This section presents mathematical formulations, which quantify the network overhead associated with the employment of the proposed hierarchical model. This quantitative evaluation builds upon previous evaluations, presented in Sections 4.5 and 5.3, using the same definitions, symbols and formalisms.

Although mobility can often be beneficial for NSM, overheads induced by MAs and MDMs in particular, e.g. due to their deployment and management should be accounted for very

carefully. Slightly different configurations for a set of MDMs may result in dramatically variant network loads [LIO98]. Hence, it is important to define concrete cost functions estimating the corresponding overheads.

In this context, let the *cost coefficients* k_{S_i,S_j} denote the cost of sending a byte of information between arbitrarily indexed subnets S_i and S_j, where S_0 is the manager host location. For multi-hop connections, the cost coefficients will be equal to the summation of the individual links coefficients. In the following investigation, we make the simplifying assumption that an MDM may manage only the hosts included in a *single* subnet and not a wider set of devices.

A simple function characterising the bandwidth consumption for our hierarchical architecture, is the following:

$$C_{hier} = C_{distr} + C_{depl} + C_{pol} + C_{deliv} \tag{6-2}$$

where the four terms represent the cost for distributing to the MAS servers the bytecode of the MA that will undertake the monitoring task, the MDMs deployment cost, the bandwidth used for the actual monitoring operation (polling) and the cost for delivering to the manager host the collected data, respectively.

Concerning bytecode distribution, the lightweight "tree multicasting" scheme described in Section 6.3.5 is adopted. The code distribution cost is therefore given by:

$$C_{distr} = \left(k_{S_0,S_0} * N_0 + \sum_{i=0}^{M} \left[k_{S_0,S^i} + k_{S^i,S^i} * \left(N_{S^i} - 1 \right) \right] \right) * C \tag{6-3}$$

where M is the total number of active MDMs, C the compressed bytecode size, S^i represents the subnet including host i, and N_{S^i} the number of hosts included in subnet S^i.

Likewise, C_{depl} equals the cost of broadcasting M MDM objects to their corresponding remote domains:

$$C_{depl} = \sum_{i=0}^{M} k_{S_0,S^i} * (C_{MDM} + ST_0 + n * \overline{S}_{PTC}) \tag{6-4}$$

where C_{MDM} is the size of the *jar* file uploaded at every MDM deployment[13] (= 23.4 Kb), ST_i represents the compressed state size of an MA when migrating from the ith host and $\overline{S}_{PTC}$

[13] See Section 6.3.2.

the average size of each of the *n* PT configurations attached to the MDM (each serialised PTC object amounts approximately 250 bytes).

C_{pol} is defined as the summation of the cost induced for polling the NEs directly managed by the manager host and the cost associated with polling the NEs that operate under the MDMs control, multiplied with the number of PIs *p*:

$$C_{pol} = \left(\sum_{i=0}^{m} k_{S^i,S^{i+1}} * ST_i + \sum_{i=0}^{M} \sum_{j=0}^{N_i} k_{S^i,S^i} * ST_j \right) * p \qquad (6\text{-}5)$$

Clearly, the first term of the summation will dominate on the overall polling cost if the *m* devices managed by the central manager platform are spread among several subnets. Specifically, cost coefficients $k_{S^i,S^{i+1}}$ are typically larger when an MA migrates from subnet S^i to another subnet S^{i+1} ($S^i \neq S^{i+1}$) rather than when it moves within the same subnet ($S^i = S^{i+1}$). As discussed in Section 4.5.2, MAs state size ST_i does not remain constant, but increases for each visited node, with the increment rate depending on the selectivity σ, as shown by Eqn. (4.7). The last term appearing in Eqn. (6-2) represents the cost associated with the delivery of the gathered data from the MDMs to the manager host:

$$C_{deliv} = \frac{D * p}{F_{del}} * \sum_{i=0}^{M} k_{S^i,S_0} \qquad (6\text{-}6)$$

where F_{del} indicates (in number of PIs) the delivery frequency, namely how often MDMs package the computed statistics of size *D* and deliver them to the manager.

It should be emphasised that MDM functionality is not necessarily limited to simply gathering and delivering data to an upper-level manager. Although this thesis concentrates on data-intensive network monitoring applications, a broad spectrum of management applications (including fault, configuration and security management) could also be performed. Upon arriving at their remote domains, MDMs may autonomously make management decisions and take actions based on the values of collected MIB values (for instance when the value of an aggregation function of several MIB objects exceeds a pre-specified threshold). These actions may include first-line support to handle trivial faults, decisions to recalibrate the management system as a response to changes sensed to traffic patterns or network configuration, e.g. to share the management responsibility of its domain with another MDM, move to a nearby domain, terminate execution, etc.

6.5. EXPERIMENTAL RESULTS

In order to evaluate the performance of the proposed hierarchical model, we have conducted a number of experiments, using the same testbed described in Section 5.4.2, i.e. a PC playing the role of the manager and 10 PCs with active MAS servers, simulating the managed devices. In the experiments described below, we make the assumption that the managed devices are separated from the manager platform by a WAN link (see Figure 6.15), although they all actually reside on the same network segment.

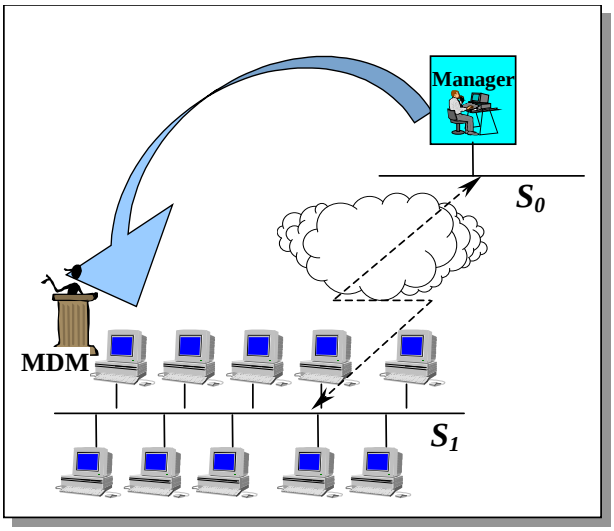


Figure 6.15. The experimental testbed

The examined scenario involves the collection of a data sample from each host at every PI; the size of each data sample can either be 50 (Figure 6.16) or 2000 bytes (Figure 6.17). We compare the performance of flat management, GnG (employing 2 MAs) and GnS polling (for data delivery frequencies of 10 or 100 PIs) against that of the hierarchical model. Regarding GnS polling, we have applied the *cloning* data delivery method described in Section 5.2.2.1, as that was proved the most efficient according to the experimental results reported in Section 5.4.2.

Referring to hierarchical management, a number of data collection approaches have been evaluated. In particular, the monitoring process is performed either through applying GnG (launching one or two MAs) or GnS polling (with data delivered every 10 or 100 PIs). Aggregated results are sent to the central manager with frequency of F_{del} = 100 PIs.

The deployment overhead associated with the MDM's accompanying *jar* file (23.4 Kb) has also been taken into account. The monitoring task is carried out by an MA object, whose definition (class file) amounts 2.2 KB. When employing, the GnG or GnS polling schemes, that file is multicasted to all active MAS entities, while in the hierarchical approach, the "tree multicasting" technique is applied (see Section 6.3.5).

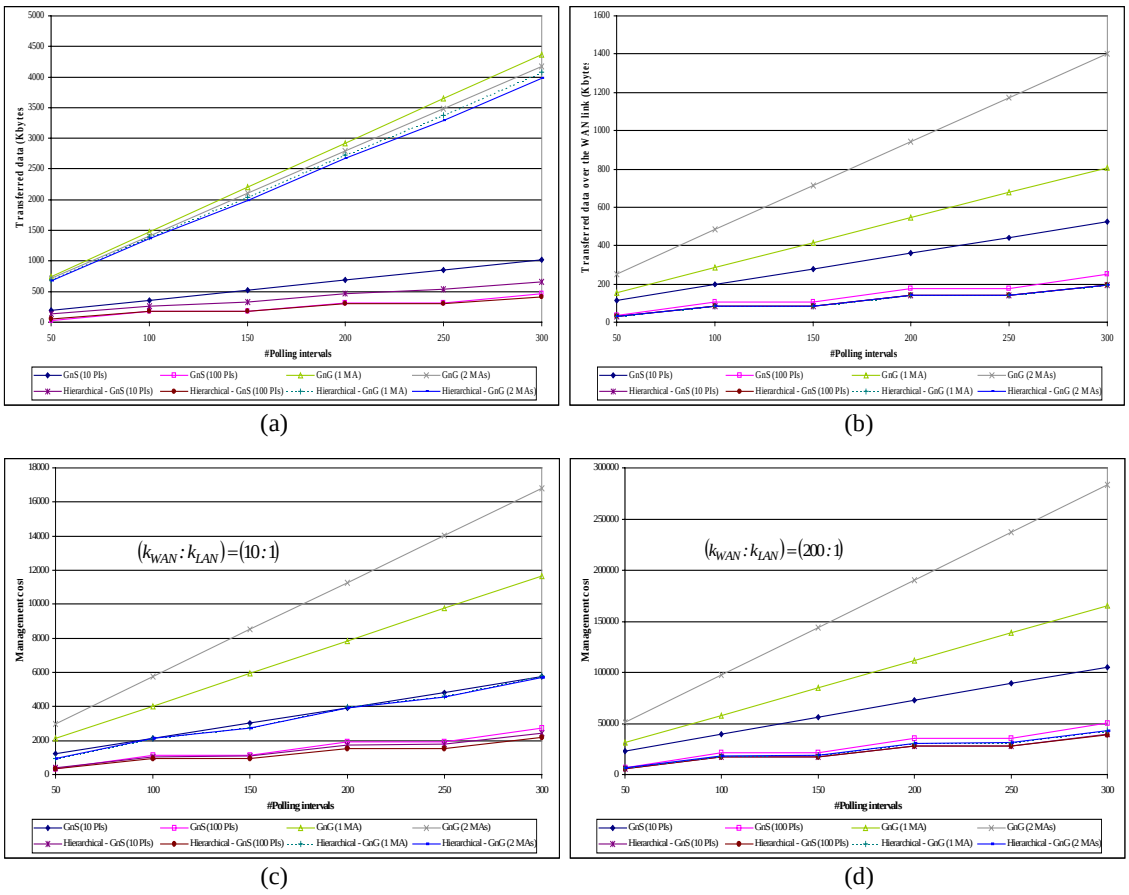


Figure 6.16. Network overhead measurements for data samples of 50 bytes: (a) Overall management traffic, (b) Network traffic through the WAN link, Overall management cost in the case that the cost coefficient for the WAN link is (c) 10, or (d) 200 times higher than for the high-speed LAN

The overall network traffic is illustrated in Figure 6.16a and Figure 6.17a, as a function of the number of PIs. As expected, the plain GnG approach scales worse in all cases. The combination of hierarchical model with GnG polling slightly reduces network overhead as it localises MA transfers within the MDM's local domain. Accordingly, the coupling of hierarchical management with GnS polling improves the performance of the plain GnS approach (that has also been expected).

The suitability of our hierarchical model for the management of geographically dispersed networks is depicted in Figure 6.16b and Figure 6.17b, which report the data volumes transferred over the 'WAN link' connecting subnets S_0 and S_1 (see Figure 6.15). Since that link does not physically exist, the curves actually depict the traffic sent/received by the manager host. It is noted that the various flavours of the hierarchical result in transferring identical data volumes, as the traffic over the 'WAN link' is restricted in aggregated data deliveries from the MDM to the manager.

An invited side-effect of the hierarchical approach is that following the MDM deployment, transferred NSM data are associated with the management of only *N*-1 hosts, out of the overall *N* managed devices. That is because the MDM resides on a managed NE, with all the management interactions between the former and the later taking place locally. For that reason, whenever the MDM applies GnG polling, its hosting device is visited *last* by travelling MA objects in order to avoid an unnecessary migration.

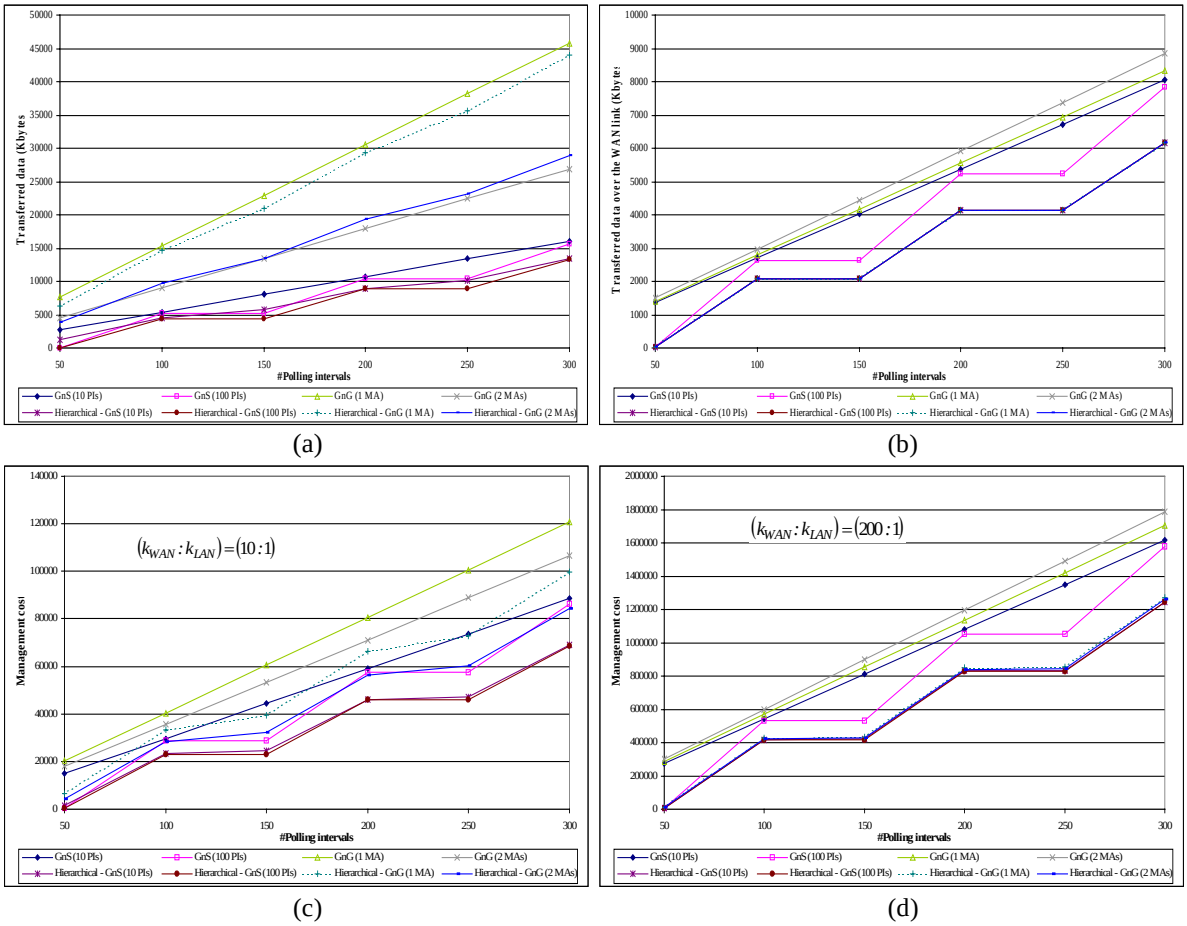


Figure 6.17. Network overhead measurements for data samples of 2000 bytes: (a) Overall management traffic, (b) Network traffic through the WAN link, Overall management cost in the case that the cost coefficient for the WAN link is (c) 10, or (d) 200 times higher than for the high-speed LAN

The network traffic comparisons discussed so far have not taken into account the increased cost associated with transferring data over a low-bandwidth link rather than over high-speed links. Figure 6.16c,d and Figure 6.17c,d illustrate the overall management cost when the ratio of the WAN link over the LAN link cost coefficients is $(k_{S_o,S_1} : k_{S_1,S_1})$=(10:1) or (200:1). It can now be observed that the coupling of the hierarchical model with GnG polling represents a cost-effective approach, as MA transfers are not performed over the 'expensive' link. However, the combination with GnS polling remains the most efficient solution. To further reduce the utilisation of the 'WAN link' and, hence, the overall management cost, MDMs

should apply a superjacent level of data aggregation, delivering to the manager only higher-level information.

The results graphically illustrated in Figure 6.16 and Figure 6.17 are also analytically presented in Table 6.1.

# Polling Intervals	GnS (10 PIs)	GnS (100 PIs)	GnG (1 MA)	GnG (2 MAs)	Hierarchical - GnS (10 PIs)	Hierarchical - GnS (100PIs)	Hierarchical - GnG (1 MA)	Hierarchical - GnG (2 MAs)
	Overall network traffic, Kbytes (data sample: 50 bytes)							
50	197	33	748	716	133	59	694	677
100	361	178	1473	1408	262	179	1391	1358
150	524	178	2198	2101	335	179	2034	1985
200	688	323	2923	2793	464	300	2732	2666
250	852	323	3647	3485	538	300	3375	3293
300	1016	468	4372	4178	666	420	4073	3974
	Network traffic over the WAN link, Kbytes (data sample: 50 bytes)							
50	115	33	154	253	30	30	30	30
100	197	106	285	483	84	84	84	84
150	279	106	416	713	84	84	84	84
200	361	178	547	943	139	139	139	139
250	442	178	678	1173	139	139	139	139
300	524	250	809	1403	194	194	194	194
	Overall network traffic, Kbytes (data sample: 2000 bytes)							
50	2704	33	7654	4501	1261	59	6347	3842
100	5374	5237	15285	8979	4511	4450	14691	9682
150	8045	5237	22915	13457	5713	4450	20987	13473
200	10715	10442	30546	17936	8963	8840	29331	19313
250	13386	10442	38177	22414	10165	8840	35627	23105
300	16057	15646	45808	26892	13415	13231	43972	28944
	Network traffic over the WAN link, Kbytes (data sample: 2000 bytes)							
50	1368	33	1407	1495	30	30	30	30
100	2704	2635	2791	2966	2078	2078	2078	2078
150	4039	2635	4175	4437	2078	2078	2078	2078
200	5374	5237	5558	5909	4127	4127	4127	4127
250	6710	5237	6942	7380	4127	4127	4127	4127
300	8045	7839	8326	8852	6175	6175	6175	6175

Table 6.1. Comparison of network traffic generated by GnG and GnS polling schemes against hierarchical management approaches

6.6. SUMMARY

This chapter has introduced the concept of *adaptive* hierarchical management that provides a rationale for the use of MA technology. A synopsis of the main contributions of the proposed model follows:

(a) The hierarchical architecture is intrinsically dynamic by employing mobile mid-level managers (MDMs) that may transparently move to a specific network domain to take over its management responsibility and localise the associated traffic. Although hierarchical MA-based management is not an entirely new concept (see [LIO98, SAH98, ZAP99]), our infrastructure goes one step beyond by offering improved adaptability to changing

networking environments and defining concrete policies regarding network segmentation into management domains, MDMs deployment, explicit determination of domain boundaries, etc.

(b) Apart form their ability to move from a management domain to another, MDMs may also move within their managed domain. In particular, MDMs periodically inspect the resources availability of their managed nodes and choose to move and resume execution to the least loaded host.

(c) In addition to addressing flexibility issues, management scalability is also further improved by fully exploiting the benefits of agent mobility, as MDMs rely on MAs for the data collection process; these MAs apply filtering operations locally, thereby minimising the volume of data transferred within the individual management domains. The cost associated with the distribution of code implementing new management tasks, is also minimised by applying an efficient "tree multicasting" bytecode distribution technique.

(d) The hierarchical NSM approach implies localisation of network traffic in the individual management domains, thereby providing more balanced traffic distribution and reducing the overhead in the area around the manager station.

(e) Our proposed design ideas are supplemented by a prototype that helped on gaining hands-on experiences and revealing problems related to the use of MAs in distributed management applications.

(f) Fault tolerance issues have been addressed, securing that distributed MDM objects continue to perform their decentralised tasks even if an interconnecting link or the manager platform fails.

(g) An analytical quantitative evaluation oriented to our specific framework design has been undertaken, deriving formulations that define the management cost associated with the proposed architecture.

A prototypical implementation of the introduced MAF has been tested in a realistic topology scenario, comparing its performance against flat MA-based management and plain GnG/GnS polling schemes. The experimental results section, which complements the quantitative evaluation study, has shown that the proposed architecture outperforms the other candidate approaches with sufficient distinct, both in terms of the overall management cost and the bandwidth usage of low-bandwidth WAN links.