# Evaluation of Constrained Mobility for Programmability in Network Management

*C. Bohoris, A. Liotta, G. Pavlou*
*Center for Communication Systems Research*
*School of Electronic Engineering and Information Technology*
*University of Surrey, Guildford, Surrey GU2 7XH, UK*
*{C.Bohoris, A.Liotta, G.Pavlou}@eim.surrey.ac.uk*

## Abstract

In recent years, a significant amount of research work has addressed the use of code mobility in network management. In this paper, we introduce first three aspects of code mobility and argue that *constrained* mobility offers a natural and easy approach to network management programmability. While mobile agent platforms can support constrained mobility in a rather heavyweight fashion, optimized approaches such as our CodeShell platform presented here can provide performance and scalability comparable to those of static distributed object platforms such as Java-RMI and CORBA. Properly implemented constrained mobility is thus of great importance in network management, resulting in flexible, extensible, programmable systems without prohibitive performance overheads.

## Keywords

Code Mobility, Mobile Agents, Java-RMI, CORBA, Performance Evaluation

# 1. Introduction and Background

Network management has been the subject of intense research over the last decade, with the relevant progress being twofold: on the one hand, approaches and algorithms for solving management problems have been devised; and on the other hand, different management technologies have been proposed and standardized. From the protocol-based approaches of the early 90's, exemplified by the Simple Network Management Protocol (SNMP) [1] and OSI Systems Management (OSI-SM) [2], the focus moved to distributed object-based approaches in the mid to late 90's, exemplified by the Common Object Request Broker Architecture (CORBA) [3] and more recently by Java's Remote Method Invocation (Java-RMI).

The paradigm of moving management logic close to the data it requires is a technique that has been conceived early in the evolution of management architectures, the relevant framework known as "management by delegation" [4]. Subsequent research showed the applicability of this concept in the context of OSI-SM [5] with a similar approach subsequently standardized, the Command Sequencer Systems Management Function (SMF). More recently, the same concept has been proposed in the context of SNMP through the IETF Script MIB [8]. While such approaches are specific to the respective management frameworks, the most general approach to delegation in the context of distributed object frameworks is through *object mobility*. Mobile objects are usually termed *Mobile Agents* (MAs) when they act on behalf of other entities and exhibit properties such as autonomy, reactivity, and proactivity.

The emergence of mobile agent frameworks has led many researchers to examine their applicability to network management and control environments. [6] considered code mobility in management and presented a taxonomy of the relevant aspects while [7] discussed the general issues of using mobile agents for network management. Since then a number of other researchers have attempted to use mobile agents in order to solve better specific network management

problems. Despite the research efforts until now, there have been little encouraging results regarding the exploitation of "strong" mobility in network management, as defined below.

Mobile agents may move around the network in a reactive or proactive adaptive manner and clone / destroy themselves according to their intelligence. We term this situation "strong mobility" and it is this property that has not yet been shown to achieve better results that static approaches in network management. An alternative possibility for mobile agents is to move from node A to B, typically guided by a "parent" stationary agent, and stay there until their task is accomplished. We term this situation "constrained code mobility" and we believe it is this simpler approach that can be readily exploited in management environments. In this case, instead of predicting the required functionality, standardizing and providing it through static objects in network elements or management systems, mobile code can support it in a dynamic, customizable fashion. The key advantage in this case is that the target node needs only to provide the required "bare-bones" capability which could be dynamically augmented through mobile code, with the additional logic able to change to reflect evolving requirements over time. Such a capability would obviate the use of functionality such as the OSI-SM Systems Management Functions (SMFs) and similar functionality provided in SNMP agents.

The key advantage of constrained code mobility is that it provides the vehicle for programmability through the enhancement of pre-existing functionality in the target node. In [15] we showed how constrained mobility can be used for programmable management systems which can be customized by clients for dynamic connectivity management services. In [9] we showed how the same principle can be used to enhance network elements with add-on functionality for performance monitoring. In both cases we used a general purpose mobile agent platform in order to support constrained mobility. A brief performance comparison with static object platforms in [15] showed that mobile agent solutions are rather heavyweight for constrained code mobility and this led us to the design and implementation of the CodeShell platform (section 3).

In this paper we layout the concepts of constrained, weak and strong code mobility in the context of network management and provide a detailed experimental evaluation of three different approaches to distributed management: 1) static distributed management based on Java-RMI and CORBA respectively as distributed object platforms; 2) dynamic distributed management based on CodeShell, an optimized mobile code platform supporting the constrained mobility paradigm; and 3) dynamic distributed management based on Grasshopper, a general-purpose mobile agent platform.

In section 2 we provide an overview of the general application of code mobility to network management and layout the concepts of constrained, weak and strong mobility. In section 3 we describe the CodeShell platform; this section also serves to identify the architectural aspects and necessary support for constrained code mobility. In section 4 we report on our performance experiments among static distributed object platforms, the CodeShell platform for constrained mobility and the Grasshopper mobile agent platform [14]. We close with our summary and conclusions which show that constrained mobility, if implemented efficiently, leads to performance comparable to the one obtainable with static distributed object solutions, achieving the same level of scalability while providing at the same time an easy and natural approach to programmability with all its associated advantages.

## 2. Code Mobility in Management

The key benefits code mobility may bring into the network management arena, for each of the five management functional areas, are identified in [7]. These benefits include reduction in network traffic, efficient utilization of computational resources, support for heterogeneous environments, and increased flexibility. Nevertheless, the use of mobile code does not come without costs. In particular, code migration incurs additional traffic into the network, absorbs considerable resources from the agent hosts, and is associated with migration delays of the order

of seconds or even tens of seconds, depending on the agent configuration and functionality [16] (see also Section 4.3). Code migration overheads often outweigh its benefits and make this approach inconvenient. It is therefore important to identify the various aspects of code mobility and relate them to network management in order to identify aspects that are particularly beneficial.

In the following subsections we define three different types of code mobility, ranging from the simplest, lightweight form of mobility to the most heavyweight one. For each case we elaborate on its benefits and limitations, identifying advantageous scenarios.

## 2.1 Constrained Mobility

One of the most elementary forms of code mobility is defined in [6] as Remote Evaluation (REV), after the pioneering work described in [17]. In REV, an application in the client role can dynamically enhance the server capability by sending code to the server. Subsequently, clients can remotely initiate the execution of this code that is allowed to access the resources collocated within the server. Therefore, this approach can be seen as an extension of the client-server paradigm whereby a client in addition to the name of the service requested and the input parameters can also send code implementing new services. Hence the client owns the code needed to perform a service, while the server offers both the computational resources required to execute the service and access to its local resources.
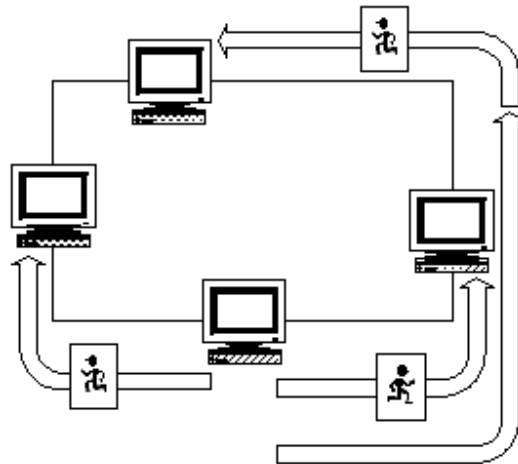


**Figure 1: Constrained mobility. The agent is created and initialized by a client application and is then shipped to an agent host. The agent execution is then confined to this host.**

A natural evolution of the REV model involves sending code not restrained to be a remote service but which can also act as a fully autonomous software entity. This type of code mobility we term *constrained mobility* since the code, upon its creation at a client site, is only allowed to migrate to a remote server where its execution will be confined.

When constrained mobility is adopted in management the code is created by a client acting in the manager role and is, then, dispatched to a target network element acting in an agent/server role (Figure 1) – in this case the term *agent* is used according to the manager-agent model rather than denoting a mobile agent.

This approach is particularly suited to dynamically programming or upgrading network devices. In this case, the code does not need to be particularly sophisticated. In a simple scenario it could be as simple as collections of objects that can be executed in a remote virtual machine. Therefore, mobility degenerates into a simple dynamic mechanism to efficiently deploy or upgrade network protocols or services.

3

Code deployment overheads, namely *deployment traffic* and *delay*, represent the drawbacks of general MA approaches. In constrained mobility the code does not need to incorporate complex migration features since its destination is predefined at creation time and is not changed afterwards. As a result, its size and the incurred network traffic are minimal compared to the other forms of mobility (see sections below). Similarly, it will not be necessary to use general purpose MA platforms – usually associated with heavyweight migration mechanisms [16] – and, thus, the code migration time can be considerably reduced (see Section 4.5).

In conclusion, constrained mobility is a particularly well-suited mechanism to dynamically program network elements. It can outperform traditional centralized management for data-intensive tasks and when high degrees of semantic data compression need to be achieved – e.g., through data aggregation or analysis. Constrained mobility is typically advantageous to perform off-line analysis of bulk data and, more generally, to implement tasks whose duration is at least comparable with the overall agent deployment time.

## *2.2 The Weak and Strong Mobility Models*

### Weak Mobility

Similarly to constrained mobility, in weak mobility the code is created and initialized by a client application and is, then, shipped to its host. However in the latter, code is not confined to that host since it its meant to perform the same task in more than one location. Mobile agents that follow the weak mobility model do not retain any knowledge of the data processed or of the actions performed in previously visited hosts and, consequently, they can only implement tasks in which this information is not required.

A convenient use of weak mobility is for dynamic decentralization of management tasks that are otherwise performed in a centralized fashion. The agent is delegated part of the management responsibility and will incorporate functionality such as procedures aimed at data semantic compression or aggregation.

A trivial example showing the main advantages of weak mobility is the case in which the management station has to search for a single value in a table, a data structure typically used to store information inside devices. In SNMP management the whole table has to be transferred from the remote element to the management station, where the table rows are searched for the value. Hence, large tables will incur heavy unnecessary traffic into the network and will result in computational overload on the management station.

A more efficient approach is adopted by OSI Systems Management [2], which supports remote scoping and filtering operations. Thus, the searching logic is executed in the device and, consequently, only the retrieved value is transmitted to the manager. The drawback of this approach is that the logic implementing scope and filtering has to be "hard-wired" in the network elements, which tend to become more complex. In addition, such functionality needs to be agreed upon and standardized beforehand.

If constrained mobility were to be used, the agent incorporating the search routine would be shipped to the network device, where it would retrieve the requested values from the local table and return them to the manager. This solution addresses the shortcomings of the OSI approach, retaining its scalability and performance benefits. However, constrained mobility is not suitable in the more general case in which similar tasks are to be run on multiple network elements. In fact, as the number of network elements grows, the management station will be overloaded and the network capacity around it will be saturated by the simultaneous generation and transmission of the agents.

**Strong Mobility**

With strong mobility, agents are able to access and process data from network elements but can also accumulate information and preserve it upon migration. This feature allows for the implementation of more elaborate tasks in which the agent operations depend on data gathered in previously visited hosts.

In network management, strong mobility is more suited to configuration tasks and to data-intensive tasks involving data aggregation from highly distributed network elements and on-line data analysis. A simple example is a task involving the collection of utilization information from a relatively large number of network elements. In a traditional SNMP-based system, the management station has to poll every single element in order to collect the required raw performance information before it can produce a useful utilization rate. OSI management offers a more efficient mechanism for obtaining the utilization rates, as this is done locally at the network element, but still requires further aggregation of this information at the management station.

Constrained mobility does not suit this task since it would require the deployment of a number of MAs equal to the number of network elements. Each MA would typically be executing for a time negligible with respect to its deployment time and, then, the agent deployment overheads would be unacceptable.

With strong mobility, the agent will be able to preserve the utilization rates of previously visited elements and will then be able to perform a further level of data aggregation independently from the manager. The main drawback associated with strong mobility is the agent size. 'Strong' agents tend to incorporate more intelligence, being larger in size. More critically, the agent size can vary significantly depending on the amount of information that has to be preserved during migration. It is, therefore, important to design the agents in such a way to limit their size variations – e.g., by allowing only semantically compressed information to be carried.

Though in principle there should be uses for weak and strong mobility in network management, research work until now has not resulted in identifying compelling use cases. On the other hand, constrained mobility can be used for network management programmability and we believe that this should be done through optimized platforms such as our CodeShell one presented in the next section.

# 3. The CodeShell Prototype Platform for Constrained Mobility

## 3.1 Introduction

Constrained mobility requires at least the following two important facilities:

- A mechanism for migrating management logic along with initial parameters to a destination machine hosting the necessary resources.
- A naming service in order to distinguish between objects and also bind one object to another.

Mobile agent platforms include both these features but also many other features that allow them to be used as general purpose solutions for weak and strong mobility. As such, they are rather heavyweight and have scalability problems. On the other hand, the performance of distributed object frameworks such as Java-RMI and CORBA is more acceptable but these support only static objects that can communicate remotely with other objects in different network nodes. In order to build constrained mobility applications, a distributed object framework's communication and naming service facilities could be re-used. In addition, a thin layer of functionality is needed, allowing the migration of management logic and supporting naming and binding of objects. As a means of validating this approach, the CodeShell prototype platform was designed and

implemented providing such facilities over Java-RMI. The CodeShell platform supports constrained mobility and it is specifically tailored to network management.

Constrained mobility is particularly efficient for providing a number of programmable, customizable network management services. In the case of performance management for example, in order to collect performance data intended for off-line analysis, a performance monitor can be conveniently transferred near the resources of the node that needs to be monitored. The transfer of logic in this case allows the easy customization of the monitoring activities and the modification of monitoring characteristics in a dynamic way. In fault management, constrained mobility can be used to send event filtering and correlation logic to the node that is monitored for faults. Performance and fault management services along with a configuration management system, can be integrated and provided to customers in a dynamic, customizable fashion using the constrained mobility model as described in [15].

## 3.2 Design and Implementation

The CodeShell platform consists of the following components:

- CodeShell Communication Service (CCS): Allows the communication between remote objects. It is also responsible for the transfer of byte-code between two remote machines.
- CodeShell Naming Service (CNS): Provides functionality related to the names assigned to objects. All names and important object information are stored in a local database. This allows the lookup of objects and the binding between them for local communication.
- CodeShell Core System: Coordinates the operation of an individual CodeShell. Also provides a single interface allowing CodeShell objects to perform CCS and CNS related operations.
- Base CodeShell Object: Provides basic functionality that should be inherited by any object intended for use inside a CodeShell. Most functionality relates to basic CCS and CNS related operations.
- CodeShell Textual User Interface: A user interface that allows the user to manage (create, delete, list, etc.) objects within a CodeShell. This component is completely pluggable and can be easily detached from the main CodeShell system and replaced with an alternative environment.

A minimum typical scenario of operation involves two remote machines. One Java-RMI registry and one CodeShell are initialized on each machine (see Figure 2). In the CodeShell of the machine in the client role, a "master" CodeShell object is created. This object will then contact the CodeShell's CCS in order to send byte-code containing management logic to the remote machine {Step 1}. The CCS will then locate the CCS component of the remote CodeShell and transfer the byte-code and a list of initial parameters {Step 2}. When the byte-code arrives at the machine in the server role, a "logic" object is created from it and it is initialized with the provided parameters. In this CodeShell a "target" object is already created by the user waiting to provide the necessary raw resources. The "logic" object contacts the local CNS and performs a lookup for the "target" object {Step 3}. When this is located, the two objects are bound so that they can communicate locally with each other {Step 4}. The logic object is now ready to perform its task using the target object to obtain the necessary raw information {Step 5}. When it is time for a report to be sent back to its "master" object it will contact the CCS {Step 6} which will transfer it to the remote CodeShell {Step 7}. From there the local CCS is responsible for the report to be passed to the "master" object {Step 8}. The CodeShell platform was developed in Java using Sun's JDK and uses the runtime environment version 1.2.1.
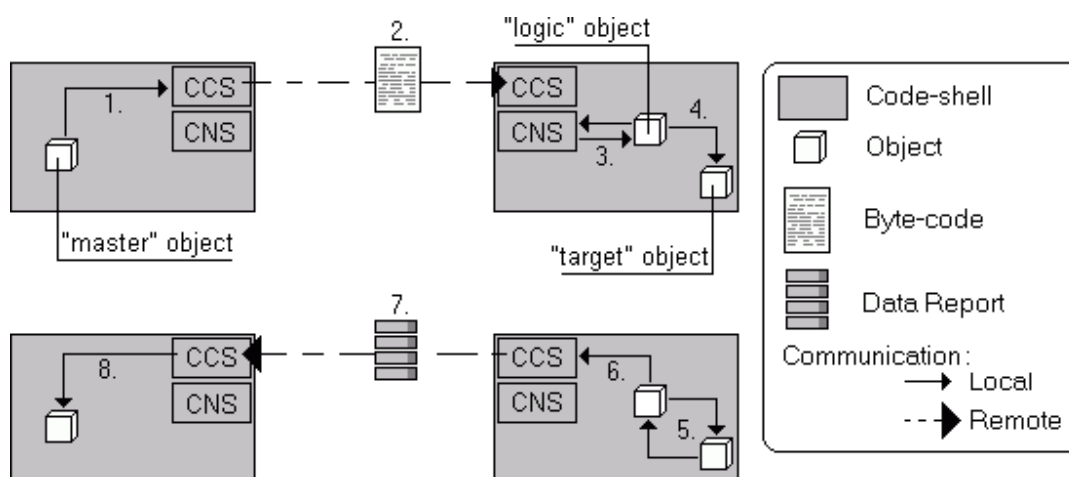
**Figure 2: A typical constrained mobility application scenario running inside two remote CodeShells.**

# 4. Experimental Evaluation

## 4.1 Case Study

In order to evaluate the performance overheads of Java-RMI, CORBA, CodeShell and Grasshopper respectively, we used the performance monitoring case study described in detail in [9]. The aim here is to provide traffic rates with thresholds, quality of service alarms and periodic summarization reports by simply observing raw information such as traffic counters in network elements. This is functionality similar to the OSI-SM metric monitoring and summarization facilities (X.739/X.738) [10][11] but when it is provided through code mobility, users of the service are also able to customize it according to the semantics of a particular application as explained in [9].

When constrained code mobility is deployed using either CodeShell or Grasshopper, a performance monitor object is created by a "master" object somewhere and is sent to execute within a target node. When Java-RMI and CORBA are used, the performance monitor object is created at the target node through an object factory and the relevant intelligence needs to pre-exist at that node. The performance monitor gathers information locally, applies thresholds and sends QoS alarms or periodic summarization reports to the master object.

The "target" object is actually an adapter for the underlying SNMP. For its implementation and operation the AdventNet SNMP version 2.0 libraries were used in order to query an SNMP agent for raw performance information. Given this functionality, we are interested to measure the creation/migration overheads, the cost of remote invocation that models the reporting of results, both in terms of response time and packet sizes, and the computing requirements at a the target node. Though our case study is specific to a particular problem domain i.e. performance management, the described measurements are general enough to give us insight on the overheads of the constrained mobility approach in network management in general.

## 4.2 Method

The performance monitoring system has been implemented over four different infrastructures, the Grasshopper mobile agent platform, Java-RMI, CORBA, and Code-Shell. The aim was to assess the impact that these underlying technologies may have on the monitoring system.

Grasshopper is not one of the most efficient MA platforms. It has been chosen because, due to its functionality, it can be considered a general-purpose MA platform. Moreover, Grasshopper

7

follows the current standardization directions, since it is compliant with both MASIF [13] and FIPA [12].

Java-RMI and CORBA have been chosen as representative of the most popular 'static' distributed object technologies. In addition, CORBA is emerging as a significant technology for network and service management.

Finally, Code-Shell is our implementation of a platform supporting constrained code mobility. It is an approach that lays in between two extreme solutions. In the first case the monitoring system is enabled with strong code mobility, which is supported by a general purpose MA platform. In the second case, the systems cannot rely on any form of code mobility since it is entirely based upon static distributed object technologies. For this reason, Code-Shell can be regarded as an optimized version of a general-purpose MA platform, which aims at achieving performance comparable to the one obtained with static distributed object technologies.

During our experiments we have run the same performance monitoring tasks over the four different platforms, measuring in each case the total response time, the traffic incurred in the network and the total memory requirements. In the cases of Grasshopper and Code-Shell the measurements have been taken at steady state, that is after the code had been shipped to the remote elements. In this way we could perform a direct comparison with the implementations based on Java-RMI and CORBA, respectively. The additional overheads incurred by code mobility – namely code deployment time and network traffic incurred during the transmission of the code from manager to network elements – were measured separately.

In order to measure the total remote invocation response times, timestamps were taken using the *currentTimeMillis* method of the *java.lang.System* class. An array of objects (class *java.util.Vector*) containing 25 numbers of type *java.lang.Double* was remotely transferred 100 times between two objects located in different machines, measuring the total transmission time. The same procedure was repeated while increasing the number of elements in the list to 50, 75, and 100. This operation in fact models the periodic summarization reports generated and remotely sent by the entity equipped with the performance monitoring logic.

For the same experimental cases and in order to calculate the total incurred traffic, we measured the TCP packet sizes using the *tcpdump* program originated at the Lawrence Berkeley laboratory, reporting the total payloads at the TCP level.

To measure the memory requirements, program sizes were measured for the server side of the Grasshopper, Code-Shell and Java-RMI systems. The measurements were taken using the *totalMemory*, and *freeMemory* methods of the *java.lang.Runtime* class. The first method provides the total amount of memory allocated by the Java Virtual Machine (JVM). The second one returns an approximate value of the amount of memory left free inside the JVM – i.e., memory available for future object allocation. The difference of these two values provides the amount of memory required by the performance monitoring system under evaluation which includes the required platform related classes in addition to the "target" and to the "logic" objects.

The experiments have been repeated 100 times for each of the above cases in order to perform a statistical analysis and study the significance of the measurements. The experiments where carried out using two different machines over a lightly loaded 100 Mbit/sec Ethernet in the role of the management network with the following specification: Sun Microsystems Ultra-10, 256MB of memory, Sun's Solaris 2.5.1 version of UNIX.

## 4.2 Response Times Measurements

The response time of management operations for each of the four cases of performance monitoring systems are reported in Figure 3 which, for an easier comparison, combines in a single chart the mean values and best liner fit of the results.
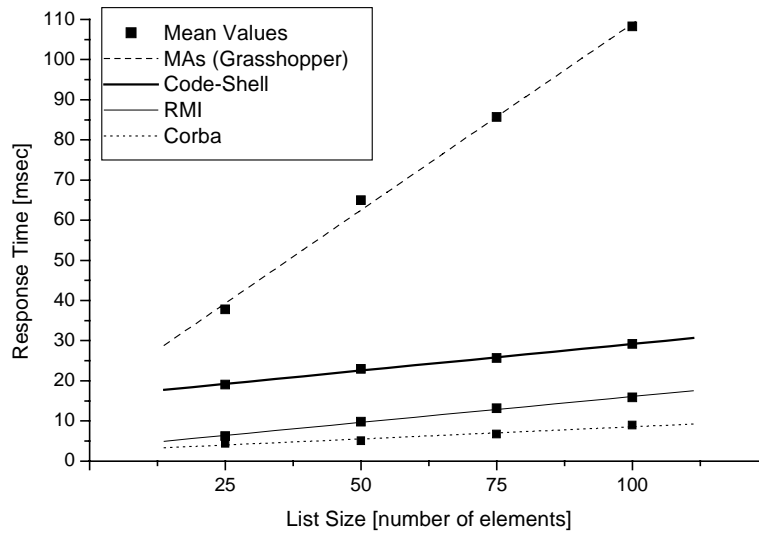
**Figure 3: Mean values and best linear fit of response times.**

The first conclusion that can be drawn by observing the plots is that the system based on Code-Shell exhibits the same degree of scalability as the one of the systems based on Java-RMI and CORBA. In fact, the slopes of the curves of those three cases have a comparable value. On the contrary, the Grasshopper system exhibited a much bigger slope showing its intrinsic inability to perform well under more demanding conditions.

From the performance point of view the Code-Shell system gave a response time in the order of 2-3 times larger than the one of the Java-RMI system and in the order of 4 times larger than the one of the CORBA system.

The conclusions based on Figure 3 have been validated by statistical analysis. Figure 4 depicts the results of this analysis in the form of statistical box charts. It should be noted that these boxes do not have overlapping values and this fact leads us to the conclusion that the response times of the four different solutions are indeed statistically different. Therefore, the mean values and the line slopes of Figure 3 are statistically representative and can be employed to carry out the above comparative analysis.
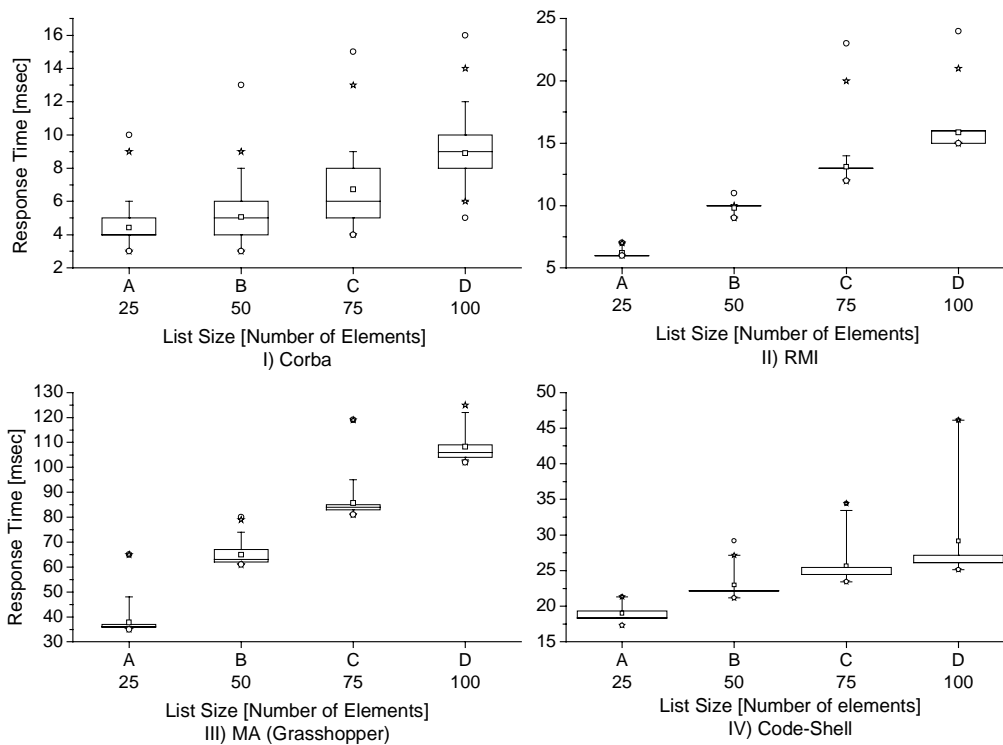
**Figure 4: Statistical Box Charts showing response times for each of the four experimented cases. The boxes include the 25-75% boundaries, the mean values (a black square) and the median values (a line). The 5-95% range boundaries are delimited by whiskers. The outliers are depicted with circles and stars.**

## *4.3 Traffic Measurements*

We also measured the packet sizes in all four cases. An array of objects (class *java.util.Vector*) containing 25, 50, 75 and 100 *"Double"* numbers respectively was remotely sent using remote invocations in the Mobile Agent, CodeShell, RMI and CORBA systems. Each time, the payload of the TCP packets was measured. A chart of the results gathered can be seen in Figure 5.

It is interesting to observe that, within the measured range of values, the four solutions incurred a comparable level of traffic. The Grasshopper and RMI systems performed better for small scales whilst the CodeShell and CORBA systems exhibited better performance for larger scales. The CodeShell platform transparently optimizes the transfer procedure and this is the reason why, for a large number of elements, it incurred less traffic in the network compared with the standard RMI system.
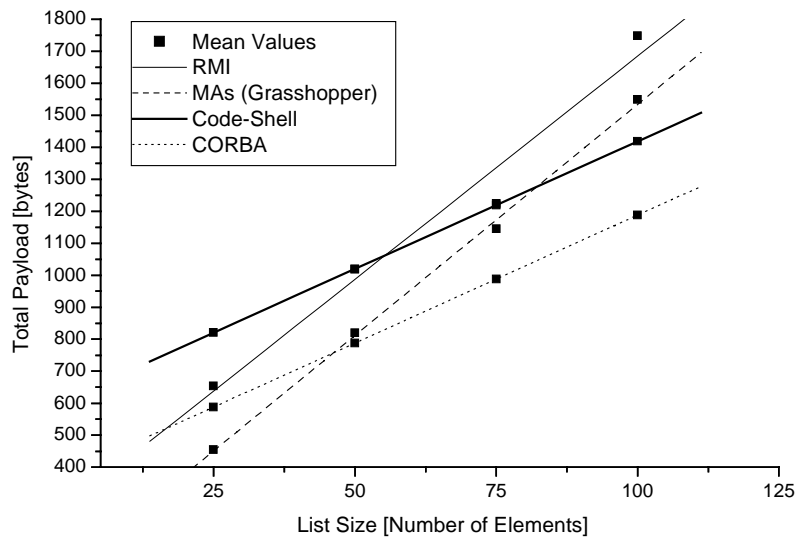
**Figure 5: Mean and best linear fit of total incurred TCP payloads, measured as the sum of all the bytes incurred in the network to complete the given network performance monitoring task.**

## 4.4 Memory Measurements

The memory requirements for the monitoring systems based on Grasshopper, Java-RMI, and CodeShell, are compared in Figure 6.
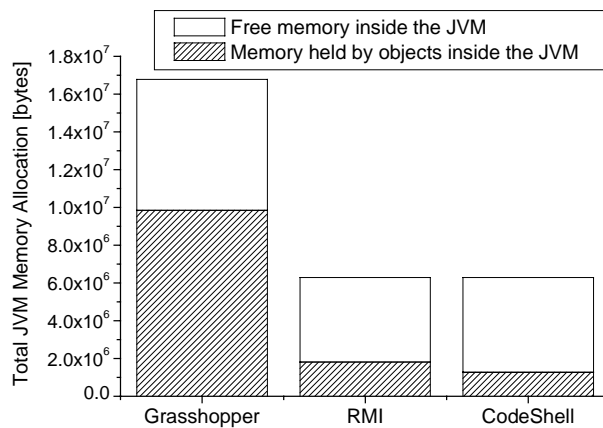


**Figure 6: Memory requirements for the Java-based network performance monitoring systems.**

It can be observed that CodeShell performs as well as Java-RMI and significantly better than Grasshopper. The latter, resulted in a fivefold occupation of memory which is yet another dramatic drawback of relying on a general-purpose MA platform.

## 4.5 Code Migration Overheads

The delay and traffic involved during code migration have been measured for the two programmable network performance monitoring systems based on Grasshopper and Code-Shell,

11

respectively. The times involved in the migration of performance monitoring logic from the manger to the network elements are reported in Figure 7 in the form of statistical box charts.

By substituting the generic code migration mechanism of Grasshopper with a simpler code deployment protocol we have been able to reduce by four times the time required to program a network element. The fact the statistical boxes do not overlap proves that the difference in code migration time between the two approaches is statistically significant.
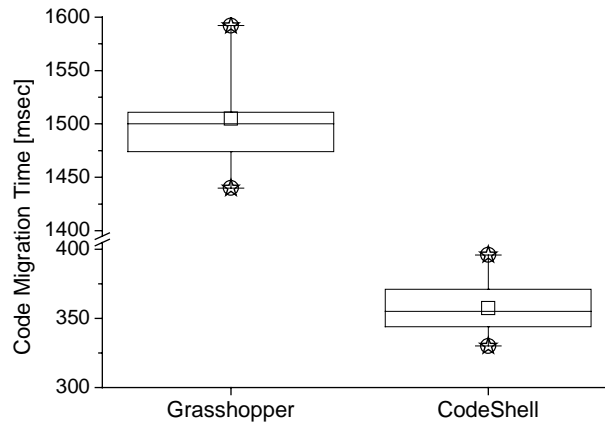


**Figure 7: Statistical Box Charts showing code migration times involved in the programmable network performance monitoring systems. The boxes include the 25-75% boundaries, the mean values (a black square) and the median values (a line). The 5-95% range boundaries are delimited by whiskers. The outliers are depicted with circles and stars.**

We also measured the additional traffic incurred by code migration. The transmitted data for the CodeShell system was 2,236 bytes; for the Grasshopper system it was 2,854 bytes. There was no need to repeat the measurements since we were able to measure the exact payload by discriminating it from the background traffic.

# 5. Discussion and Conclusions

Over the last three years we have been engaged in the task of designing and implementing an integrated Network Management system based on the use of Mobile Agent technologies. Our investigation led us to conclude that general-purpose MA platforms are not a viable infrastructure over which dynamic, programmable management systems can be realized. This is due to the fact that MA platforms tend to be rather heavyweight and do not scale well. In particular, code migration involves delays which in several cases are orders of magnitude larger than the timescales typical of network management systems. Moreover, following the MbD idea originated in 1991, nearly ten years of discussions in the management community have failed to identify a single case in which the use of strong mobility can have a significant impact on NM applications.

On the contrary, we believe that constrained mobility can be the vehicle to realize network programmability and MbD functionality with current technologies.

These observations have induced us to shift our investigation towards the constrained mobility concept in order to assess more precisely its effectiveness in the specific context of network management. We have implemented CodeShell in order to establish whether it was possible to realize management systems based on constrained mobility, achieving at the same time performance levels comparable to the ones of systems based on the most popular static distributed object technologies. We also aimed at quantifying the performance gain achievable by

giving up on general-purpose MA technologies, retaining only the most basic form of code mobility exemplified by constrained mobility.

The results presented herein suggest that constrained mobility is easily integrated in network management systems. Moreover, using constrained mobility it is still possible to achieve performance and scalability typical of static distributed object technologies.

To draw these conclusions we have realized a performance monitoring systems over CodeShell, a constrained mobility platform based on Java-RMI. We believe that other management functions such as configuration and fault management can similarly benefit from constrained mobility.

While mobile agent frameworks were initially thought as rivals to static distributed object frameworks, the two approaches need to coexist. We believe that constrained mobility is the required level of code mobility that can find concrete application in network management. Real synergy could be achieved if stationary agents could be provided using static objects, with method invocations being possible between mobile and static objects in both directions. Such an environment would combine the best of both worlds but it is not clear at present whether this type of seamless integration is achievable.

## REFERENCES

[1] J.Case, M.Fedor, M.Schoffstall, J.Davin, *A Simple Network Management Protocol (SNMP)*, IETF RFC 1157, 1990.

[2] ITU-T Rec. X.701, Information Technology - Open Systems Interconnection, *Systems Management Overview*, 1992.

[3] Object Management Group, *The Common Object Request Broker: Architecture and Specification (CORBA)*, Version 2.0, 1995.

[4] Y. Yemini, G. Goldszmidt, S. Yemini, *Network Management by Delegation*, in Integrated Network Management II, Krishnan, Zimmer, eds., pp. 95-107, Elsevier, 1991.

[5] N. Vassila, G. Pavlou, G. Knight, *Active Objects in TMN*, in Integrated Network Management V, Lazar, Saracco, Stadler, eds., pp. 139-150, Chapman & Hall, 1997.

[6] M. Baldi, S. Gai, G.P. Picco, *Exploiting Code Mobility in Decentralised and Flexible Network Management*, Proc. of the 1st International Workshop on Mobile Agents 97 (MA'97), Berlin (Germany), K. Rothermel and R. Popescu-Zeletin eds., April 1997, Springer, Lecture Notes on Computer Science vol. 1219, ISBN 3-540-62803-7, pp. 13-26.

[7] A. Bieszczad, B. Pagurek, T. White, *Mobile Agents for Network Management*, IEEE Communications Surveys, Vol. 1, No. 1, http://www.comsoc.org/pubs/ surveys, 4Q1998.

[8] J. Schoenwaelder, J. Quittek, *Script MIB Extensibility Protocol Version 1.0*, RFC 2593, May 1999.

[9] C. Bohoris, G. Pavlou, H. Cruickshank, Using Mobile Agents for Network Performance Management, Proc. of the IEEE/IFIP Network Operations and Management Symposium (NOMS '00), Hawaii, USA, J. Hong, R. Weihmayer, eds., pp. 637-652, IEEE, April 2000.

[10] ITU-T Rec. X.739/X.738, Information Technology - Open Systems Interconnection, *Systems Management Functions - Metric Objects and Attributes/Summarization Function, 1992/*1993.

[11] G. Pavlou, G. Mykoniatis, J. Sanchez, *Distributed Intelligent Monitoring and Reporting Facilities*, IEE Distributed Systems Engineering Journal (DSEJ), Special Issue on Management, Vol. 3, No. 2, pp. 124-135, IOP Publishing, 1996.

[12] Foundation for Intelligent Physical Agents, web page: http://www.fipa.org/.

[13] Object Management Group, *Mobile Agent System Interoperability Facilities Specification*, orbos/97-10-05, 1997, ftp://ftp.omg.org/pub/docs/orbos/97-10-05.pdf

[14] The Grasshopper Agent Platform http://www.ikv.de/products/grasshopper/index.html.

[15] D. Griffin, G. Pavlou, P. Georgatsos, *Providing Customisable Network Management Services Through Mobile Agents*, Proc. of the 7th International Conference on Intelligence in Services and Networks (IS&N'00), Athens, Greece, G. Stamoulis, A. Mullery, D. Prevedourou, K. Start, eds., pp. 209-226, Springer, February 2000.

[16] G. Knight, R. Hazemi, *Mobile Agents based Management in the INSERT Project,* Journal of Network and Systems Management, Vol.7, No.3, pp. 271-293, September 1999.

[17] J.W. Stamos, D.K. Gifford, *Remote Evaluation,* ACM Transactions on Programming Languages and Systems. 12(4), pp.537-565, October 1990.