Object Transport and Reference Management in the MobJeX Framework

Stephen Perry

Supervisor: Dr. Caspar Ryan

Honours Thesis

School of Computer Science and Information Technology
RMIT University
Melbourne, AUSTRALIA

October, 2003

Abstract

With increasing numbers of more capable mobile devices, and the pervasiveness of more powerful desktop machines, improvements in the design of client server systems may be possible. In a next generation Internet environment, client server systems should be able to take advantage of currently under utilised resources on client devices. The heterogeneous nature of clients means that for an application to take full advantage of all available resources, it must adapt dynamically to suit its environment.

From an examination of existing methods, this thesis provides a rationale for mobile objects as a means for efficient adaptation and usage of client resources. We provide an analysis of several existing frameworks attempting to address the issues involved in dynamic adaptation using mobile objects, and develop a solution for object transport and reference management in the recently proposed MobJeX framework.

Contents

1 Background and Rationale **4**

2 Literature Review **6**

2.1 Client- and Resource-Aware Adaptation 6

2.2 Application Adaptation 6

2.2.1 Adaptation using Mobile Code 7

2.3 Techniques for Object Mobility 8

2.3.1 Reference Management 8

2.3.2 Object Grouping 10

2.3.3 Object Substitution 11

2.4 MobJeX 12

2.4.1 MobJeX Architecture Overview 13

2.4.2 Distribution Manager 13

2.4.3 Transport Manager 14

2.4.4 Resource Monitor 14

2.5 Summary 15

3 Project Goals and Contributions **15**

4 Methodology **16**

4.1 Overview 16

4.2 Reference Management 17

4.2.1 Object Creation 18

4.2.2 Object Migration 20

4.2.3 Object Substitution 22

4.2.4 Proxied Object Usage 22

4.2.5 Non-Moveable Objects 23

4.3 Serialization 23

4.4 Transparency 24

4.5 Remote Class-loading 25

4.6 Limitations 26

5 Results **27**
5.1 Test 1 - Object Creation Efficiency 27
5.2 Test 2 - Object Migration Efficiency 28
5.3 Test 3 - Remote Object Method Invocation Efficiency 29
5.4 Test 4 - Object Location Transparency and Reference Integrity 30
5.5 Test 5 - Object Substitution 31
5.6 Limitations 31

6 Summary and Conclusion **31**
6.1 Future Work 32

A Application code modification example **36**

1 Background and Rationale

In traditional client-server systems, the majority of processing and data management is delegated to the server, with the client presented with the 'view' of the service provided by the server. This is primarily because clients lacks the resources (data, processing power, memory, software) needed to provide the required service [BP98]. Managing a centralised server is also more convenient than maintaining many separate databases and high powered clients.
With the advent of more powerful mobile devices, such as Personal Digital Assistants (PDAs), and Smart Phones, and the pervasiveness of high-powered client desktop devices, the possibility of transferring more responsibility to client machines exists. Many clients are now able to perform tasks locally that were previously performed by a high powered remote server. However in certain situations, client devices may not have the software resources to take on this responsibility, so the need arises to dynamically provide the clients with the knowledge (code and data) needed to perform the task locally. This capability known as *code mobility* opens up the opportunity for developers to utilise under used client hardware resources instead of relying on a resource heavy server to support applications.
In a *next-generation Internet environment (NGIE)* [IBM03], where Internet connectivity will be pervasive and essential part of our daily lives, heterogeneous mobile clients will be common, each with varying software and hardware support. Unfortunately, current wireless networking technologies do not provide cheap, reliable, high speed bandwidth to mobile users, and it is unlikely that these technologies will be available in the near future. Devices limited by their network technologies are considered to have *weak connectivity* [MES95] and may not be able to fully support applications where high bandwidth is needed. Some devices may also lack memory or processing power, and thus are unable to perform many of the functions required by applications.

Another possibility with the increasing power of mobile devices is the ability to develop *Migratory Applications* [BC95]. Migratory Applications allow users to 'swap' their application from one client device to another, whilst maintaining their current session within the application. A typical scenario would be when the user is using an application on their desktop client, and needs to leave the office. Rather than closing the application, synchronising data with their other device, and then re-opening the application on their laptop or PDA, the application could be automatically swapped over to the other device at the click of a button. This requires the application to adapt to each device it migrates to, possibly retaining some components on the originating device and moving only components that the receiving device can support.

To efficiently utilise client resources, and provide the most effective functionality to the user, the application must be aware of it's operating environment, and the resources available to applications, and be able to adapt accordingly. This allows a range of heterogeneous clients to efficiently run applications under a diverse range of conditions. This may involve dynamic distribution of code to clients to ease the load on the server, or relocation of the application components when the client loses reliable network access. Reconfiguration could also occur when a less capable device device begins to run low on resources, or when a device's network throughput changes or becomes unreliable. Additionally, re-configuration could occur after a component of the system is moved from one device to another. These systems can be described as *resource-aware*.

Resource and client aware adaptive systems offer performance benefits to users by efficiently distributing functionality over available devices, but also offers benefits to developers and service providers.
From the perspective of application developers, writing a distributed application supporting a diverse

range of client operating platforms may require the maintenance of multiple code bases, possibly spread across a range of technologies. It is for this reason that client applications are kept as thin as possible, with the majority consisting of presentation markup such as XHTML or WML [WAP00]. Thin client systems usually lack the ability to support heavier application components, and thus rely on a server capable of taking on this burden for all client devices.

Although attractive to developers, the thin client approach requires Application Service Providers to maintain powerful servers and reliable bandwidth to cater for an increasing amount of clients [RP03b]. The need to scale services proportional to the number of clients results in higher costs in keeping up with increasing demand, and limits scalability of the application services provided. Therefore, the ability to make use of client resources where available is attractive, as this would allow distribution of load across all devices, rather than focusing the load on servers. From an administrative point of view, it is also beneficial to dynamically load clients with application components, allowing tighter control over code distribution and maintenance. It is also desirable for rules controlling adaptation of applications to be separate from application code, to allow application distribution strategies to be specified at deployment time by the service providers.

The execution of main application components on the client device allows applications to be less reliant on reliable network connectivity.

Local application object access reduces the applications need to access remote services, lowering bandwidth costs and response times, resulting in a better user experience of the application. It has been shown that response times can be greatly improved when processing tasks are delegated to the client instead of the server [RP03b]. Dynamic adaptation of the layout of components in the system also allows applications to be responsive to changing operating conditions and device heterogeneity.

Existing frameworks supporting client and resource aware adaptation of applications provide a varying degree of adaptation, ranging from transformation of the presentation layer and content to device specific format, to application component migration in response to changing resource availability. Content transformation lacks the flexibility required to adapt applications with rich user interfaces, as adaptation is usually limited to transformation of markup such as the conversion from XHTML to WML, or the filtering of redundant content. Existing mobile object adaptation systems allow the migration of components between devices, but lack support for the heterogeneity of client devices and developer transparency.

This lack of existing support provides the rationale for a cross-platform framework supporting adaptive applications to address the problems associated with applications in a next generation Internet environment. Given this need, a core component of such a framework is implemented and tested in this thesis. To further support our rationale for this research, section 2 provides an extensive literature review in order to introduce the major issues and existing work relevant to adaptive applications and code mobility. Limitations of existing work are identified, providing rationale for the MobJeX framework [RP03b] for adaptive next generation Internet applications. Following an overview of the MobJeX framework, a solution for the core object transport component of MobJeX is presented, and results of efficiency testing are discussed, in order to address the shortcomings of existing work described in section 2.

2 Literature Review

There exists a significant amount of prior work relating to the use of mobile objects in adaptive applications. Many techniques for the management of mobile objects in adaptive systems exist, each with varying suitability for a NGIE. Existing systems of relevance to this thesis include FarGo [HBSG99] and Sumatra [ARS97], systems both aimed at adaptation using object mobility. In order to assess their suitability for use in a NGIE, these systems are discussed in relation to their strengths and shortcomings. As a result, this section provides a review of relevant concepts and existing systems, and identifies contributions and the opportunity for the development of a new framework.

2.1 Client- and Resource-Aware Adaptation

Due to the vast heterogeneity of devices in the next generation Internet environment, there arises a need for applications to be aware of client device capabilities and environmental factors that may affect the applications execution, in particular what resources are available to the application on each device. Resources in a client server system can take many different forms. A resource could be application code or data components needed to perform a function, or hardware resources such as network bandwidth or device memory. For the purposes of this thesis, resources are defined as any aspect of the system able to be monitored for changes that may affect the operation of applications. This includes, but is not limited to, resources such as current CPU usage, available memory, network reliability and throughput, files and data, hardware resources such as printers, or device screen size and available battery life in the case of a mobile device. Awareness of these resources allows applications to actively adapt themselves to their changing execution environment, resulting in more efficient usage of all available resources and a more responsive service to the user.

The awareness of device capabilities is important for effective decisions regarding the adaptation of the application. Capabilities of interest include user interface characteristics such as screen size and display capabilities, available program memory and temporary storage, and the operating platform of the device. Drafted standards such as the RDF based Composite Capabilities / Preferences Profile (CC/PP) [But02] allow the specification of device capabilities and user preferences for the purpose of content adaptation. The use of such standards allow more informed adaptation decisions to be made, allowing more specific adaptation behaviour to be defined.

2.2 Application Adaptation

In a device- and resource-aware adaptive system, the distinction between thin and thick client applications is blurred, as applications may move components between the client and server, changing the configuration depending on available resources. This allows the application to dynamically modify it's design to suit the changing execution environment. The responsibility for this adaptation can lie in several places, depending on the nature of the application and the type of adaptation required.
The following section describes three strategies for adaptation of client server applications.

Laissez-faire Adaptation [JHE99] - Each application handles it's own adaptation. This allows applications to specify the best way to adapt to the current environment, but requires each application to implement its own adaptation mechanisms and resource monitoring.

Application Transparent Adaptation [JHE99] - Adaptation is performed by a supporting system, and is transparent to applications in the system. This is useful for legacy applications that can not be re-designed to incorporate support for adaptation, but operating system support is limited in the way it can facilitate adaptation.

Application Aware Adaptation [SNKP95] - Applications co-ordinate their own adaptation, but system support is offered to aid in the actual adaptation. The system provides support via services such as code migration and resource monitoring. The application decides how and when to adapt, and makes use of the services offered by the system to perform it's adaptation.

Application transparent adaptation is useful for providing adaptation in legacy applications, where the existing client application cannot be modified to support application aware adaptation. The laissez-faire and application aware adaptation techniques require that the application be developed with consideration for how the application will adapt. In any case, decisions regarding how the application is to adapt must be made with knowledge of the capabilities of devices, and the current state of the execution environment. The MobJeX framework described in section 2.4 of this thesis facilitates application aware adaptation, with increased system support for adaptation offering a high amount of transparency to application developers.

A common type of application transparent adaptation in use on the World Wide Web is dynamic content transformation, which allows a single data source to be presented in a way suitable for the client device. One way this is achieved is via the transformation of eXstensible Markup Language (XML) [BPSMM00] data sources to device specific markup using XSLT style sheets. This allows the customisation of the presentation method on a per device basis [KYME03]. Application transparent methods are suitable for use with traditional World Wide Web content, but do not provide adequate support for advanced user interfaces and delegation of transformation processing to client devices. The delegation of XML transformation to client devices can also improve server response times, although may increase bandwidth usage for the client [RP03a].

2.2.1 Adaptation using Mobile Code

Mobile objects offer another method of application adaptation by moving components of the application between systems in response to resource usage and availability on the devices. It has been shown that the movement of processing between server and client devices can significantly reduce the load on the server [RP03a], and may also offer a more responsive application to the user by moving commonly used functionality to the client device [RP03b]. This allows redundant resources to be used effectively by distributing the load of an application over client and server devices, achieving more efficient operation. Methods of device specific markup and content transformation do not allow for rich application components to be placed on the client device, as the adaptation they support only concerns the presentation layer of the application.

The idea of distributing code to multiple machines to best utilise all available resources is not new. Process Migration [MDP+00] has been used in distributed processing environments to perform load balancing of the systems. This involves moving an executing process between machines, and requires the movement of all process state (program counter, stack, and local variables) and program code from one machine to another.

Code mobility techniques have also been used with many *mobile agent* systems. An agent can be described as "active, autonomous, goal-driven, and typically acting on behalf of a user or another agent" [MDW99, p.451]. This implies that agents are automatic entities, making their own decisions to perform tasks intelligently. Mobile agents are useful for location dependent processing, but do not lend themselves to a distributed application environment, where constant interaction between components is required.

Both the laissez-faire and application-aware adaptation strategies described in section 2.2 could take advantage of code mobility in their adaptation of applications. In a situation when it is required that adaptation be totally transparent to the application, it is more difficult to migrate components of the application. This technique is more suited to legacy applications or applications running on systems not supporting code mobility. Ideally, a collaborative approach using code mobility would allow an object to dynamically migrate between devices to provide efficient operation. This would be semi-transparent to the application, as object migration and adaptation decisions would be the responsibility of the supporting system. A framework supporting mobile objects would ideally provide support for applications to adapt themselves based on rules defined per application.

2.3 Techniques for Object Mobility

As the aim of this section is to investigate adaptation in a next generation Internet environment, it is useful to first provide an understanding of the techniques required to facilitate effective object mobility within a heterogenous environment. Existing systems implementing these techniques are analysed in order to highlight the limitations of existing work, and techniques of use within a next generation Internet environment are identified.

2.3.1 Reference Management

An issue highly relevant to object mobility within adaptive systems is the management of references held by the migrating object, and the references to that object from others in the system. This problem arises in an object-oriented system where objects (containing code, data, references to other objects, and possibly execution state) are moved between computational environments[1]. To ensure that an application will function correctly, links between objects must be re-established, either when moving the object, or when the next attempt to access the object is made. Although several solutions have been developed, the object reference management remains challenging within a distributed application environment due to the potential complexity of application design. In order to provide an understanding of the problem, several techniques for maintaining object references in mobile object systems are explained below.

Acharya et al. [ARS97] describe *lazy tracking* of references, as used in the Sumatra framework. Lazy tracking is where the knowledge of the new location of a migrated object is held only at the location the object is moved from. When an unsuccessful attempt is made to access an object at a location it has moved from, the new location of the object is returned. If an object has moved from the new location, it's next known location is returned upon attempting to access the object. This means the

[1] physical or virtual 'places' in which objects can reside and execute.

path of the object's movement is manually followed by the entity looking for the object.
Sumatra uses *proxy references* to implement the lazy tracking of objects. When an object moves, all references to the moving object are converted to proxy references pointing to the objects new location. No other CE is informed of the object migration. When an attempt to access the object is made, an exception containing the next known location of the object is thrown back to the calling object.
This reduces the cost of moving an object, as no reference management needs to take place until the object is next accessed, and the application can abandon the search for the object if necessary [CBNH98]. This method places the responsibility of error control and tracking timeouts on the application developers, when ideally this should be handled by the supporting framework [SNKP95]. This is not suitable where application transparency is desired, unless the supporting framework can catch exceptions thrown by failed attempts to access remote objects before they reach the application, and deal with these accordingly.

Holder et al. [HBSG99] describe a similar technique for managing references as used in FarGo , where a single 'tracker' for each object is present on each device to assist tracking of objects. Each device contains a FarGo 'core' which is treated as a location in which objects can reside, and is responsible for controlling all object creation, migration and reference management. Whenever an object moves between cores, the originating cores tracker is notified of the new location. All attempts to access an object are through it's local tracker. If this tracker holds 'old' information as to where the remote object is, it will forward any method invocations to a tracker at the next known location of the object. This causes a chain of trackers similar to that in lazy tracking, but is transparent to the calling object, and performs optimisation of all trackers links to an object once a method invocation has been fulfilled. The resolution and optimisation of this chain is handled by the supporting system, and therefore transparent to the application. A variation on the lazy reference update technique has also been implemented in the Mobile Object Layer (MOL) [CBNH98], which uses asynchronous messaging between components. If a component moves from one CE to another, the originating CE will transparently forward any messages it receives for the component on to its new location. This technique allows the cost of moving an object to be minimised, as all references do not have to be updated until the object is next accessed. All accesses to object must be performed through a system supported entity (in FarGo, the 'tracker'), as any attempts to access the existing local object reference will result in the object not being found.
As this method relies on the passing of method calls through several nodes when chaining to a mobile object, the path to an object may be lost if a node was to fail. Even if this node was to re-activate, information as to the location of the object may still be lost, thus permanently invalidating the reference to the required object. As this method of reference management delays object reference updates until an attempt to access the object is made, it is highly suited to systems where objects are not accessed frequently, but are quite mobile and migrate several times before they are accessed. One could argue that if an object is not accessed once moved, then there is little point in moving the object to achieve better application performance, thus re-enforcing the need for active reference management. In the case that a large group of frequently accessed objects are migrated, it would not be possible to reduce the overheads of updating proxies by combining all remote lookups into a single step.

A pro-active type of reference management is *active proxying*, where all object accesses are coordinated through a reference proxy that always points to the current location of the object. When an object moved, all proxies pointing to that object are updated immediately. Such a method may employ a single proxy for all devices, or synchronised local proxies for each device to allow faster access

to object locations. One advantage this technique has over other techniques is that is is not vulnerable to device failure in the same manner as chaining and lazy tracking are. When using chaining or lazy tracking, if a device making up part of a chain or sequence of previous object locations was to lose network connectivity, the path to the objects would be lost. As direct references to the current location of an object is held, the loss of a node would only affect references to objects held on that particular node.
Proxying addresses this by maintaining an up to date record of object locations on more than one machine. Unfortunately the migration overheads of such a method are higher than the variations of lazy tracking as each object migration will force all proxies to update. The cost of an object update is constant, and is always as high as the potential (and delayed) cost of reference updates in the chaining method described above. Proxy updates can also be grouped, to avoid overhead of updating each proxy individually with the new object location. Additionally, all accesses to an object must be performed via the proxy, but this is the case with any reference management technique that does not monitor all object method calls at a system level.
RAM [BSLS01] is a mobile object system implementing active proxy updates via a *reference table* held for each group of objects in the system. This removes the need to search deep within object trees for all proxies that need updating, as all the reference table holds all information about proxies within a particular object. A major limitation is the requirement of the table to be updated every time an object is created or destroyed, requiring intervention by the supporting system to retrieve information about objects.
The active proxying method is suited to systems with lower mobility and a smaller number of nodes, but with frequent accesses to service objects. This would be the case in a NGIE, where client server applications may consist of no more than a client device and several servers participating in application adaptation, but fast and efficient application operation is desired.

2.3.2 Object Grouping

To maintain efficiency in mobile object systems, objects communicating frequently should remain as close as possible in a network environment, preferably in the same CE. Ideally, each mobile object would be an atomic entity, with loose coupling to other objects in the system, but in reality this is rarely the case. It is for this reason that *co-locality* constraints must be defined, ensuring that objects with tight coupling are always in the same location. This ensures that objects are distributed in a way that most efficiently uses bandwidth. This may be extended to include objects that are tied to physical resources that cannot be moved. For example if an object frequently reads and writes to a file, it could not be moved away from that file without breaking the links to the open resource. At times, migration of an object may not be as efficient as making a copy of the object for use on the remote device. If a resource is *cloneable*, efficiency may be improved by having a local copy of the object in any device that requires it. This also presents the possibility of re-binding or replacing objects in a group with existing objects present on the device, or device specific object implementations. A set of these constraints can be defined as *object grouping*.

FarGo [HBSG99] achieves object grouping via *complets* which are components consisting of many objects encapsulated in a single Java class. Complets model the *facade* design pattern [GHJV94], where all objects are accessed via a single interface. A complet is the smallest entity that can be migrated between CEs in the FarGo system, meaning that all mobile objects must either extend the Complet class or be part of an existing complet. All complet structures must be specified at compile

time, limiting the flexibility of grouping at runtime. Co-locality between complets is facilitated by *reloactors* allowing one complet to link to, copy, or migrate another with it when it moves. Only one relocator is allowed per complet, limiting the flexibility of complet grouping.

The Sumatra framework [ARS97] allows grouping of objects via explicitly named object groups within the application. An object group is the smallest entity that can be migrated between systems in Sumatra, requiring that any mobile object be a member of a group. Dynamic composition of groups is possible by adding and removing objects from groups at runtime, but this must be handled programmatically by the application.

Since object characteristics differ between applications, grouping rules must be defined on a per application basis. Additionally, optimal object grouping strategies may differ depending on the scenario the system is faced with, especially when dealing with a wide range of devices with varying capabilities.
Therefore to allow for the variability of resources and client capabilities in a NGIE, there is a need to dynamically choose grouping strategies to best suit client devices and their capabilities. As none of the reviewed systems support dynamic grouping strategies, this provides rationale for the support of dynamic grouping by the system implemented in this thesis.

2.3.3 Object Substitution

Due to the heterogeneity of clients in a NGIE, simply moving existing objects between server and client may not achieve maximum efficiency, as there is a possibility that an object may need optimisation for different types of clients. In order to facilitate the modification of algorithms and object behaviour in response to device capabilities, it is desirable to be able to substitute objects of a given type with different implementations providing the same service. This allows more flexible adaptation than simply moving the same implementation of an object between systems with different capabilities (for example, moving an object running on Java 2 Standard Edition to a device running Java 2 Micro Edition). This allows the developer to optimise objects and algorithms to suit the type of device the object is running on. All objects of the same type must share a common interface to allow the substitution of implementations while still offering the same functions.

An extension to object substitution is the binding to existing objects providing the same service as objects currently in use. This allows binding to shared services based on the location of the device, to facilitate locality based information services, or to optimise network usage [BCM03]. This is particularly useful when used with mobile devices, as devices may roam between many localities during the lifetime of an application.

Techniques allowing dynamic binding to services are used in HADAS [BSCHL97], a framework for dynamic composition of service components. HADAS allows for services to be constructed from components when the service is started, spreading the components across all available devices depending on resource availability. No locality based binding is supported, and no services are modified after the initial layout of components, limiting it's usefulness for ongoing resource aware adaptation.

Columba [BCM03], a framework specifically targeted at location aware mobile devices, allows service objects to be described using *resource descriptors*, and focuses on the re-binding of objects based on device locality and the type of service required. Services are described by an implementation independent resource descriptor for each service. This removes the need for a common interface to

objects, but also increases the complexity of rebinding to services of different types, as the coupling between objects using the services and the interface to the services must be low to accommodate the changes in service interfaces. Additionally, Columba does not support service object mobility. Ideally service components should migrate between client and server devices to allow efficient usage of client resources.

The substitution of objects has implications when maintaining object state, as the existing objects data state (and possibly execution state) must be placed into the new instance of the object. This may cause problems as even though two objects may share the same interface, their internal implementations may vary greatly. One solution to ease the swapping of objects would be to define a *fixed part* and a *dynamic part* of the object as used in HADAS [BSCHL97]. The fixed part would represent global data members of the object, and the defined interface for the object. The dynamic part would consist of method implementations varying per device. This would allow much of the object data state to be transferred to the new implementation, but would still allow the implementation of methods to differ between objects. Without this specific object substitution support, similar functionality could be achieved via dynamic object grouping structured in such a way that a fixed object set and common interface is defined, whilst allowing any other service objects within the group free to change dynamically.

2.4 MobJeX

Because the object migration and reference management framework implemented and tested in this thesis forms part of MobJeX [RP03c], a framework currently under development, the following section provides an overview of the MobJeX architecture. MobJeX is a Java based framework with the goal of providing transparent support for dynamic adaptation of applications in a heterogeneous environment. MobJeX aims to achieve this by supporting migration of mobile objects at a configurable level of granularity, and applying a clear separation between user code and adaptation policies.
This is achieved by using an XML based deployment descriptor containing rules regarding the grouping and movement of objects in different resource scenarios. Coupled with strong mobility implemented using the standard Sun Java Virtual Machine (JVM), and transparent object reference management, supporting both object substitution and locality binding, MobJeX aims to support the development of rich adaptive Internet applications, with minimal effort from application developers.

The main goals of the MobJeX framework are to provide [RP03b]:

Platform Independence - allowing a wide range of devices to use adaptive applications.

Dynamic Adaptation - highly transparent to the applications, allowing existing applications to be easily modified to use the framework, and simplifying development of new applications supported by the framework.

Management of Object References and Grouping - adding to the transparency of adaptation and object mobility to applications.

Declarative Rule Based Adaptation and Object Grouping Configuration - defining a clear separation between application logic and dynamic adaptation, and maintaining a high degree of flexibility, using XML.

Dynamic Object Mobility - Object mobility is dictated by policies defined on application deployment.

Pluggable Resource Management - allowing resource managers for a wide range of architectures to be used in the framework.

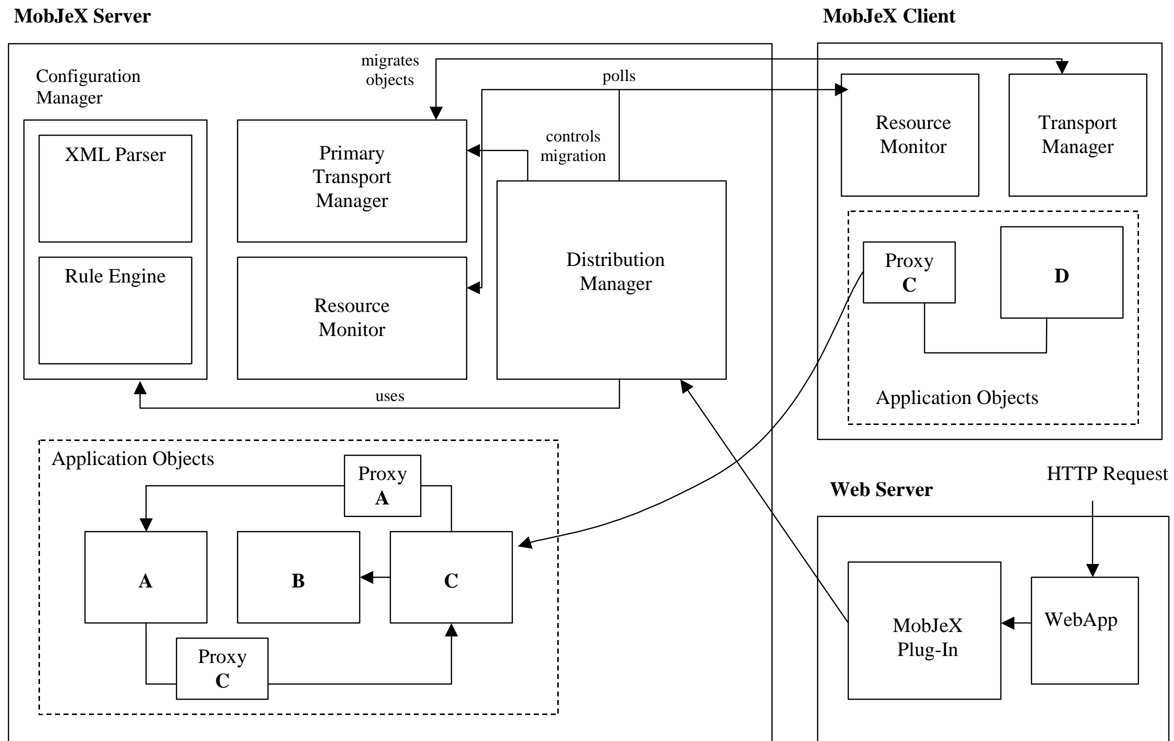

Figure 1: MobJeX Architecture (modified from [RP03c])

2.4.1 MobJeX Architecture Overview

MobJeX consists of a number of logical components, each with distinct responsibilities. The following sections give a brief overview of the MobJeX framework, and provide an explanation the components in Figure 1. An area of focus in the MobJeX framework is the clear separation between the management of mobility and the co-ordination of adaptation. This *separation of concerns* [TP01] allows developers to focus on the functional requirements of an application, and delay decisions about application layout and adaptation until deployment time.

2.4.2 Distribution Manager

The MobJeX Distribution Manager (MDM) is responsible for making adaptation decisions based on the knowledge provided by the MobJeX resource monitor (section 2.4.4) on both the client and server. These decisions are based on the an XML deployment descriptor defining rules regarding the adaptation, object movement and grouping, and resource usage scenarios for the application. The MDM

maintains communication with both devices that the application is running on, to monitor resources and issue object movement and adaptation commands to the remote Transport Manager (section 2.4.3. The MDM may also issue object substitution commands when migrating an object to a device requiring a different implementation of a certain object. The MDM is also responsible for enforcing security and physical resource dependencies of all mobile objects, within the set of constraints imposed by the framework (described in section 4).
As the MDM holds all information about the mobility characteristics of a given object, the Transport Manager may request information regarding an object's mobility or other characteristics (such as co-location rules when moving and object). Only one MDM is required in the MobJeX server, as all movement operations on the client are initiated from the server. Information regarding resource usage is collected from both client and server devices allowing the MDM to react to changes in the environment. Profiling of application objects will also be performed by the MDM with assistance from the resource monitor, allowing more accurate decisions to be made regarding the mobility of an object. For example, if an object becomes too large to fit in the client devices memory, the MDM would be notified and the object could be migrated to the server.
The exact method of specifying and evaluating XML based rules governing the adaptation of applications is a topic of further research.

2.4.3 Transport Manager

The MobJeX Transport Manager (MTM) conducts all object movement and reference management tasks within the system, and is the main focus of this thesis. A MTM is required on both the client and the server to assist in movement of object between both devices, and to ensure that object referencing integrity is maintained across all devices. Mobile objects in the MobJeX system require several additional components (as described in section 4) to allow them to move around the system whilst maintaining their normal operation. The MTM is responsible for maintaining the integrity of all mobile object references, and coordinating the migration of objects on request from the MDM. The main goal of this thesis is to develop, implement and test the MTM component, as described further in section 4.

2.4.4 Resource Monitor

MobJeX supports resource monitoring as a separate plug-in using a standard Java API (currently under development). This allows platform specific resource monitoring services to be developed, and allows the use of third party resource monitoring suites. This architecture also allows the same resource monitor to be used by many MobJeX applications on a single device. For a device to be compatible with the MobJeX framework, they must at minimum support a resource monitor that can report the available working memory on the device.
A Windows 2000 resource monitor compatible with the MobJeX Resource Monitoring API [Fri03] has been developed using the Java Native Interface (JNI) [JNI97], with view to extend it to support the Java Virtual Machine Profiler Interface (JVMPI) [JVM99] for monitoring resources in use by the JVM.

2.5 Summary

In summary, this section has presented several techniques for adaptation making use of object mobility, and analysed existing work in relation to each technique. Many of the existing systems discussed lack the required support for applications to effectively adapt to a heterogenous NGIE. Where FarGo has strong support for object referencing and mobility of object to clients, it lacks the flexibility required to effectively adapt applications for devices in a NGIE. Object grouping and co-locality is crucial to effective adaptation using mobile objects, and should be specified and enforced at a system level. Columba supports dynamic binding to middleware objects using implementation independent resource descriptors, but lacks the support for utilising client resources on a per application basis.
The MobJeX Transport Manager component, the development of which is the focus of this thesis, attempts to address the shortcomings of existing work by supporting dynamic adaptation techniques suitable for use in a NGIE. The remainder of this thesis describes the contributions, implementation and testing of the MTM component, which combines existing techniques to provide flexible and transparent support for the transportation of objects in the MobJeX framework.

3 Project Goals and Contributions

The main contribution of this project is the development of the Transport Manager component of the MobJeX framework, implementing object mobility and remote object reference management. This is done based on an in depth review, and the development of novel techniques for use in a next generation Internet environment. Further to this goal, several sub-goals for the component have been defined in line with the goals of the MobJeX framework described in section 2.4:

Goal 1. *Developer and User transparent object mobility.*
Wherever possible, neither application developers nor users need make decisions about the mobility and adaptation of an application, making the adaptation of an application as transparent as possible.

Goal 2. *Object location transparency.*
For applications to function with no knowledge of mobility and adaptation, the physical location of objects and resources should be hidden from the application. This allows the application to use all resources as if they were local, leaving the MDM to distribute the objects according to adaptation rules.

Goal 3. *Controllable granularity of mobility.*
To facilitate flexible adaptation polices, the granularity of object mobility should be configurable, allowing objects to move between devices individually, or as grouped components.

Goal 4. *Support for Device Heterogeneity.*
Due to the diversity of the NGIE, a heterogenous set of devices must be supported to assist the wide-spread distribution of an application.

Goal 5. *Flexible object bindings, allowing the use of device or location specific application objects.*
Object implementations should be interchangeable to support the heterogeneity of devices, as

should bindings to localised service components, thereby further facilitating the flexibility of adaptation in a mobile device environment.

Goal 6. *Efficiency in comparison with existing systems.*
The implementation of techniques used in the system should be at least as efficient as existing implementations.

In order to address these goals, this thesis develops a framework supporting adaptive applications to provide answers to the following specific research questions:

1. *Can all changes required to application code be done via preprocessing, thus providing transparency to the developer?*
2. *Can all mobile objects be accessed as if they were local, regardless of their actual location within the system?*
3. *Can objects be moved individually or as groups whilst maintaining all references to objects in the system?*
4. *To what extent does the framework support heterogeneity?*
5. *What pattern of design best allows the dynamic substitution of object implementations and the transparent rebinding of object instances?*
6. *How does the efficiency of the techniques used in this framework compare to existing work?*

4 Methodology

The following sections are primarily concerned with the MobJeX Transport Manager (MTM), as described in section 2.4.3 of this thesis. Each aspect of the implementation of the MTM and it's components are described in detail, with emphasis on addressing the goals described in section 3.

4.1 Overview

It is proposed that a combination between transparent active proxying and transparent lazy tracking (see section 2.3.1) systems for reference management be used in the MTM, allowing a choice between each type of management depending on the situation with which the MTM is faced (see section 4.2). This is well suited to a client-server environment as no complex reference chaining or global lookups are required.

To address the goal of heterogeneity of devices (section 3, goal 4), Java [Kra96] has been chosen for this implementation, as it's cross platform capabilities allow a single codebase to be used on many devices. Java virtual machines are available for many mobile devices, including Java 2 Micro Edition (J2ME) [J2M00] which is becoming increasingly popular on mobile phones. Java's Remote Method Invocation (RMI) [Jav03] mechanisms are used for communication between system components and application objects in this implementation of the MTM. This allows interaction with a remote object as if it were on the local device, and is supported by the all latest variants of Java (although is optional in J2ME).

4.2 Reference Management

One of the major responsibilities of the MTM in the MobJeX framework is the management of references between application objects. As objects move between devices in the system, their references to other objects within the system must be maintained to ensure that the application continues to function correctly. For this implementation of the MTM, we have chosen to use the techniques of *active proxying* (section 2.3.1) and transparent lazy updates to allow flexible reference management on object migration (section 4.2.2).

The MTM proxy class hierarchy (see Figure 2) was designed to address the goals stated in section 3, and allows implementations of objects to vary independently of the application code referring to the objects. This hierarchy also allows object proxies to seamlessly take the place of local objects to facilitate the migration of objects. A single interface is defined for each application object, and implemented by both the objects implementation and proxy, allowing proxies and application objects to be interchanged freely in application code. The use of the *bridge* design pattern[GHJV94] separates the implementation of an object from it's interface, allowing the seamless interchanging of different object implementations and instances, addressing goal number 5 of the MTM. The use of interfaces to enforce object typing leaves the object free to extend existing application classes. This differs from the design of FarGo (see section 2) which fixes object class types within the 'tracker' (proxy object) that references them. This prevents the rebinding of trackers to object types other than the type specified at compilation time. FarGo complet classes must also extend the *complet* class, preventing the developer from specifying any other type of inheritance.
All common proxy functionality is held in an abstract class extended by the proxy classes, reducing the amount of code repetition in each application object proxy.

The characteristics of RMI allow the referencing of a remote object as if it were local, with a 'stub' taking the place of the object as a local reference. This allows proxies to hold a reference to a stub or local object, depending on the location of the object.

The requirement of dynamic grouping means that all objects must be able to maintain dynamic references, allowing the re-grouping of components without losing the integrity of references. This problem does not arise in fixed object group systems where object instances will never be separated, as all inter-group references can be local, thus reinforcing the need for effective reference management.

Due to the complexity of reference management in a mobile object system, several key rules have been devised as to the referencing of objects in the transport and reference management scheme of the MTM.

1. *All mobile objects must be referred to via a proxy.*
   All distinct object instances that may be mobile in the lifetime of the system must have it's own proxy, and be referred to via this proxy.

2. *All objects referenced from a mobile object must be referenced via a proxy.*
   As there is no guarantee of co-locality between objects (as this is specified on deployment), the migrating object must be able to maintain it's references once moved. If all external references are proxied, this allows the references to be updated to point to the originating location when an object is migrated.

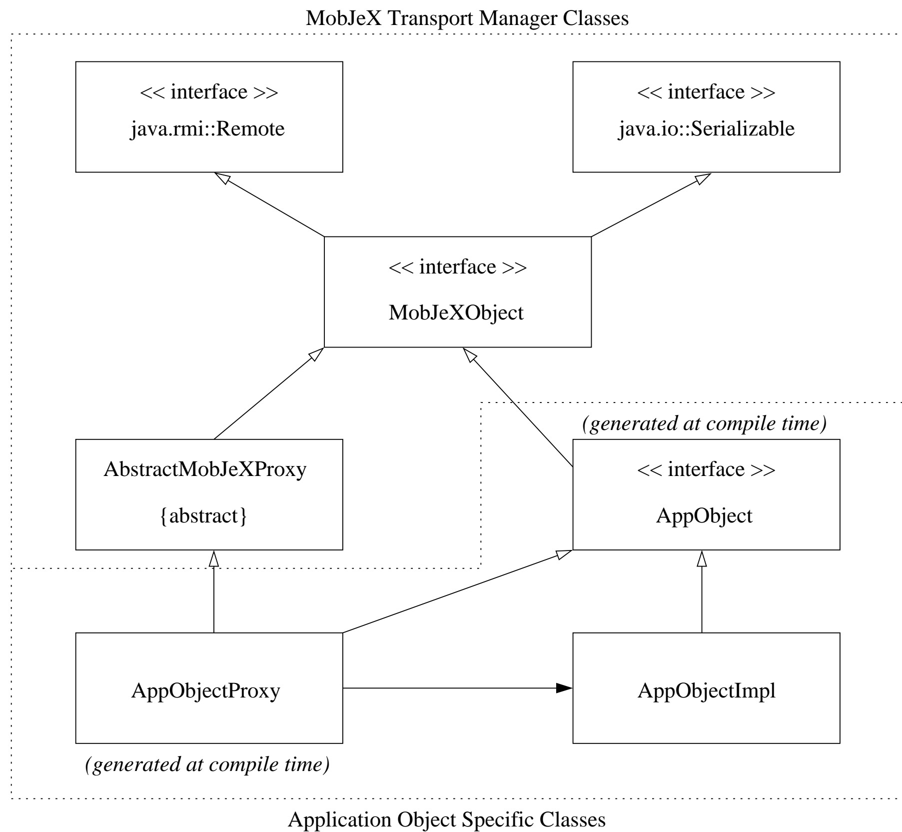

Figure 2: MTM Application Object Class Hierachy

To facilitate the enforcement of these rules, all instantiation of objects that may require proxies are done via a creation service provided by the MTM (see section 4.4). Additionally, if an object moves to a system where there is no instance of a proxy, a proxy may need to be created to allow local objects to reference the mobile object.

4.2.1 Object Creation

To satisfy the above constraints, the creation of objects must be controlled to allow the substitution of a newly created object with a reference to a local proxy where necessary. The decision as to whether to reference an object by proxy is ultimately made by the MDM component, by advising the MTM that an object may possibly migrate in the lifetime of the application. All decisions as to the initial location (local or remote) and implementation of a given object are also made by the MDM prior to object creation. By creating all objects via a creation method in the MTM (see section 4.4), the framework gains control over the newly created object, and can return either a proxy or a normal object reference to the application code. This is not possible with FarGo, as all mobile objects are referenced by their tracker. As object mobility characteristics are specified at application deployment time, the mobility of an object are not known until runtime. The creation of proxies only when needed facilitates efficient dynamic mobility.
A unique ID is also created for the object, to identify it within the MobJeX system. Figures 3 and 4 show the process of creating an object and the resulting reference tree in each situation. If the

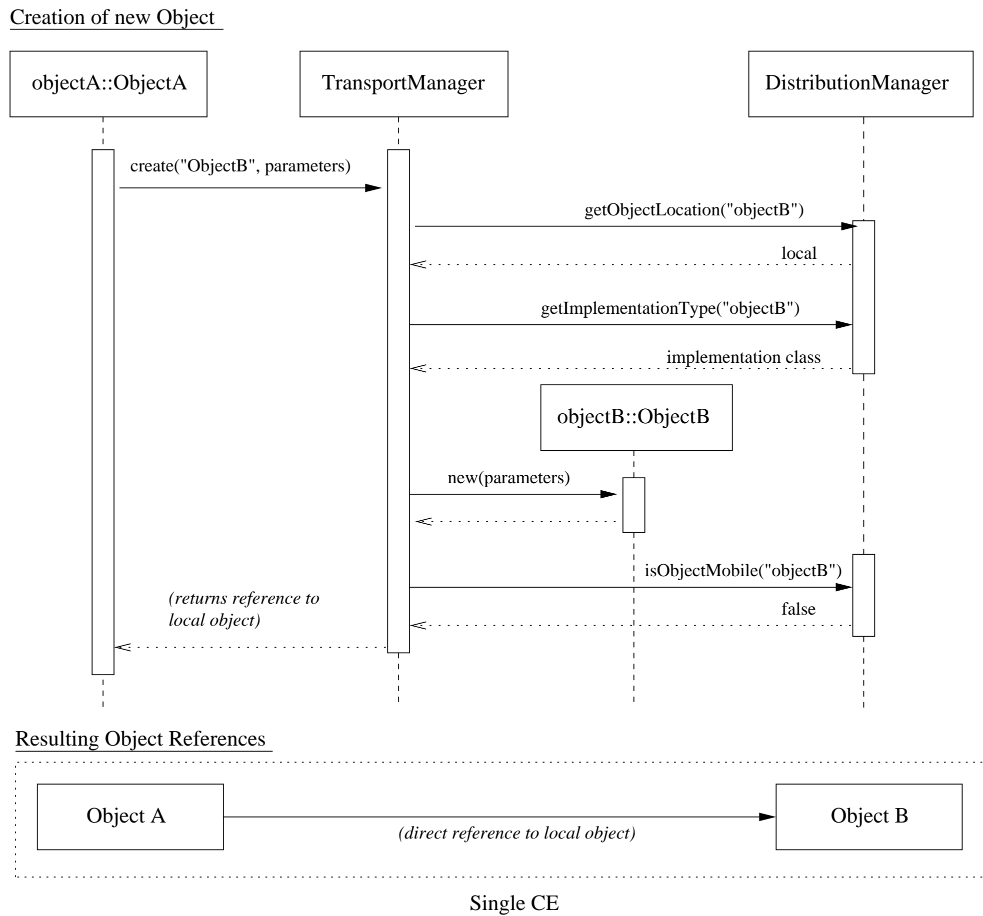

Figure 3: Object Creation for a Non-Mobile Object

object will never be mobile, or is not being referenced from a mobile object, no proxy will be created, and a reference directly to the local object will be returned. If a proxy is required, it is created by name using Java Reflection. Overheads are reduced by linking the proxy directly back to the local object, instead of an RMI stub as in the case of a remote object. A naive implementation may have referenced any object by a stub regardless of it's location, but this would create unnecessary overheads when accessing local objects via a proxy. Only one proxy for each application object on each device is ever created, although copies of proxies may be passed in migrating objects and as parameters of remote method calls.

In order to address goal 5 of the MTM by assisting in the separation of object types and their implementations, objects are created using the object's interface name, utilizing Java Reflection [Ref03] to perform dynamic object creation. This allows the MTM to consult the MDM for the actual implementation to create, but would require standardisation of the objects constructors.

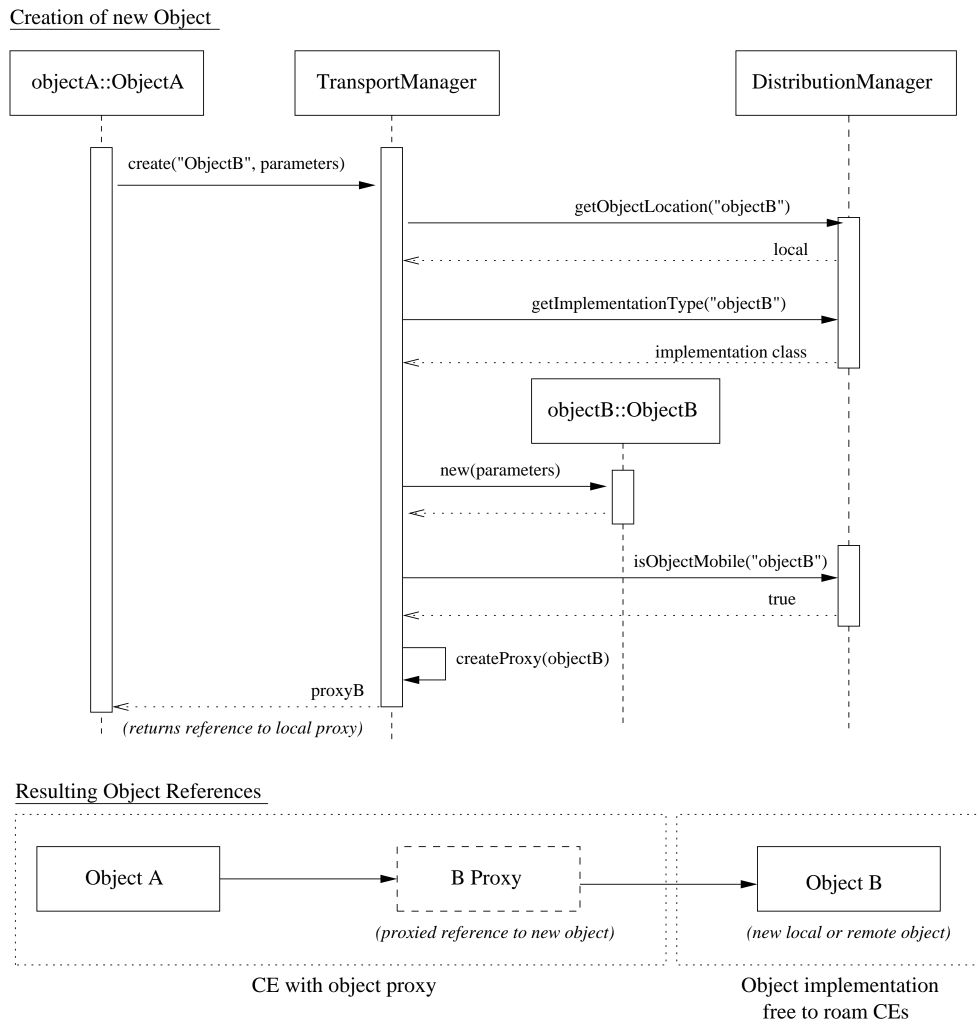

Figure 4: Object Creation for a Mobile Object

4.2.2 Object Migration

Figure 5 shows an example of an object migration requiring proxies on both devices to be updated. When an object migrates between devices, the proxies for that object on both devices must be updated to point to the new location. Additionally, any proxies that the migrating object holds references to will be copied and transported to the new location. As proxies contain only the ID of the object they represent and a single object reference, their serialized representation is quite small. After migration these proxies must also be updated to point to the current implementation of the object they represent. If a proxy for an object of that type already exists on the device, it is reused and the migrated proxy discarded.

As proxies may be held by any object held by the migrating object, a recursive search for proxies would need to be performed to find all the proxies within that object's reference tree. This poses a possible performance issue, as a deep search of an object for proxies could take up considerable processing resources. It is for this reason that a 'lazy update' has been implemented for all proxies

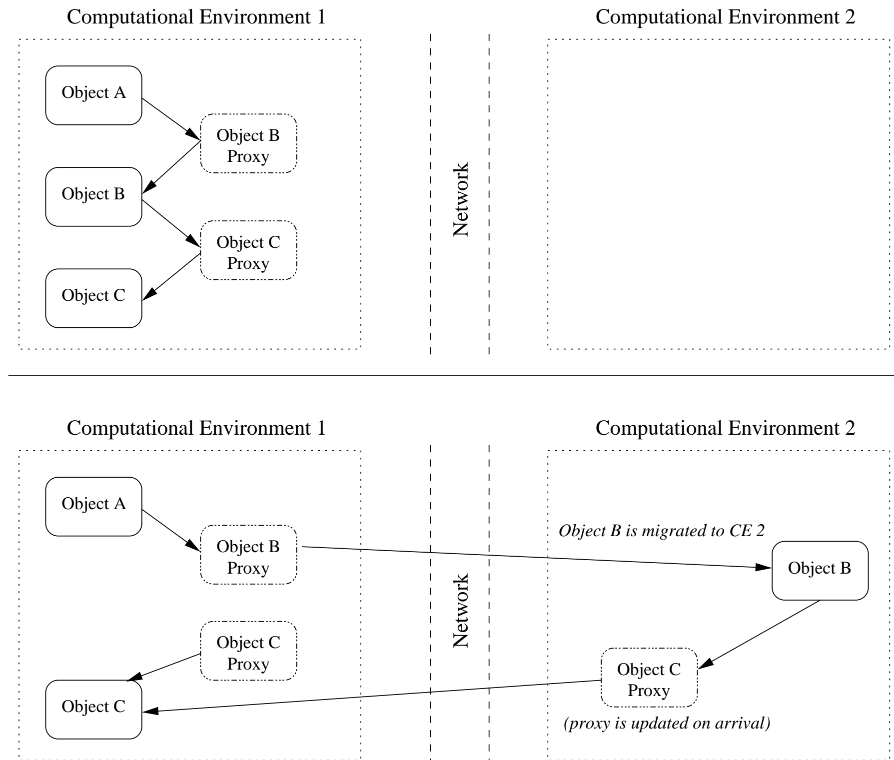

Figure 5: Object Reference Management when moving Object B

except those directly linked to the migrating objects. As references within proxy objects are held as *transient* (see section 4.3), on migration these references will be invalidated as part of the serialization process. Therefore, to update the proxy when a method is invoked, a simple check is performed to ensure that the proxy link is valid. If it is found to be invalid, a request is made to the local MTM to update the proxy with a local reference, or an RMI stub representing a remote object. This reduces the processing required on object migration, but may be less efficient if all proxies contained within a recently migrated object are invoked soon after migration. Additionally, it is not possible to maintain a single proxy per device with lazy updates, as proxies that are migrated with the object will be re-used when updated. To force the use of an existing proxy, the parent object referencing the proxy must be known and its reference retrieved. As there is no mechanism in Java to facilitate the retrieval of referencing objects, this information is not available unless the object reference tree is traversed until the parent is found. In this situation, it may be more efficient to perform a full update of all proxies on migration, allowing a reduction in the number of proxies on the devices for each object, and allowing all RMI stubs to be retrieved from remote devices in one step. This method of update has been implemented in the MTM in addition to transparent lazy updates, to allow flexibility in the way the MTM operates depending on application usage. A recursive search for proxies would allow the compilation of a list of proxies for which there is no valid proxy already on the device. This list is then sent to the remote MTM which responds with a collection of RMI stubs for the specified objects. Existing proxies can be reused by substituting references to copied proxies with references to existing proxies. These two steps are separated to allow them to be delegated to different devices if necessary. This allows the delegation of recursive proxy searching and remote stub creation to the more powerful device, relieving less capable devices of the need to perform this processing. Additionally, for proxy

searching to take place, the device must support Java Reflection. The method of proxy searching also relies on access to retrieve and change the referenced objects, which may cause security violations if a Java Security Manager is present and denies access to these objects. In a heterogenous client environment, there is no guarantee that the client contains the required Java APIs for Reflection, requiring that all reflective functionality be performed on the server. (see section 4.6).
Object copy operations are supported to allow cloneable data resources to be migrated whilst leaving a local copy behind. Situations where copying would be appropriate would be specified in application adaptation and object grouping rules as handled by the MDM.

4.2.3 Object Substitution

In order to achieve the goal of allowing flexible object bindings (section 3, goal 5), the MTM is designed to support object rebinding and implementation substitution through the use of a common implementation independent interface for each object. The use of the *bridge* design pattern [GHJV94] (see section 4.2) allows many different implementations of the one application object to be interchanged at runtime. To achieve object substitution, the MTM need only to manage the creation of a new object (or retrieval of an existing object) and the updating of proxies on both devices. Creation of objects by their interface name allows dynamic creation of the most appropriate object for the device on which it will be created. The MDM would also need to support object substitution, as it is responsible for all decisions regarding which object implementations are used in different resource and device scenarios. A major limitation of object substitution in the MTM is the lack of support for transferring object data and execution state when substituting an existing object with a new implementation of the same interface type. This would involve copying and conversion of data types between the objects, and would require a standard of object design which allows interchangeable object data and state. One possible method would be similar to HADAS'[BSCHL97] fixed/dynamic service composition, but at an object level. The fixed part would be transferrable between object implementations, whilst the dynamic part would differ on a per object basis. This could be achieved in MobJeX by composing a group of objects where data state object implementation objects are static, but service object implementations are free to change on a per device basis.
Execution state migration when changing object implementations would be near impossible with current state migration techniques due to the different algorithms and method bodies implemented in each object.

4.2.4 Proxied Object Usage

The transparency of object proxying allows method calls to be performed on an object with no knowledge that the object is proxied, or even on a remote device, therefore addressing the goal of object location transparency (section 3, goal 2). Unfortunately, the use of proxied objects as anything but parameters of local method calls can become problematic, especially when trying to serialize an application object represented by a proxy, or when passing the proxy as a parameter in a remote method call. The usual behaviour of RMI is to convert 'remote' objects to stubs when passed as a parameter in a remote method call, or when returned from a remote method. This is advantageous when immobile un-proxied objects are passed as parameters, as these are converted to stubs. This essentially means all MobJeX application objects are passed by reference. Any object not of type 'MobJeXObject' will be passed as a copy of the object as in normal RMI operation. This has implications in the develop-

ment of the system, as the developer must be aware that parameters and return values may be passed by reference or copy depending on the object type, and the location of the object within the MobJeX system. It is not safe to assume that all parameters passed into a method call to a MobJeX object are being passed by copy, as if the object is local, proxied or un-proxied, non-primitive objects will be passed by reference. It is important to note that due to the granularity of object serialisation, when a non-mobile object is passed by copy, any mobile object that is reference via proxy will not be copied. Instead, these proxies are updated to reference the remote proxy, essentially forcing all mobile objects to be passed by reference.
FarGo (see section 2) behaves in a similar way, whereby complets are always passed by reference through remote method calls, and any non-complet object is passed by copy.
Proxies passed as parameters to remote method calls must be treated specially by the MTM, as they must be updated to maintain the correct references on the remote device. To avoid the need for a deep search of parameters for proxies to update, which would require all remote method calls to be made via the MTM, the lazy update technique is used for parameter passing.

4.2.5 Non-Moveable Objects

Certain objects are immobile by nature, usually because they are tied to physical resources that cannot be moved with the object. An example of this kind of object is a Java I/O 'FileInputStream', which is bound to a file in the device's storage. As this object may need to be accessed from a mobile application object, we must be able to generate a proxy for the object to allow remote objects to maintain a reference to the file from any device in the system. To avoid the recompilation of core Java classes, it is suggested that 'wrapper' objects be created for the immobile types in use by the application. This may be a voluntary action by the developer, or an automated task performed by the MobJeX Compiler (see section 4.4). Alternative solutions include the copying or moving of files between devices, but this may create problems with data synchronisation and partitioning. There are many research initiatives dealing with the management of distributed data and immovable resources in a mobile computing environment [MES95].

4.3 Serialization

In order to address the goal of controllable object migration granularity (see section 3, goal 3), which in turn facilitates the flexibility of object migration strategies, we must be able to control the depth of object serialization. The standard Java serialization mechanisms recursively serialize every object referenced by the object in question, provided it is not declared *transient*[2] within the class. Rather than serializing all objects that are referenced by a mobile application object in the MobJeX system, we want to be able to control the depth to which serialization occurs, allowing us to be more selective of the objects that are sent to the receiving device, so as to better fulfill the requests of the Distribution Manager.
The decision of how to implement this is influenced by constraints of object reference management (as described in section 4.2) whereby all objects referenced from a moveable object must be via a proxy. Thus to stop serialization of any proxied object, we declare the proxy's reference to the object as *transient*. As transient fields are not serialized in the standard Java serialization process, this allows us to take advantage of existing Java functionality as opposed to re-writing serialization code.

[2] fi elds declared as transient are not serialized.

As the framework and it's components mature, more sophisticated methods of determining what objects are automatically serialized with the mobile object will be implemented, including moving any object to which the migrating object holds the only reference. Additionally, a more structured integration with the MDM (see section 2.4.2) would allow serialization of all objects with co-locality constraints in a single step.
One important characteristic of Java RMI that influenced the serialization method used was the conversion of exported RMI objects to RMI stubs when they are passed as parameters in remote method calls. As all active application objects are exported to allow remote method invocation, they must be serialized in such a way that Java will not be able to recognise them as exported RMI objects and perform the conversion to stubs. As such, migrating objects are serialized to an array of bytes which is then passed to the remote MTM as the parameter in an RMI call.

4.4 Transparency

To address the goal of application developer transparency (section 3, goal 1) of the framework, the MobJeX compiler component performs preprocessing on application code, allowing standard Java code to be easily incorporated as an application in the framework. For an example of the modification required to existing application code, see appendix A.
The choice was made to generate proxies for all application objects in the system at compile time, regardless of whether that object could be mobile or referenced from a mobile object. This removes the need to consult the MDM for information regarding the mobility of an object, defined by the adaptation policies, which may change at deployment time. When using multiple object implementations for a single object, the signatures of public methods must be identical across all implementations, allowing the extraction of a common object interface.
An alternative to compile time generation would be to generate proxy classes and application object interfaces at runtime using Java Reflection. This would avoid the generation of unused proxy classes, but would add to the existing overhead involved in object creation.

The use of a common interface to identify objects (see Figure 2). was chosen over an abstract class defining public abstract methods, as this leaves application objects open to extend any existing class providing extra functionality. As Java classes can implement many interfaces, this solution requires no change to existing application objects in regards to the inheritance and implementation of existing classes and methods. As Java interfaces are implementation independent, this provides a clean separation between proxies and implementations of objects. As part of the MTM design, application objects and their proxies share a common interface, allowing either type of object to be returned from the MTM's creation method without any casting or explicit type changes required.

To allow communication with remote objects via RMI, stubs need to be generated and compiled for any object that can be accessed via proxy. These can be generated using Sun's `rmic` utility that is distributed with the Standard Java SDK [Kra96]. This function can be easily integrated with the MobJeX pre-processor to generate and compile stubs as the final step of proxy generation.

To facilitate the framework's control of the creation, distribution and interaction between objects, all creation of objects must be performed via the MTM. This necessity means that the application code must be modified to call an MTM method when creating new objects, looking up existing remote objects, and calling remote methods on certain types of objects. All application source modifications can be made via a Java source code pre-processor incorporated into the MobJeX compiler. There

exist several mature open-source Java pre-processors that could be incorporated into the MobJeX Compiler. One such pre-processor is the Extensible Java Preprocessor Kit (EPP)[IR97], which allows extensions to the Java language to be developed and implemented as a separate module for EPP. The implementation of a MobJeX EPP module and the integration of EPP into the MobJeX Compiler is a topic of further research.

The `create` method defined in the MTM consults the MDM in order to ascertain whether a proxy is needed for the newly created object, and returns either a reference to the existing implementation object, or a proxy created for the object. This allows runtime decisions to be made as to the mobility of an object, as a proxy is only created and referred to when the adaptation policies define that an object can be mobile. FarGo (see section 2) performs an objects creation via its tracker, allowing a creation call to the supporting system within the trackers constructor. As there is no guarantee that a MobJeX application object will be mobile, we cannot perform creation of objects via an object proxy.

As the type of reference held to an object within the application code is dynamic, and depends on the current adaptation policies, flexible references to mobile objects must be held. This allows the MTMs `create` method to return either a proxy for the implementation of a mobile object, or the local implementation if the object will never migrate between devices.

In remote method calls, any proxies held in objects sent as parameters of the method call are passed by copy (unless they are active *Remote* objects - see section 4.3). These proxies must be updated to refer to the valid location for the object being proxied before they can be used. As the lazy update technique is used for proxies passed as parameters, no modification to remote method calls is required.

As all remote mobile objects must be referenced via proxy, the MTM must take on the role of the RMI registry with regards to remote object lookup. A *lookup* similar to that performed on an RMI registry can be conducted via the MTM. As all mobile object references are held in the MTM, a proxy to the required object can be provided when a lookup is requested. This relies on the object being bound by a name known to the application, which can be done by calling a *bind* method on the MTM.

Additional changes to existing application code include the generation of proxy and 'wrapper' classes for Java utility objects representing non-movable resources (e.g. I/O objects, threads). This allows mobile objects to hold references to non-mobile objects without them needing a separate wrapper class. As there is no guarantee that the source code for these objects will be available on development devices, Reflection would be used to ascertain the methods needed for generation of the proxy class.

To assist in the transparency of mobility to developers, MobJeX provides its own compilation and preprocessing strategy involving the creation of wrapper proxy classes, modification of existing class source, and the compilation of the required RMI stubs. This allows the above changes to be made to existing Java applications. This also reduces the developers required knowledge of the framework and application adaptation.

4.5 Remote Class-loading

As Java separates instances of objects from the class files that represent them, class files may need to be provided to devices as needed. For the purposes of this MobJeX MTM implementation, all class files were made available to the virtual machines on both devices, avoiding the need for remote class loading. In a realistic scenario, this would not be possible, as there is no guarantee that all clients will have the required classes available on the device. Therefore, there is a need for the client

to dynamically load the required application classes from the sending server when and object first moves. One possibility is to make use of existing Java remote class loading and caching techniques. This could be implemented by setting up the MobJeX Web Server as a Java codebase server, and using the URL of the server with the Java `URLClassloader` class. This would allow classes to be loaded automatically via HTTP when needed, and cached within the JVM for future use. A major advantage of dynamic class loading is the conservation of device memory, as only the classes that the application needs will be loaded, and the memory used to cache the classes can be reclaimed once the JVM stops executing.
A more efficient method may be to serialize the required classes with the migrating object, and send these with the object as it moves. This would require knowledge of the classes already loaded by the remote JVM to avoid sending classes unnecessarily. This reduces the need for a code base server, and may reduce the communication between MTMs if class caching was implemented.

4.6 Limitations

Due to the design of the MTM, several constraints are placed upon the operation and development of applications.

Certain objects types cannot migrate due to ties to physical resources as described in section 2.3.2. Such Java objects include many of the classes in the `java.io` package, and any heavy-weight (ie operating system rendered) GUI components such as `java.awt.Frame`. No migration testing has been performed on light-weight GUI objects such as Java Swing components, but no problems are envisaged as long as migration policies specify a strong co-locality between all GUI components, as there is currently no support within the MTM for the distribution of GUI objects across devices. GUI migration is a topic of further research.

Due to the serialization of full object trees on migration, any object that must be shared by several non migratory application objects must be defined as a MobJeX object. If this were not to occur, these objects would be copied to the remote device when a referencing object migrates. This can be enforced by the preprocessor at compile time if necessary.

Limitations are placed on how an object can be accessed and manipulated due to the proxying wrapper classes used. As remotely invocable methods must be defined in the remote interface as *public* for use with RMI, no direct access to protected methods is supported. This could be achieved by defining protected methods as public but requiring that these methods be invoked via the MTM to allow the enforcement of Java language access control via Reflection. Additionally, no direct manipulation of an object's data members is permitted, as this is not supported over RMI. This could be overcome by implementing accessor and mutator methods for each data member, which can be generated and integrated into application code by the MobJeX compiler. This limitation is a topic of further research.

The MTM is currently implemented as a single component, with no capacity for moving the responsibilities of the MTM between devices. This restricts the devices able to support a MTM component, as object creation relies on Java Reflection which is currently not supported in J2ME. As a consequence, the goal of device heterogeneity was not fully met, as despite the MTM being fully compatible with Java 2 Standard Edition, the current implementation is not supported by J2ME, severely limiting it's usage on devices with limited capabilities (such as mobile phones). Ideally the MTM itself will be adaptive, allowing characteristics of it's behaviour to change depending on the capabilities of the device on which it is running. This would allow the delegation of all functions requiring Reflection to

the server in the case that a device does not support it. The separation of MTM components may also allow the user or service provider to make their own decision as to the method of reference management used (recursive update or lazy tracking). The optimal partition of MTM components is an area of future research.

5 Results

In order to answer the questions described in section 3, five tests were devised. Each test is discussed individually below.

Testing of object migration and usage was performed over two single CPU 2GHZ Pentium 4 machines each with 512MB RAM, connected via an isolated local area network. The MTM was compiled and tested using the Java 2 SDK version 1.4.1. FarGo was chosen for comparative purposes, as it is well suited to a next generation Internet environment, is well documented and freely available for testing. FarGo version 0.2.2, requires Java 1.1 therefore it was tested using the JDK version 1.1.8.

To allow testing of both object migration and object reference management efficiency, test objects for both FarGo and the MTM were created containing an array to hold remote references. The implementation and method signatures of objects for both FarGo and the MTM were identical, but differed in the inheritance of classes and interfaces due to the requirements of each framework.

5.1 Test 1 - Object Creation Efficiency

In order to provide an evaluation of the overhead of object creation in both the MTM and FarGo, and to answer research question 6 regarding the efficiency of the MTM, a comparison of object creation efficiency in both FarGo and the MobJeX Transport Manager was conducted. To evaluate the overhead of creating an object via reflection, as is the case when creating an object by name through the MTM, both creation by name and standard Java object creation by the MTM were tested. The performance of standard Java object creation is provided as a baseline for comparison of FarGo and MTM creation times.

Standard Java Creation	125
FarGo	612
MTM (by existing object)	4141
MTM (by name)	4391

Table 1: Average time (ms) taken to create 1000 objects.

Table 1 shows the time taken to create 1000 application objects with each different configuration. These results show that there is a significant increase in the time taken to create an object through the MTM, either by the objects name, or by passing it an object created in application code. It it important to note that both FarGo and the MTM create another object in addition to the implementation of the application object. In the case of FarGo, a tracker object is created directly by the application code, which in turn creates an application object. As the class types are compiled in to the generated tracker

code, no reflection is required to create any objects. This advantage is due to the fact that no interchangeable object implementations are supported, and as a result class names can be compiled into tracker classes, removing the need for reflection to create object implementations and trackers. As the choice was made for the MTM to support remote object creation and object implementation substitution, compilation of class names into proxy code is not possible. Therefore, the MTM's object creation speed is heavily disadvantaged by the use of reflection to create proxies. Additionally, if objects are created by name through the MTM, another reflective object creation must take place. This overhead is justified by the extra functionality added by controlling the creation of objects. Additionally, the high overhead involved in object creation could be partly due to inefficiencies in the algorithms used to create an object within the MTM, as no optimisation of this process has been attempted.

5.2 Test 2 - Object Migration Efficiency

In order to provide a comparison of different object referencing techniques, and to test the goal of efficient reference management (section 3, goal 6), a comparison of object migration and reference management efficiency in both FarGo and the Mobjex Transport Manager was conducted.

As the amount of references contained in an object affects the load involved in migrating an object, an array of references was held in each object, allowing the testing of migration with a varying amount of reference updates needed per object. Evaluation of the MTM was conducted for both the active update proxying and lazy update methods of reference management, to provide an analysis of the efficiency of both methods.

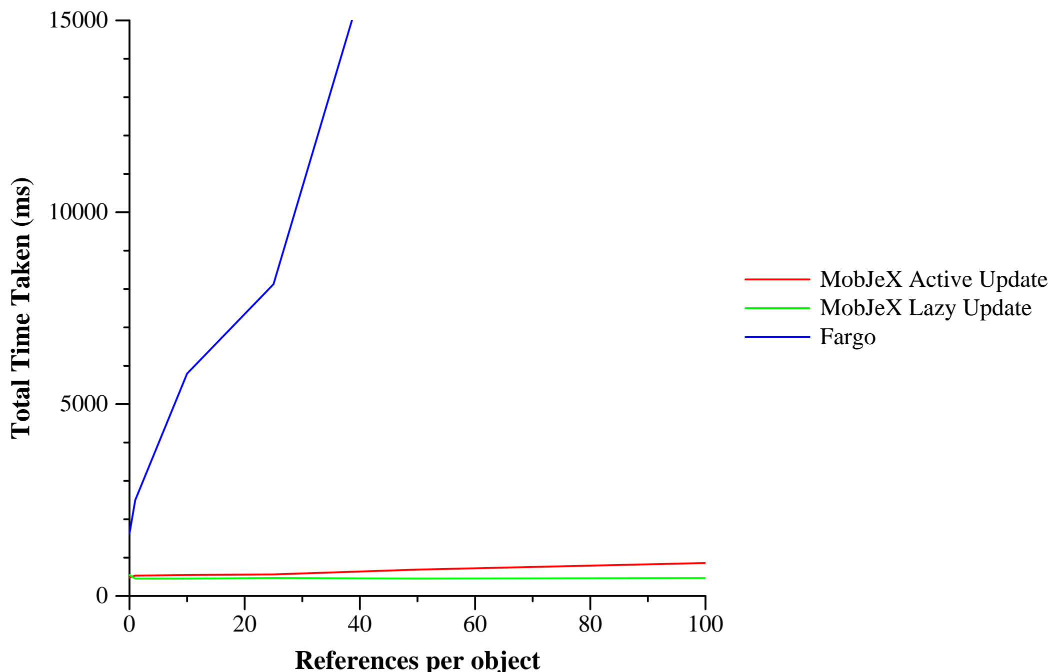

Figure 6: Object migration times for 100 objects with increasing external reference counts.

Figure 6 shows the object migration times for 100 objects, over a range of external object reference values. The purpose of this test is to show the scalability of each migration and reference management strategy, and to compare migration times of the MTM in both modes against FarGos object movement.

It can be seen that the movement of objects in FarGo is significantly slower than in the MTM, regardless of what reference management strategy used. Although not represented in the figure, times for migration of object with 50 external references were upwards of 20 seconds. When migrating objects with 100 external references, the receiving FarGo core began displaying memory errors and would not respond to requests. Since results of efficiency testing of FarGo were not published, it is hard to tell whether this is caused by the design of the framework, or by an error in the version of the framework used. As no source code was available for FarGo, no further investigation into the reason for this inefficiency is possible.

These results also show that there is a linear relationship between the number of an objects external references and the time required to migrate that object. This could be due to the fact that for each unique reference held in an object, a copy of the proxy representing that reference is serialized and migrated along with the object. Although the data added by a serialized proxy is not significant, the cumulative cost of serializing and de-serializing a migrated object would be affected by the number of proxies held within the object.

It is also shown that lazy update reference management was found to be slightly faster than active update reference management by the MTM. This is due to the fact that no proxies are updated on object migration when using lazy update reference management (see section 4.2), and instead are updated when a method is next invoked via each proxy. The low times exhibited by active proxying reference management may have been aided by the simplicity of the test object structure. As all external object links are held in an array, no excessive deep searching would be required to update all links. If the object referencing was structured in such a way that proxies were embedded in objects several references away from the migrating object, migration times may be much higher.

5.3 Test 3 - Remote Object Method Invocation Efficiency

To effectively test the efficiency of object reference management on object migration, reference updates were tested with the following scenario:
Half the objects in the set did not move. All other objects were linked to objects that did not move, and then migrated to the remote server. Every link held in the migrated objects was then accessed twice. This allowed the comparison between active proxying and lazy update techniques as implemented in the MTM (see section 4.2).

Figure 7 shows the time taken to access all object references in all objects migrated to the remote device. This ensures that all object links are valid, and requires any invalid references to be updated when accessed. This shows a significant difference between accessing a FarGo object via it's tracker, and accessing a MobJeX object via it's proxy. This may be due to all method invocations being passed via a FarGo core to pass remote method invocations to the remote tracker. As FarGo optimises tracker links on the first invocation after an object has moved, one would expect the second invocation to be more efficient. This is not reflected in these results, possibly due to the simplicity of the object setup, with the tracker being immediately pointed to the new location of the object. If the object were moved to another device before it was invoked, the initial access time may increase. The results show that the first access to an object when using lazy tracking is significantly slower than when proxies are updated on migration. This is to be expected, as the proxy must first be updated before the request can be fulfilled. As each proxy is updated individually, the overheads of updating all proxies are high due to each proxy's reference being retrieved individually from the remote device. Subsequent

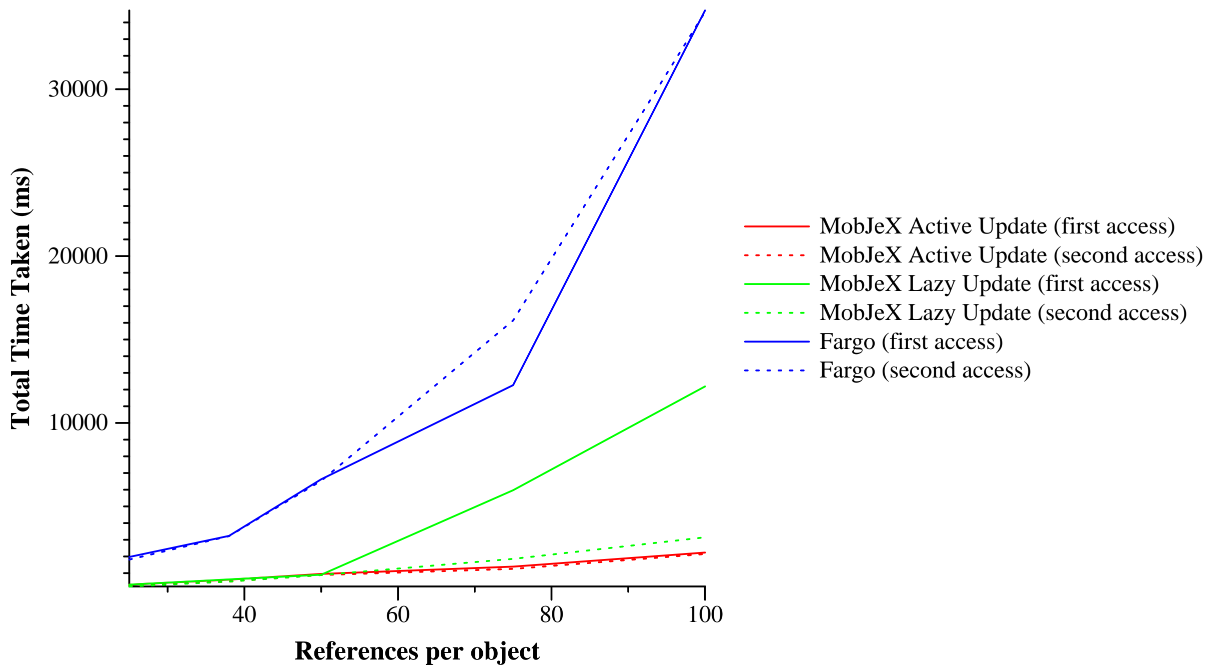

Figure 7: Object access times for objects with a varying number of external references.

invocations of an object took approximately the same time as those made to proxies already holding a valid object reference. As proxies always hold a valid object reference when active proxying reference management is used, the times for both first and second invocations do not differ.

5.4 Test 4 - Object Location Transparency and Reference Integrity

In order to answer the research questions regarding location transparency and integrity of object linking (section 3, questions 2 and 3), a test of object reference integrity was conducted. This test consisted of the creation of 100 objects on the server system, 50 of which were migrated to the client. To facilitate the testing of all scenarios of object linking, each non-mobile object was linked to one mobile object, and each mobile object linked to a different non-mobile object. Figure 8 shows an example of this reference structure for 4 objects. Once all mobile objects were migrated, the first object in the list was accessed, which in turn accessed the reference it held. This step was repeated until the end of the object 'chain' was reached. This ensured that every object was accessible from either device after migration. For comparative purposes, this test was conducted with both FarGo and the MTM. Both FarGo and the MTM completed this test successfully. If was found that with both the active update and lazy tracking methods of reference updates, that all objects were accessible from both locations after object migration. Furthermore, the application needed no knowledge of the location of any given object, as all objects were accessed as if they resided on the local device. This satisfies the goal of location transparency as stated in section 3.

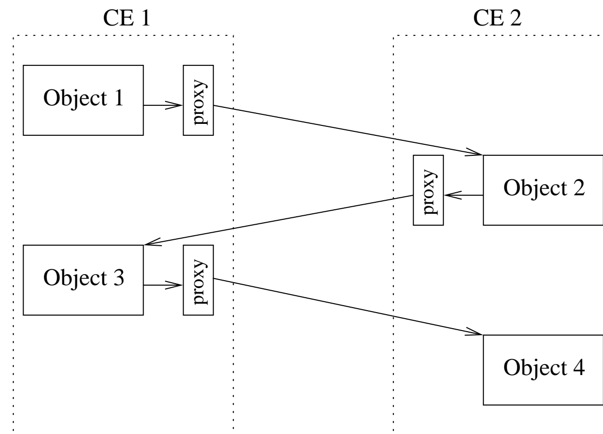

Figure 8: Example object reference structure for link testing.

5.5 Test 5 - Object Substitution

To test how well the MTM meets the previously described goal relating to seamless object substitution (section 3, goal 5), a test of object implementation substitution was performed.

The test scenario consisted of a single object interface, with two implementations provided for the same object. A test application was also developed which accessed the object both before and after object substitution. This allowed us to test the outcome of substituting one implementation with the other during a program's execution.

The result of this test showed no interruption in the execution of the test application whilst the behaviour of the object was changed. All object references were maintained, and object implementations substituted without incident.

5.6 Limitations

As the only available version of FarGo required Java version 1.1 to operate correctly, the efficiency of the Java 1.1 JVM may have limited the efficiency of FarGo. As far as we are aware, FarGo is no longer maintained, and no source code is available to allow it to be updated to use the latest Java technologies.

The design of test objects limited the testing of deep recursive proxy searching, as all references were held in a single array in the object. This caused unusually low update times for active proxy reference management, as no recursive traversal of an object tree was required. This could be improved by testing with a more realistic object structure, with a random distribution of proxies within all objects.

6 Summary and Conclusion

This thesis has reviewed existing techniques for the management of application adaptation in a distributed environment, in particular the use of object mobility as an effective technique to distribute

application components across heterogenous devices. An analysis of the shortcomings of existing work in object mobility showed the need for a solution specifically designed for adaptation in next generation Internet applications, with support for dynamic object mobility, grouping and binding in response to device capabilities and resource usage.

The implementation and testing of suitable object mobility techniques was conducted, with the conclusion that the solution presented is suitable for use in a next generation Internet environment. The MobJeX Transport Manager supports the dynamic object mobility and grouping of objects in a way transparent to both users and application developers, whilst maintaining a level of efficiency surpassing that of similar systems in most situations. Transparency and the support for dynamic object mobility was achieved via a separation between an objects implementation and it's interface, allowing the use of a 'proxy' reference in place of a direct object reference. This allows object instances to move freely between devices, with proxy references updated either on object migration, or when the first attempt to access a migrated object is made.

This design facilitated dynamic migration and substitution of objects within the application, allowing the customisation of adaptation policies depending on device capabilities and resource availability, within the MobJeX framework of which the work presented in this thesis forms a core.

6.1 Future Work

Further application testing is required, as this thesis has focussed only on the evaluation of chosen techniques from an efficiency and suitability point of view. Testing of the framework with existing applications is important to assess the potential value added to distributed applications with the use of the MobJeX framework, in particular the MTM component.

In addition to further evaluation of the framework, the following improvements to the MTM are subject to further research.

Within the current implementation of the MTM, no remote class loading is available. Future improvements to the MTM include the integration of class loading into the MTM, allowing classes to be sent with migrating objects if the device does not already have the class. This would eliminate the need for a separate codebase server, and allow class file transport to be taken into account when making decisions regarding the movement of objects.

Alternative transport mechanisms such as SOAP-RPC will be examined to assess their suitability to the MobJeX framework. As the MTM matures, support for existing interoperability standards such as Jini service discovery may be included.

Additional future work will involve the optimisation of all algorithms and the partitioning of MTM responsibilities into separated components. This will improve support for less capable devices, as functions of the MTM not supported by devices can be delegated to the server, with the device's MTM containing the minimum amount of functionality required to participate in adaptation. The empirical testing of the MTM's efficiency in a range of situations is planned, to confirm the heterogeneity of the MTM and it's components.

As the MobJeX framework is a work in progress, future findings regarding the specification of adaptation policies and object grouping strategies may affect the decisions made with regards to the MTM.

Additionally, ongoing work into Java thread migration [Dor03] is directly related to the strong mobility of application objects, a goal of the MobJeX framework.

Acknowledgements

I would like to thank my supervisor, Dr. Caspar Ryan for his much appreciated help in the development of ideas and the MobJeX framework. I would also like to thank my fellow honours students Andrew, Rebecca and Simon for their ongoing support and assistance.

References

[ARS97] A. Acharya, M Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, pages 111–130. Springer-Verlag, Heidelberg, Germany, 1997.

[BC95] K. A. Bharat and L. Cardelli. Migratory Applications. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag: Heidelberg, Germany, 1995.

[BCM03] P. Bellavista, A. Corradi, and R. Montanari. Dynamic Binding in Mobile Applications : A Middleware Approach. *IEEE Internet Computing*, 7(2):34 – 42, 2003.

[BP98] M. Baldi and G.P. Picco. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In *Proceedings of the 20th International Conference on Software Engineering*, pages 146 –155, Kyoto, Japan, 1998.

[BPSMM00] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition), 6 October 2000. `http://www.w3.org/TR/2000/REC-xml-20001006`.

[BSCHL97] I. Ben-Shaul, A. Cohen, O. Holder, and B. Lavva. HADAS: A Network-Centric Framework for Interoperability Programming. *International Journal of Cooperative Information Systems*, 3:294–314, 1997.

[BSLS01] N. M. N. Bouraqadi-Saâdani, T. Ledoux, and M. Südholt. A Reflective Infrastructure for Coarse-Grained Strong Mobility and its Tool-Based Implementation. Technical report, Ecole des Mines de Nantes, France, 2001.

[But02] M. H. Butler. Using capability classes to classify and match CC/PP and UAProf profiles. Technical report, HP Laboratories, 2002.

[CBNH98] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel. The Mobile Object Layer: A Run-Time Substrate for Mobile Adaptive Computations. *ISCOPE*, pages 71–82, 1998.

[Dor03] A. Dorman. *Execution Context Migration within a Standard Java Virtual Machine Environment*. Honours thesis, RMIT University, 2003.

[Fri03] D. Friedman. Java Implementation of a Resource Performance Monitor API. Technical report, RMIT University, 2003.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison Wesley Longman, Inc., Massachusetts, USA, 1994.

[HBSG99] O. Holder, I. Ben-Shaul, and H. Gazit. Dynamic Layout of Distributed Applications in FarGo. In *21st Int'l Conf. Software Engineering (ICSE'99)*, pages 163 –173. ACM Press, 1999.

[IBM03] IBM. Next Generation Internet, 2/10/2003 2003. `http://www.ngi.ibm.com`.

[IR97] Y. Ichisugi and Y. Roudier. Extensible Java Preprocessor Kit and Tiny Data-Parallel Java. *ISCOPE*, 1997.

[J2M00] J2ME Connected, Limited Device Configuration Specification. Technical report, Sun Microsystems, 19 May 2000. `http://jcp.org/aboutJava/communityprocess/final/jsr030`.

[Jav03] Java Remote Method Invocation Specification, 2003. `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`.

[JHE99] J. Jing, A. Helal, and A. Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, 31(2):118–157, 1999.

[JNI97] Java Native Interface Specification. Technical report, Sun Microsystems, May 16 1997. `http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.htm%l`.

[JVM99] The Java Virutal Machine Profilier Interface (JVMPI). Technical report, Sun Microsystems, 1999. `http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html`.

[Kra96] D. Kramer. The Java Platform - A White Paper. Technical report, Sun Microsystems Inc., 1996.

[KYME03] A. Kinno, Y. Yonemoto, M. Morioka, and M. Etoh. Environment Adaptive XML Transformation. In *2003 Symposium on Applications and the Internet (SAINT'03)*. IEEE, 2003.

[MDP+00] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys*, 32(3):241 – 299, 2000.

[MDW99] D. Milojicic, F. Douglis, and R. Wheeler, editors. *Mobility - Processes, Computers and Agents*. The Association for Computing Machinery, Inc., Massachusetts, USA, 1999.

[MES95] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploting Weak Connectivity for Mobile File Access. In *15th ACM Symposium on Operating Systems Principles*, volume 29, pages 143 – 155, USA, 1995.

[Ref03] Java 1.4.2 Reflection Guide, 2003. `http://java.sun.com/j2se/1.4.2/docs/guide/reflection/`.

[RP03a] C. Ryan and S. Perry. Client/Server Configuration in a Next Generation Internet Environment: End-User, Developer, and Service Provider Perspectives. In *2003 Australian Telecommunications, Networks and Applications Conference (ATNAC)*, Melbourne, Australia, 2003.

[RP03b] C. Ryan and S. Perry. Dynamic Application Configuration using Mobile Java Objects in a Next Generation Internet Environment: End-User, Developer, and Service Provider Perspectives. *Submitted for publication*, 2003.

[RP03c] C. Ryan and S. Perry. MobJeX: A Declaratively Configurable Java Based Framework for Resource Aware Object Mobility. In *On The Move Federated Conferences (OTM '03) Workshops*, Catania, Sicily, 2003. Springer-Verlag.

[SNKP95] M. Satyanarayanan, Brian Noble, Puneet Kumar, and Morgan Price. Application-aware adaptation for mobile computing. *ACM SIGOPS Operating Systems Review*, 29(1):52–55, 1995.

[TP01] E. Tanter and J. Piquer. Managing References upon Object Migration: Applying Separation of Concerns. In *21st Int'l Conf. Chilean Computer Science Soc. (SCCC'01)*, pages 264 – 272. IEEE Press, 2001.

[WAP00] Wireless Application Protocol White Paper. Technical report, Wireless Application Protocol Forum, January 2002 2000.

A Application code modifi cation example

The following is a sample of the modifications required to a simple ‘Hello World’ type application. Changes shown below are necessary for original code to be made compatible with the MTM object model. These changes can be fully pre-processed prior to compilation.

Original Application Code

```
1   public class HelloWorldObject extends classA
2   {
3     private String helloMessage;
4
5     public HelloWorldObject(String helloMessage)
6     {
7       this.helloMessage = helloMessage;
8     }
9
10    public String getHelloMessage()
11    {
12      return helloMessage;
13    }
14
15    public void setHelloMessage(String helloMessage)
16    {
17       this.helloMessage = helloMessage;
18    }
19  }
20
21  public class HelloWorldApplication
22  {
23
24    public static void main (String[] args)
25    {
26      HelloWorldObject hello = new HelloWorldObject("Hello World");
27
28      hello.setHelloMessage("Hello Again");
29      System.out.println(hello.getHelloMessage());
30    }
31  }
```

Modified Classes

```
1 // class renamed from HelloWorldObject to HelloWorldImpl
2   public class HelloWorldObjectImpl implements HelloWorldObject
3   {
4     private String helloMessage;
5
```

```
6     public HelloWorldObject(String helloMessage)
7     {
8       this.helloMessage = helloMessage;
9     }
10
11    public String getHelloMessage()
12    {
13      return helloMessage;
14    }
15  }
16
17  public class HelloWorldApplication
18  {
19
20    public static void main (String[] args)
21    {
22      HelloWorldObject hello =
23          TransportManagerImpl.getInstance().create(
24              "HelloWorldObject",
25              new Object[] {"Hello World"}
26              );
27
28      hello.setHelloMessage("Hello Again");
29      System.out.println(hello.getHelloMessage());
30    }
```

Additional Classes

```
31  // interface generated from methods in original HelloWorldObject
32  public interface HelloWorldObject extends MobJeXObject
33  {
34
35    public String getHelloMessage()
36        throws java.rmi.RemoteException;
37
38    public void setHelloMessage(String helloMessage)
39        throws java.rmi.RemoteException;
40  }
41
42
43  // proxy generated from HelloWorldObject interface
44  public class HelloWorldObjectProxy extends AbstractMobJeXProxy
45      implements HelloWorldObject
46  {
47    public String getHelloMessage()
48    {
```

```
49      try {
50         // check for proxy validity
51         this.lazyUpdate();
52         return ((HelloWorldObject) object).getHelloMessage();
53      }
54      catch(java.rmi.RemoteException re) {
55         // error handling and recovery performed here
56      }
57    }
58
59    public void setHelloMessage(String helloMessage)
60    {
61      try {
62         // check for proxy validity
63         this.lazyUpdate();
64         ((HelloWorldObject) object).setHelloMessage(helloMessage);
65      }
66      catch(java.rmi.RemoteException re) {
67         // error handling and recovery performed here
68      }
```