

OPTIMISING TRANSPARENCY, PERFORMANCE AND RELIABILITY IN THE MOBILITY SUBSYSTEM OF A MOBILE OBJECT FRAMEWORK

CHRISTOPHER WESTHORPE

Supervisor: DR. CASPAR RYAN

Honours Thesis

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

October, 2004

Abstract

This thesis describes a number of advances made in order to produce a mobile object framework that is better suited for use in ad hoc networks than its predecessors. The focus of this thesis is on improving the performance, transparency and reliability of the mobility subsystem, which is responsible for the migration of objects from one device to another.

Based on a comprehensive review of the literature, a new mobility subsystem has been proposed and implemented. The resulting runtime environment and proxy structure provides increased performance, and by managing erroneous situations within the proxies and supporting architecture, the reliability and transparency have been increased to both developers and end users.

A series of empirical tests are executed against the new mobility subsystem, demonstrating the advantages of its reference management strategies and its fitness for use in ad hoc networks of heterogeneous devices.

Contents

1	Introduction	4
2	Rationale	4
3	Literature Review	6
3.1	Framework Overview	7
3.2	Declaration	8
3.3	Runtime Environment	9
3.4	Object Creation	9
3.5	Proxies	9
3.6	Managing Local and Remote References	12
3.7	Static Methods	13
3.8	Error Handling	14
4	Research Questions	15
5	The MobJeX Mobility Subsystem	15
5.1	System Architecture	17
5.1.1	Service	17
5.1.2	Runtime	18
5.1.3	Transport Manager	18
5.1.4	Mobject	19
5.2	Declaration	19
5.3	Object Creation	20
5.4	Proxies	20
5.4.1	Domain	21
5.4.2	DomainRemote	21
5.4.3	DomainImpl	21
5.4.4	DomainProxy	22
5.5	Managing Local and Remote References	23
5.6	Static Methods	23
5.7	Error Handling	24
5.7.1	Proxy	25
5.7.2	Transport Manager	25
5.7.3	Runtime	25
5.7.4	Service	26
5.7.5	System Controller	27
5.7.6	Summary	27
6	Evaluation	27
6.1	Limitations	28
6.2	Remote Get/Set Test	28
6.3	Remote Stagger Test	31
6.4	NGI Device Test	33
6.5	Error Handling Test	33

7 Conclusion and Future Work	34
A Sample Code Transformation	38
B Complete Test Results	40

1 Introduction

The next generation of Internet capable devices will be heterogeneous in hardware, infrastructure and usage patterns. Targeting these devices is currently difficult and tedious because there is no single common programming language or framework. This means that a single code base cannot be developed and deployed to all machines in the environment.

Next Generation Internet (Moyer and Umar, 2001) (NGI) devices such as PDAs and smart phones are becoming increasingly popular. The processing power of these devices is on the increase, hence it becomes possible to utilise additional resources in order to run more complex applications. An early approach to mobile applications was to run ‘cut-down’ versions of desktop applications with reduced functionality on mobile devices. However, due to the increased processing power available to the device, it becomes possible to use the same application code on mobile devices as on desktop computers. As the amount of code required to cater for specific devices is reduced, application development costs are reduced proportionately. If applications can be run on all devices in an NGI environment then it becomes possible to migrate those applications that have been traditionally run on higher-powered centralised servers to client devices, remote from the server.

Mobile Object Frameworks allow for individual application components to migrate between different Computational Environments (CE) at runtime. A mobile object framework is responsible for ensuring that references between objects remain intact, regardless of the physical location of implementation components. This functionality allows objects to be distributed over many devices across a network, allowing each application to maximise their usage of available resources within the network.

Some mobile object frameworks have the capability to make autonomous decisions based on changing computational environmental conditions such as memory or CPU load. Such systems are known as *Adaptive Mobile Object Frameworks* (Jing et al., 1999) and they utilise *asynchronous mechanisms* (Spiegel, 2000) for object migration. These systems allow application components to be dynamically redistributed over the environment in response to changing environmental conditions, hence maximising resource use and providing better user response times.

If the decisions for object migration are not made by the framework but rather by the application itself then *synchronous mechanisms* (Spiegel, 2000) for object migration are utilised. Hence the act of migrating an object is “tied to the control flow of the program” (Spiegel, 2000), when the program reaches a certain point in its execution the object is migrated. Such systems are useful in the context of parallel programming as tasks can be statically distributed over fixed networks, however they lack the flexibility to respond to a changing environment.

2 Rationale

Having previously introduced terminology and provided a background of NGI and mobile object frameworks, this section will provide the reasoning behind active research into the area of mobile object frameworks and methods of migration.

In the context of application frameworks, transparency can be defined as a measure of the impact the framework has on the program code. Alternately, transparency can be viewed as the extent to which existing code must be modified or new code written to gain the features provided by the framework. The benefits of a mobile object framework transparent to the developer, end user and system have been identified (Ryan and Perry, 2003). According to Ryan and Perry, transparency benefits developers because they are not required to code specifically for the framework, while end users are isolated from migration details in their daily execution of applications. Transparency should be in-

corporated throughout such a framework via the use of an asynchronous mechanism that dynamically migrates objects based on environmental factors. In an NGI environment these factors could include battery life and wireless network signal strength for mobile devices such as PDAs.

The framework should also be transparent to the system, such that no modification of the standard execution environment is required.

An adaptive mobile object framework as described in Section 1 can be broken down into two core subsystems or responsibilities. The first subsystem is responsible for the migration of objects from one CE to another and is hereafter referred to as the *Mobility Subsystem*. The second subsystem is responsible for decisions governing when objects can or should be migrated and is hereafter referred to as the *System Controller*. The System Controller is a component within the runtime system which makes changes to the locations of objects based on informed decisions about the state of the objects and their interactions (Spiegel, 2000). In some systems, the System Controller is a separate centralised application that utilises the migration services of the mobility subsystem in order to achieve performance gains in an application or network. This thesis is concerned with optimisations of transparency, performance and reliability that can be made within the mobility subsystem of such adaptive mobile object frameworks.

Ideally, a mobile object framework should be able to leverage existing code (Chiang, 2001). The changes that may be required to enable such applications to be used within the framework may be complex. In order to provide transparency to the developer, any code changes required for object mobility must be able to be automatically inserted into the code. This transformation may occur by automatic source code manipulation or by bytecode instrumentation at runtime (Bruneton et al., 2002).

A core requirement of mobile object frameworks designed for use in NGI environments is the need to migrate objects to any number of heterogeneous devices. As these devices by definition have varying hardware and software, support in the form of consistent programming interfaces is not always readily available. Devices such as PDAs and smart phones support operating systems and programming languages different to that of traditional desktop computers and thus there is not always a compatible medium between the two. New devices being developed have support for a number of different programming environments. These environments (for example Java, SOAP and .NET) are being developed with the goal of being cross platform, however because a number of competing standards are currently under development, the result is a market that is again becoming fragmented. Device manufacturers and programmers cannot agree upon a single 'cross platform' language.

The Virtual Machine (VM) approach encourages developers to develop 'write once run anywhere' applications, which remain cross platform as long as there is a VM on the device that supports the language. Therefore, any mobile object system destined for operation in a heterogeneous environment requires a VM in order for the same executable artifact to migrate between disparate hosts. Java and .NET provide two such environments, however the majority of developers choose Java, as the Java VM (JVM) (Lindholm and Yellin, 1999) offers more accessible support for remote methods in the form of its RMI libraries (Sun Microsystems, 2003).

NGI networks are ad hoc in nature with nodes joining and departing as they come within range of each other. This ad hoc nature allows for dynamic group formation to become possible. Using existing wireless technologies disconnections are common (Chatzigiannakis and Nikolettseas, 2004), thus departure cannot always be assumed voluntary. Hence, frameworks designed for use in this environment must be able to correctly deal with network errors, enabling applications to continue operating and thus minimising the impact of the network on users.

In summary, development of applications for mobile devices is currently tedious, since applications written for more powerful devices cannot always be easily adapted. In order to harness the increasing power and portability of mobile devices, it is feasible to dynamically migrate applications

and application objects between mobile and fixed devices based on available resources. A transparent mobility subsystem is beneficial to the system, developer and end user, with transparency being aided through the use of tools able to convert existing application code. A mobility subsystem designed for NGI environments must provide error correction mechanisms that deal with network faults, ensuring a consistent user experience. To support such a subsystem, a layer of abstraction in the form of a Virtual Machine is required to separate hardware level details from application code.

Given the issues identified in this section the rest of this paper is organised as follows: Section 3 contains a review of current literature regarding the state of the art in object mobility, with existing frameworks compared and contrasted, whilst Section 4 poses research questions arising from that review. Section 5 presents documentation of the new system developed in order to answer the Research Questions, with Section 6 providing empirical tests contrasting the solution against existing systems. Finally, Section 7 provides a conclusion and suggests a number of paths for further research.

3 Literature Review

A number of approaches have been identified to implement the core behaviour of a mobility system, with the proxy-based approach described by Shapiro (Shapiro, 1986) being the most common. This approach minimises the change required for objects in existing code to become *Mobile Capable*, that is able to migrate from one CE to another. This section presents a review of a number of such mobile object frameworks which utilise Shapiro's Proxy Principle.

Systems included in this review are FarGo (Holder et al., 1999a), Voyager (Recursion Software Inc, 2004), JavaParty (Philippsen and Zenger, 1997) and Mobile Java Objects (MobJeX) (Ryan and Perry, 2003). A discussion of earlier work in the Sumatra (Acharya et al., 1997) project as well as related ideas and concepts present in Proactive (INRIA, 2004) is also presented.

Object migration can take a number of different forms, Carzaniga, Picco, Vigna (Carzaniga et al., 1997) broadly categorise migration into *strong migration* and *weak migration*. Strong Migration is the process of migrating an object while it is in a state of execution. The object itself is moved to the remote host along with all of the current state information, allowing the object to migrate pre-emptively to a remote host and continue processing. Strong migration is possible without the object or process having any knowledge of the migration occurring if a copy of its current execution context is also migrated. In contrast, weak migration is the process of only moving the object instance itself, it is non-preemptive and hence the object must have knowledge of how to execute the migration. A review of the literature did not identify a mobile object framework that supported strong mobility in a transparent manner. All strong mobility frameworks identified required either developer instrumentation of code or modification to the standard Java execution environment in order to transfer the execution context. This requirement of JVM modification precludes the frameworks from consideration as they violate transparency, one of the main goals of this thesis. Consequently, mobile object frameworks discussed in this review only expose mechanisms for weak mobility, therefore future references to mobility can be assumed to refer to weak mobility unless otherwise specified.

FarGo and MobJeX are examples of complete adaptive mobile object frameworks as discussed in Section 2. This review will only address the mobility subsystem aspect of such projects since this thesis is concerned with optimising the transparency, performance and reliability of such systems. Voyager and JavaParty provide mobility support in the form of synchronous mechanisms which can be utilised by application developers to enable object migration within their applications. Projects such as Pangaea (Spiegel, 2000) provide a System Controller component to complement the synchronous mechanisms provided by these frameworks, however a review of such systems is again beyond the

scope of this thesis. For a more in-depth literature review of adaptive frameworks containing issues not directly related to their mobility subsystem, see Perry (Perry, 2003).

All of the systems reviewed are implemented in Java as it provides the Virtual Machine environment necessary for NGI environments as discussed in Section 2.

A number of factors are relevant in the definition and evaluation of mobility systems and thus to adequately cover each of these important areas the rest of this review is organised as follows. Subsection 3.1 addresses the history and aims of each project in order to review them in the correct context. Subsections 3.2 to 3.4 are concerned with aspects of individual implementations. Techniques for specifying object mobility, capabilities and responsibilities of the runtime environment and instantiation and transparency of remote objects are covered in these subsections. Subsections 3.5 to 3.8 focus on features provided by the mobility system and details the structure and responsibility of proxies, optimisation of local references, static method support in proxies, and error handling techniques.

3.1 Framework Overview

Holder, Ben-Shaul and Gazit present the FarGo (Holder et al., 1999a) project for the development of mobile applications. Since the FarGo project is the successor to the HADAS (Ben-Shaul et al., 1997) framework, this review treats Hadas and FarGo as the same project and does not provide a separate review of HADAS. An extension to the FarGo framework called FarGo-DA (Weinsberg and Ben-Shaul, 2002) has added support for “disconnected operation” and is currently under development, however there is no implementation of that framework currently available.

Voyager (Recursion Software Inc, 2004) is a commercial object request broker developed by Recursion Software that supports object mobility.

Philippsen and Zenger propose the JavaParty framework (Philippsen and Zenger, 1997) for mobile objects. The aim of the JavaParty project is to add mobile capable objects to Java by simple declaration. The core focus of JavaParty is on adding distributed and parallel programming support to heterogeneous workstations in clustered environments.

MobJeX is an adaptive mobile object framework currently under development that aims to improve on existing systems by having transparency to the developer, user and system as a core aim. The MobJeX framework specifically caters for ad hoc networks (Ryan and Westhorpe, 2004) and provides declarative rules for specifying object migration. In effect MobJeX aims to provide an implementation of weak mobility and build in some of the strong migration features such as preemption and migration without application awareness. Of most relevance to this thesis is an initial prototype of the MobJeX mobility subsystem that has been implemented as a proof of concept, known as the Transport Manager (Perry, 2003), but referred to in this thesis as Perry’s Transport Manager (PTM). The relevance of PTM stems from its similar proxy structure (Section 5.4) and its role in the evolution of the MobJeX framework. PTM differs in its nature from the other frameworks under review as it is strictly client-server in nature with the focus being on web-based applications, while other systems strive for a more peer-to-peer or clustered approach.

MobJeX (Ryan and Perry, 2003) and JavaParty have similar goals, in that both systems aim to provide object mobility while minimising the change made to existing programs in order for them to be incorporated. However, MobJeX is a full adaptive mobile object framework while JavaParty only provides the mobility system of such a framework.

The following subsections present an analysis of each framework on a factor-by-factor basis.

3.2 Declaration

In each of the frameworks reviewed, it is necessary to explicitly declare which classes are to be mobile capable. It is possible that instances of these classes will not be migrated during the execution of the application, however they do have the capability to migrate if required. The requirement of declaration is necessary so that the framework only compiles or manipulates the necessary classes; placing its proxy structure in as surrogates for implementation objects. There are two approaches for explicit declaration of mobile capable classes, firstly application code can be modified to contain keywords or reflect an object structure or secondly the declaration can be independent from the code and specified externally. JavaParty and FarGo require internal modification of application code, whereas the PTM provides mechanisms for external declaration and Voyager does not require explicit declaration.

Unlike the mobile classes used in other frameworks, the smallest unit of movement in FarGo is the *complet*. While a complet could reflect a single class in an application, FarGo recommends that a complet reflect an entire subsystem or component. The FarGo system allows for complet co-locality, such that if one complet is migrated, others will migrate to that location if possible. Existing classes are retrofitted into complets in a two stage process involving implementation of an empty interface and renaming of the file containing the class definition (Holder, 1999). Each class modified in this way must be run through the FarGo pre-compiler, `fargoc`, which creates the proxy structure required.

Mobile capable classes are specified in JavaParty via the use of a `Remote` keyword, hence the smallest object that can be migrated is a single `Remote` class. Code is then parsed with a pre-compiler before compilation that inserts the additional Java code required to support mobility. As JavaParty classes are statically distributed there is no mechanism for specifying co-locality constraints.

In order for a class to be mobile capable in the Voyager system it must implement a Java interface that defines all publicly available methods (Wheeler, 2002). When these classes are used in situations where they need to be mobile capable, a special syntax is used for their creation (described in detail in Section 3.4), hence the declaration is dynamic. This approach provides a level of transparency on par with JavaParty and FarGo as developers are still required to write additional code in order to support the framework. There is no means for specifying co-locality constraints in Voyager, as like JavaParty distribution is assumed to be static.

The Proactive system proposes an alternative to code modification whereby it uses a system of XML-based descriptors (Baude et al., 2002) which describe the deployment scenario of the distributed application.

Perry's Transport Manager incorporates a similar approach to Proactive by utilising XML to specify the deployment of classes, however the approach is extended by using XML to define both mobile capable classes and their co-locality constraints. An XML file containing a collection of these rules is known as a *deployment descriptor* and is modelled on the deployment descriptors in the J2EE environment (Shannon, 2003). Once a deployment descriptor is created it is utilised by the MobJeX pre-processor, `objexc`. Although declaration of mobile capable classes was not implemented in the prototype, the basic unit of movement in PTM was expected to be a class. The deployment descriptors were also envisaged to address issues such as object location and co-locality as described in Perry's thesis. The approach taken by Perry is more transparent than the previously described declaration strategies, as it provides a means for declaration without additional code. Additionally this approach provides the application deployer with a greater degree of flexibility.

3.3 Runtime Environment

Each of the frameworks under review requires an executable runtime component to be deployed and run on each device in the distributed environment. This runtime component is responsible for receiving remote objects but may also have additional responsibilities, such as object tracking and metric gathering. In order for each application instance to be contained within a separate process, as required for complete termination, a separate runtime component may be required per application instance.

FarGo requires that all computers participating in the distributed system have a FarGo *core* running. Similarly, each host in the JavaParty environment runs a single *LocalJP*. LocalJPs are needed in order to facilitate object creation and migration. Unlike FarGo, each runtime component is registered with a central entity known as the *RuntimeManager*. In addition to knowing details about each of the connected LocalJPs, the RuntimeManager also knows the location for each mobile object. Information contained in the RuntimeManager is replicated among the LocalJPs to reduce the load on the central RuntimeManager (Philippsen and Zenger, 1997).

The Voyager runtime is called *voyager* and it can be started from either the console or from within the code. By having a programmatic interface to the runtime environment, Voyager ensures that applications can be distributed to end users and the appropriate runtime environment can be started transparently without user intervention.

PTM requires each application instance to be maintained within a single executable environment known as a *TransportManager*. Perry's thesis introduces the concept of a "Distribution Manager" which is responsible for overseeing the creation and management of individual TransportManagers on the local host as well as functioning as a distributed System Controller. Like Voyager, PTM provides a programmatic interface to its Runtime Environment. PTM uses a 'singleton' (Gamma et al., 2003) to ensure that there is always a runtime component running within the local VM, if one doesn't exist it is transparently created on access.

3.4 Object Creation

In all of the frameworks under review, it is necessary to modify the way in which mobile capable objects are instantiated.

FarGo, JavaParty and PTM all manage to overcome this challenge transparently to the developer by substituting appropriate instantiation code when the program is pre-compiled.

As a side affect of its simplified remote object declaration, Voyager requires that creation of mobile capable classes differ from the norm. Instead of the normal object instantiation like FarGo, new objects must be created via a "Factory" (Gamma et al., 2003). If an object were to be created in Voyager without using the factory, additional code required for mobility would not be inserted and hence the object could not be migrated.

3.5 Proxies

FarGo, JavaParty and PTM all utilise proxies (Shapiro, 1986) to separate mobile capable objects into a number of different classes, promoting cohesion. Instance members and method are bundled into an implementation object, while the code responsible for finding the appropriate instance and invoking methods is bundled into a proxy class.

There are no papers available for Voyager that document exact implementation details since it is a commercial product. Hence, no class diagram of mobile capable objects will be provided in this thesis, however it is stated in the documentation that proxies are utilised within the Voyager system.

As introduced in Section 3.2 during the development cycle each mobile capable class in FarGo, JavaParty and PTM is pre-processed by its respective framework. The pre-processing stage results in application code where references to object instances are modified to point to a proxy for that same instance. The proxy or *handle* that the application utilises acts as a “remote proxy” (Gamma et al., 2003) capable of tracking the implementation object, which may be located within any number of virtual machines on any number of physical computing nodes. The implementation object exposes a *Remote Procedure Call* (RPC) interface allowing it to be called remotely by the proxy. Furthermore, a proxy-based approach allows dynamic substitution of object implementations based on the capabilities of the hardware (Raverdy et al., 1998).

There are two common techniques for updating proxies, these are known as *forwarding addresses* (Fowler, 1986)(also known as *lazy tracking* (Acharya et al., 1997)) and *active proxying*. Forwarding addresses involves having only the runtime environment that previously stored the object know where the object has been migrated to. When a proxy needs to access an object that has migrated, a chain of forwarding addresses as described by Fowler (Fowler, 1986) is followed until the object is located. In order to make this approach more efficient and less susceptible to machine failure the reference chain can be shortened (Baude et al., 2000) so that the calling proxy has a direct reference once the object is found.

The alternative to forwarding addresses is active proxying which involves notifying each proxy to an object when that object has migrated. This introduces additional overhead as each object must additionally keep track of the location of it’s proxies.

When classes designated `Remote` are compiled in JavaParty, the class structure illustrated in Figure 1 is created (note that the “Static Parts” of Figure 1 are discussed in further detail in Section

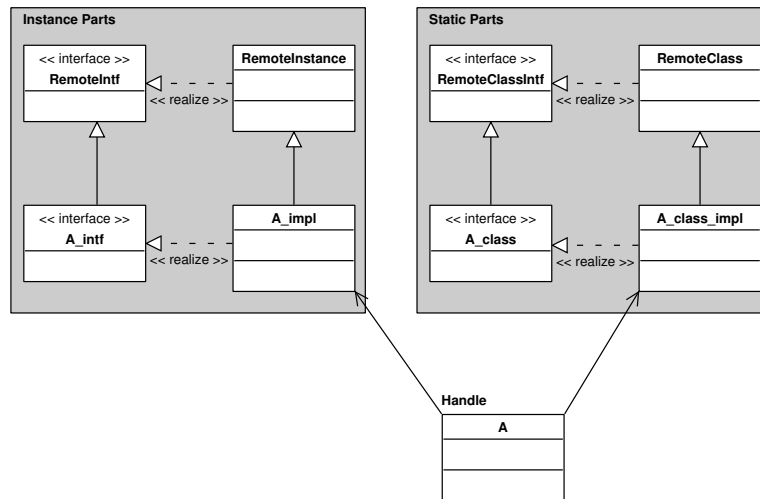


Figure 1: Proxy design for JavaParty

3.7). The handle referred to by the calling code implements the same interface as the original class and is responsible for calling the correct implementation. If an implementation is migrated to another host then it leaves behind a “proxy”, when methods are invoked on this object a `MovedException` is thrown back to the client containing a forwarder address. The handle then uses this information to update it’s internal reference and to call the method at the new location.

When classes are compiled for use in PTM the proxy structure illustrated in Figure 2 is generated. Perry’s Transport Manager used a combination of both *forwarding addresses* and *active proxying* for

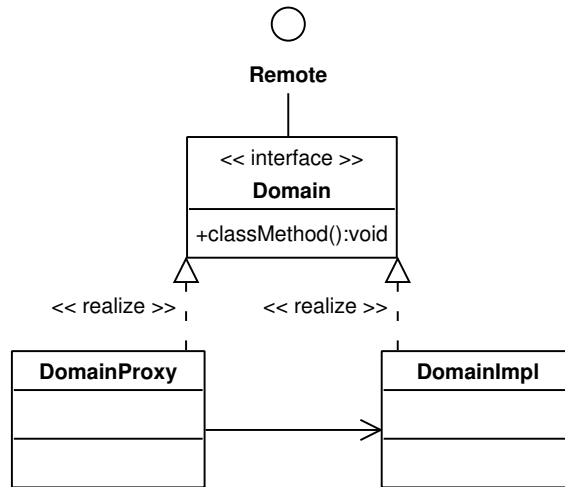


Figure 2: Proxy design for PTM

the purposes of testing and evaluation with the conclusion being that performance times were comparable for both approaches. Active proxying was deemed better for simple tests and structures, however for larger structures such as those present in full applications the cost of maintaining a large number of proxies would lead to prohibitive object migration times. For more comprehensive summary of that research see Perry (Perry, 2003).

In FarGo, there is additional overhead in tracking objects. While forwarder references are used to keep track of an objects new location, upon object migration, the FarGo system notifies the core where the object was initially created that the object has migrated to a new location. If that particular core was to become unavailable, then the entire system may become unstable. Each object is monitored by an associated tracker, which provides forwarding facilities. The relationship between objects and their trackers is illustrated in Figure 3, note that the `DataObject_` class contains the actual implementation of methods, while the code would refer to a `DataObject`. Figure 3 illustrates that there are actually two proxies in the FarGo system. Firstly calls are made on a local proxy object, the call is then passed to a tracker object, which exposes a remote interface (it is compiled with a RMI) and acts as a remote proxy, finding the implementation and invoking the method on it.

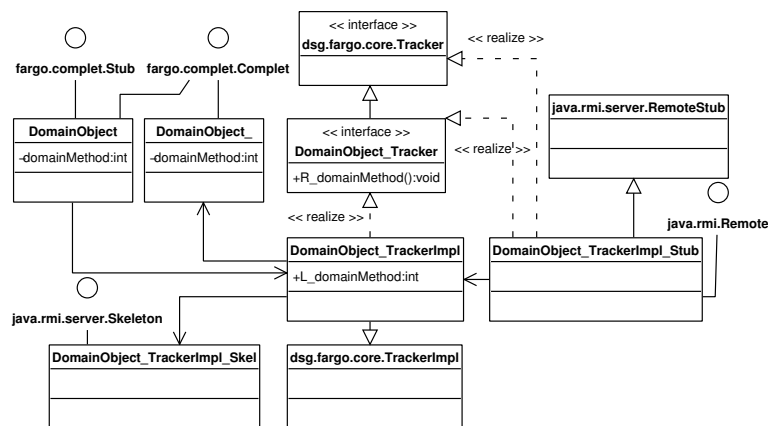


Figure 3: Proxy design for FarGo

An novel approach to proxy generation was developed by Baude, Caromel, Fabrice, Julien and Vayssière (Baude et al., 2000) who implemented the Proactive framework on top of the Java// (Java Parallel) library (Caromel et al., 1998), which enabled dynamic proxy generation using a combination of bytecode engineering and Java reflection. This approach means that classes no longer require pre-compilation or declaration and therefore can be substituted into running applications when required. This approach suffers performance disadvantages when compared to directly modifying the source of mobile object, as the transform is performed at runtime.

An implication of the design and structure of proxies is the ability to cater for inheritance relationships between classes. FarGo and JavaParty, allow for inheritance albeit with a limitation. When the source of all classes in the inherited relationship is available to the compiler, an inherited relationship between proxies is established by the pre-compiler, which allows classes to behave in the correct manner.

Since source code is not always readily available for commercial libraries or system classes containing native code, this represents a limitation in these systems. The proxy structure of PTM also presents the same limitations as FarGo and JavaParty and requires any inheritance relationships be hand coded.

As previously discussed, Voyager requires a Java interface specifying available method per mobile capable class. In order to gain access to inherited methods they must be explicitly stated in the interface for the object.

3.6 Managing Local and Remote References

In his thesis on the ‘Transport Manager’, Perry poses the question “Can all mobile objects be accessed as if they were local, regardless of their actual location within the system?” (Perry, 2003).

The first step to ensuring this is to make sure mobile objects have the same calling semantics as their mobile counterparts. Java normally has *call-by-reference* semantics, meaning a reference to the object is passed into methods. RMI on the other hand has a combination of call-by-reference and *call-by-copy* semantics, call-by-copy means a copy of the object is made and the method operates on that, with no effect on the original. In order to facilitate correct call-by-reference behaviour required for complete transparency, it is necessary to either pass proxies to objects as parameters, or alternatively return any values requiring modification along with any previous return values. Regardless of transparency concerns, it is important for application developers to know that the behaviour of mobile capable classes will be predictable regardless of location. Hence calling semantics must be documented and adhered to regardless of object status.

The disparity between local calling strategies and that of RMI does not appear to have been factored into the design of PTM. Applications do not behave the same way when methods are invoked on local objects as when they are called on objects that have migrated and become remote.

In the case of JavaParty, the local call ‘transport’ is responsible for the duplication of parameter and result values, in effect this gives local calls the calling semantics of remote procedure calls (Haumacher and Philippsen, 2001). Applications transformed for use in JavaParty will always use call-by-copy and can be guaranteed to behave consistently, nevertheless this comes at the cost of transparency. If the program has been used and tested in its original form, the modified version will not behave in the same manner.

In order to achieve partial transparency of calling semantics, FarGo passes complets by reference but passes regular objects by value (Holder et al., 1999b). If a regular object that contains reference to a complet is passed as a parameter, then the reference is not copied and hence also referred to by value.

Voyager offers complete remote reference transparency as it automatically creates a proxy for any object that is passed by reference (Wheeler, 2002). This approach offers transparency but at the cost of efficiency, since every remote method call results in the creation of a proxy for each parameter. If the method results in a number of discreet changes to the parameter, each change results in a remote call via the proxy.

An important performance aspect of managing local and remote references is how methods are invoked on local objects. The current versions of PTM, FarGo and JavaParty use Sun's Remote Method Invocation library (RMI) (Sun Microsystems, 2003) or a variant as the underlying remote communication mechanism. The design of RMI is such that one class referring to another via its `java.rmi.Remote` interface with both instances residing on the same machine, will result in the method invocation being passed though the network stack. Consequently invocation of an empty method is "about 4000 times slower" (Philippsen and Zenger, 1997) via RMI than via local reference for calls on the same device.

Prior to migration, the overhead of method invocation on two complets in the FarGo system that are local to each other is comparable to that of a local call. However, results from empirical tests (Section 6) conducted as part of a review into the performance of mobility frameworks described in this section, shows overhead comparable to that of a remote call when invoking methods on complets local to each other after migration, even when both complets are placed back on the original machine. Therefore it appears that local inter-complet connections are not re-established after complet migration.

Unlike FarGo, JavaParty "specifically addresses the locality problem: If objects reside locally, they are used locally". Although specifically stated as an advantage, local method invocation only occurs when it is clear that local invocation was intended (Haumacher., 2003). The conversion of remote calls into local ones occurs at compile time, when the location of the object at runtime cannot be asserted, hence there are instances when methods on a local object are still be invoked remotely.

While it is the case that proxies in PTM allows for mobile objects, "MobJeXObjects", to be accessed via a local reference when initially created, when migrated to a remote host and back, the local link is not resolved and the object is referred to via its remote interface.

Although there is no mention in the Voyager documentation as to whether local references are used when objects are located within the same VM, empirical tests (Section 6) show that the overhead of a local method invocation before and after migration away from and back to the same host, is comparable, showing that local references are resolved. This overhead is slightly higher than simple method invocation as discussed later in this thesis (Section 6), indicating that there is a slight performance penalty associated with the use of the voyager proxies and their runtime generation.

3.7 Static Methods

Java does not allow for the declaration or definition of static methods in interfaces, consequently static methods cannot be defined in Voyager classes that are intended for mobility. This interface behaviour also dictates that "RMI cannot be used for static methods and static variables" (Philippsen and Zenger, 1997). This presents a limitation for PTM, FarGo and JavaParty because as previously discussed they use RMI or a variant as their communication mechanism.

JavaParty is able to overcome the limitation by separating out the static methods into a non-mobile class. The instance methods and variables are then placed in a separate class, which is mobile. All proxy instances make calls to the same non-mobile class to resolve static methods, but they may refer to any instance of the mobile class for non-static methods. This arrangement has performance penalties, as each static call always results in a remote call and therefore is not very useful in practice.

Consequently, JavaParty is currently the only mobile object framework with any support for Static Methods.

3.8 Error Handling

In current mobile environments with existing wireless technology, transient disconnections are common (Chatzigiannakis and Nikolettseas, 2004). As a result it is important that any framework designed for use in such an environment provides mechanisms for error detection and recovery.

It appears to be the position of most of the mobile object frameworks under review that exceptions or errors simply do not occur or that if they do then the application code can deal with them. For instance, FarGo-DA, an extension to the FarGo system, which is explicitly designed for disconnected operation on mobile computers, states “fault tolerance for NLPCs¹ are an important topic ... but beyond the scope of this work ... we focus on voluntary disconnection”. Due to the complex nature of mobility subsystems most application code will not have the capability to correctly analyse and correct erroneous situations, for instance where network errors have prevented a remote call from proceeding correctly. Therefore the responsibility for error detection and correction particularly in regards to network errors belongs to the framework itself.

JavaParty has been explicitly designed for workstation clusters connected via high speed networks as the target environment, where network latencies and packet loss, which are the common causes of error in ad hoc networks, are reduced. Therefore when deployed in its target environment, JavaParty has no need to throw or deal with the exceptions thrown by the other frameworks.

There are three additional causes for exceptions in classes adapted for use in mobile object frameworks. These causes are object creation, migration, and invocation of methods on remote objects that are unavailable.

FarGo does not require special code to deal with the creation of objects. In contrast, both Voyager and PTM require special code in order to create objects. The Voyager `Factory` (see Section 3.4) throws `Exception`. Since all Java exceptions extend the `Exception` class, it is difficult to easily identify which exception has occurred and the correct cause of action to take (Bloch, 2004). Similarly, PTM requires that all calls to the Runtime Component catch an RMI specific exception.

In contrast, FarGo and Voyager throw specialised Exceptions such as `MovementException` and `MobilityException` when a migration error occurs. When synchronously requesting the framework to migrate an object, developers must explicitly catch these exceptions, despite there being little that can be done to correct the situation, and thus the entire application may fail. PTM does not throw any custom exceptions, instead it simply throws on the RMI specific `java.rmi.RemoteException`. Similarly to previous cases, a developer who is not proficient in the underlying mechanisms employed by the framework cannot correct this type of exception. Consequently, the overall transparency of the framework is reduced as migration errors will cause the program to be terminated.

Unlike the other frameworks the structure of the proxies in PTM is such that each method throws a `java.rmi.RemoteException`. As this is a checked exception, each call to the proxy must be wrapped in a try-catch block thereby reducing the transparency of the proxies and limiting their use to sections of the code that explicitly catch or throw this exception. In the pre-processed code proposed, each call is contained within a catch block consisting of a generic `printStackTrace()` call which prints the exception to the console.

Since the literature for mobile object frameworks has shown that there is a lack of a comprehensive error detection and correction model within these systems, it is necessary to look outside of these

¹networked lightweight portable computers

systems for approaches that can be adapted for use.

One use for mobile object frameworks is to provide system support for mobile agents. Mobile agents can be considered mobile capable classes that are aware of their environment and make decisions based on it. The JavaParty and Voyager frameworks have both introduced mechanisms such that mobile objects may act as agents via the execution of methods on arrival and departure from nodes. Similarly pure agent frameworks such as Agents (Izatt et al., 1999) have many of the features of mobile object frameworks such as the ability to “freely migrate objects to heterogeneous hosts” (Izatt et al., 1999).

Due to the similarities in functionality between the two environments it is worth considering the approaches taken by these frameworks. Strasser and Rothermel (Strasser and Rothermel, 2000) propose a method by which Mobile Agents can be rolled back. The rollback system relies on the agent save points being declared and at these specified times the steps the agent has taken is saved in an “agent rollback log”. This process is described further in Section 5.7.

4 Research Questions

The review of mobility systems in the previous section has identified a number of outstanding issues with regards to transparency, performance and reliability in mobility subsystems and thus the following questions are posed.

1. How can we improve transparency in a mobility subsystem ?

This question will be addressed by investigation into more transparent object declaration. Additionally, proxy creation, structure and calling semantics will be explored to discover areas where transparency can be improved upon over existing systems. Finally, the structure of the runtime environment will be investigated and transparency improvements within that area will be identified.

2. How can we improve performance in a mobility subsystem ?

More dynamic management of local and remote references has been identified as a key area for possible performance gains. Additionally, improved proxy creation and object migration algorithms are likely to yield performance gains.

3. How can we improve reliability in a mobility subsystem ?

This will be accomplished by exploring error correction strategies for ad hoc NGI networks. Furthermore, techniques providing error handling in more traditional environments will be investigated.

5 The MobJeX Mobility Subsystem

As the first step to addressing Research Questions 1 and 2, a review of the literature was undertaken and has been presented earlier in this paper. The literature identified and distinguished between mobility systems and adaptive mobile object frameworks that are at the minimum composed of both a mobility system and System Controller.

An adaptive mobile object framework currently under development within ATcrc² is MobJeX. The MobJeX framework is specifically aimed at NGI devices, hence it serves as a suitable platform for exploring the main goals of this thesis namely, transparency, performance and reliability as they fit directly within the MobJeX project scope. Given these facts, the MobJeX architecture was identified as a suitable platform for testing a revised mobility system.

In order to provide context to the mobility system presented in this thesis and described in this section, the architecture for MobJeX is briefly described below.

The MobJeX system architecture is illustrated in Figure 4, and described more completely in (Ryan and Perry, 2003) and (Ryan and Westhorpe, 2004). The mobility subsystem provides the ability to migrate objects between runtime components as previously discussed.

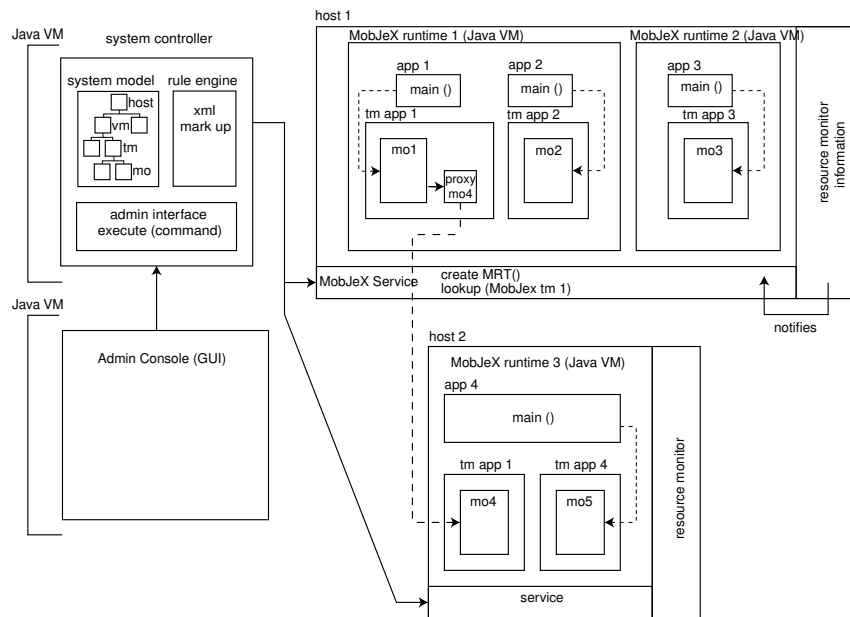


Figure 4: MobJeX System Architecture

The *System Controller* illustrated on the left hand side of Figure 4 provides a central point of control for the distributed environment. The System Controller maintains an updated model of all components within the system and evaluates rules specified as part of the deployment process against this model. When a rule (e.g. system memory is less than 50% free) is matched the System Controller evaluates how to re-arrange the model in order to best meet the criteria of the application. When an optimal solution has been evaluated it issues commands to the mobility subsystem directing the relocation of objects. The System Controller may itself be distributed and run on each host as originally proposed by Ryan and Perry (Ryan and Perry, 2003).

A graphical user interface (GUI) component for the MobJeX system has also been independently developed. The GUI incorporates the System Controller's model and presents a graphical representation of system state. The GUI issues requests for manual object migration by way of the System Controller which forwards requests to the mobility subsystem.

A revised mobility subsystem has been designed and implemented, and serves as the main contribution of this thesis by addressing many of the transparency drawbacks and limitations of previous

²Australian Telecommunications Cooperative Research Centre <http://www.telecommunications.crc.org.au>

systems. This new system has been designed from the ground up to address many of the issues facing ad hoc NGI networks. Further discussion of the solution is presented in this section.

The system described in this thesis has little resemblance to its predecessor PTM, as its architecture is peer-to-peer in nature and explicitly designed for ad hoc networks rather than the web-based architecture employed by Perry.

In order to answer Research Question 3, the characteristics of mobile ad hoc networks were investigated. Candidate solutions for dealing with the behaviour of these networks have been proposed and implemented, and are presented later in this section.

The system described in this section was subjected to a number of empirical tests showing quantitative and comparative performance, thus providing answers to Research Question 2. The results of these tests are presented in Section 6.

The concluding section of this thesis will identify areas of further research both in regards to the research questions and extension points for the MobJeX mobility subsystem.

5.1 System Architecture

The mobility subsystem architecture was modelled around the design used by the Java Virtual Machine and is composite in nature, with each host having a single *Service* encapsulating the functionality of the device. Each *Service* contains multiple *Runtimes*, each of which encapsulates a Virtual Machine or JVM. A *Transport Manager* encapsulates a single application and contains any number of application objects. Each *Runtime* contains as many *Transport Managers* as necessary to support the application instances that are currently running. Application objects within the *Transport Manager* are mobile capable in nature and are referred to as *Objects*. This architecture addresses a number of limitations presented in Section 3.3. Each host is capable of functioning as both client and server via the *Service* interface, causing the system to become more peer-to-peer in nature than the prototypical PTM (Perry, 2003), with objects able to migrate to between peer hosts. This arrangement better reflects the nature of ad hoc NGI networks, with devices able to send and receive objects upon arrival and departure from dynamic network groups.

The system has been implemented using purely Java and is capable of running on hardware-constrained devices. The following subsections describe the implementation used to address transparency, performance and reliability and thus Research Questions 1, 2 and 3.

5.1.1 Service

The *Service* has been introduced in order to combat the situation faced by FarGo (Holder et al., 1999a), Voyager (Recursion Software Inc, 2004) and PTM (Perry, 2003). In these systems each runtime component must be explicitly launched on separate ports, as each instance is locked to a specific TCP port. Applications using the framework must use unique port numbers for each running instance.

An alternative to running components on individual TCP ports is enforcing a single runtime component per device. This component must then be written in such a way that all objects are grouped together by application instance. This logical grouping enables removal of all object groups in the distributed environment upon notification of application termination.

Earlier systems fail to provide an application running on a fixed well-known port that can be queried for information regarding existing runtime components. JavaParty introduces such a concept with its *RuntimeManager*, responsible for overseeing all runtimes in the environment. When *LocalJPs* are launched they contact and register with the *RuntimeManager* in their network cluster.

The Service in the mobility subsystem is similar in responsibility to the RuntimeManager, however in order to enable the flexibility required for ad hoc networks, an instance is required per host rather than per cluster. The Service is bound to a single known port enabling a point of contact for other devices in the distributed environment, including but not limited to a System Controller or rule engine as required in adaptive mobile object frameworks. A Service provides a point for launching new Runtimes and contacting existing ones, in this way, it encapsulates all of the functionality available at a host level. In addition the Service exposes an interface for metric gathering, allowing the System Controller to make informed decisions based on environmental state. When Runtimes are launched the Service is responsible for the discovery and allocation of port numbers.

5.1.2 Runtime

A Runtime encapsulates a Java Virtual Machine. The Runtime is responsible for managing several Transport Managers in much the same way as a JVM is responsible for managing threads within a multi-threaded application. A pre-processed application only ever makes calls to the Runtime, which in turn is responsible for servicing requests or delegating them to the correct Transport Manager. If a call is made to a Runtime that does not exist yet, a new one is created transparently, enabling the users to have zero awareness of the framework. Hence, the system is more transparent to users in response to Research Question 1.

FarGo, JavaParty and PTM only allowed one instance of the runtime component per machine by default. The cause of this limitation is the fact RMI requires each service to be bound on separate TCP ports and in the case of PTM, the port value was specified in advance. The introduction of the Service enables Runtimes to be contacted regardless of port number, permitting the number of Runtimes to be limited only by free ports.

5.1.3 Transport Manager

The Transport Manager in the mobility subsystem is loosely based on the client-server Transport Manager developed by Perry (Perry, 2003).

While the client-server approach used in PTM does not directly affect application programmers, it places constraints on the functionality of the system. Objects could only be migrated one way from a server to an individual client, which has limitations when considering the framework for use in ad hoc networks (as mandated by the Research Questions in Section 4), where devices are constantly arriving and departing.

A Transport Manager encapsulates the portion of an application that is running within a Runtime. Thus two different instances of the same application running in the same Runtime will be contained within different Transport Managers.

The arrangement of Transport Managers and Runtimes provide a means for scaling the system. As more applications are required, more Transport Managers are automatically created. If one Runtime has reached a limit of Transport Managers because of memory or some other constraint being reached, then the Transport Manager itself can be migrated to a remote Runtime.

Due to the distributed nature of applications, it is vital that application objects are contained within a logical grouping such as provided by the Transport Manager. This grouping allows an application to be successfully garbage collected upon termination, by notification to all associated Transport Managers. The collection process is aided by RMI's distributed garbage collection routines. However, as each Transport Manager has both local and remote references to the Mobjects contained within, the RMI mechanisms cannot be employed without the awareness of the Transport Managers.

5.1.4 Mobject

A mobject is composed of the implementation of the original object along with some additional methods as detailed in Section 5.4. It is the mobject that actually migrates between Transport Managers.

5.2 Declaration

The three alternatives for mobile capable object declaration identified in the literature were dynamic, internal and external (Section 3.2).

Dynamic declaration as used by Java// (Java Parallel) (Caromel et al., 1998) and to a lesser extent Voyager, requires classes be declared mobile by creation syntax rather than in a prior declaration stage. This approach has four drawbacks that must be taken into consideration. Firstly the performance hit of constantly creating proxies for every object that requires mobility must be compared with compiling in support for all mobile classes. Secondly the creation syntax must be changed wherever mobile capable objects are required, to achieve this transparently, declaration is still required for classes which have intended mobility. Thirdly, it is reasonable to assume that not all classes are of a size or provide levels of functionality whereby migration will be beneficial. Compared with internal and external declaration, dynamic declaration requires no effort be invested in categorising which classes should be made mobile capable. Hence, objects whose migration could reduce functionality or performance of the application (Jing et al., 1999), may be made mobile capable. Finally, as mobile objects are explicitly created in the code, there is no opportunity to return non-mobile versions where appropriate without recompilation.

The techniques of internal declaration used by the JavaParty and FarGo frameworks aim for transparency, however they present a number of distinct disadvantages in terms of transparency over the external declaration proposed by Perry (Perry, 2003).

Firstly existing development tools such as Integrated Development Environments (IDE) cannot be fully utilised. The addition of non-standard keywords causes Java syntax recognition to fail, unless specific plug-ins are designed and installed preventing the problem. Consequently, during the development cycle, JavaParty's `Remote` keyword must be constantly inserted and removed to gain full benefit from the IDE. A similar situation applies to the declaration of classes in files with different names, which is the case in FarGo. The code will not be recognised as conforming to the Java specification and errors will be raised. In contrast, external declaration allows code to be written and debugged in an IDE. The code can then be compiled as either a standard application for testing or an application designed for use in the framework by simply using a different compiler.

External declaration also means that the roles of application developer and application tuner can be separated. The XML format for rule specification and evaluation proposed by Donald (Donald, 2004) will enable the tuning of an application by persons with no knowledge of Java but experienced in distributed or mobile computing. In addition it may be possible to automatically generate deployment descriptors for an application that are based on statistical analysis of application performance and mobility characteristics.

Keeping Research Question 1 and the goal of transparency in mind, the revised mobility subsystem incorporates the idea of external declaration. Objects are declared as mobile capable in XML deployment descriptors, an approach which could be built upon further by specifying additional object properties.

Work on the declaration of rules for mobility is beyond the scope of this thesis, but rather it is the concern of a different research project (Donald, 2004).

5.3 Object Creation

The literature review has shown that object creation syntax can remain unchanged from the perspective of the software developer by utilising a pre-processor, as used in FarGo, JavaParty and PTM.

The PTM provides object substitution at an early stage as all request for object creation are routed through the runtime component. Thus, when it is beneficial to do so, a local non-proxied object may be returned instead of a proxy, providing an increase in call performance.

This additional performance advantage has been implemented in the Runtime as part of the new system and reliance is placed upon the System Controller to make decisions as to the type of object to return. The choice could be based on static code analysis or options specified in the deployment descriptor. In order to facilitate this, at the stage of pre-compilation code that creates new objects is transformed so that it delegates responsibility to the Runtime. A sample of this code transformation is listed in Appendix A.

The combination of external declaration and the provision of a pre-compiler have provided a contribution to Research Question 1, as mobile objects be created transparently with only the class type being required. The framework then compiles in extra code as required to support mobility transparently, reducing the effort required by developers.

5.4 Proxies

The proxy structure for the revised mobility subsystem varies greatly in terms of its flexibility over its predecessors. Although it shares similarities to the PTM’s proxies previously illustrated in Figure 2, the notable difference in the new design is the introduction of an additional `DomainRemote` interface.

References to ‘domain’ in the following sections can be interpreted as the problem or class domain, the name of the original class is substituted upon pre-compilation. The new proxy design is illustrated in Figure 5.

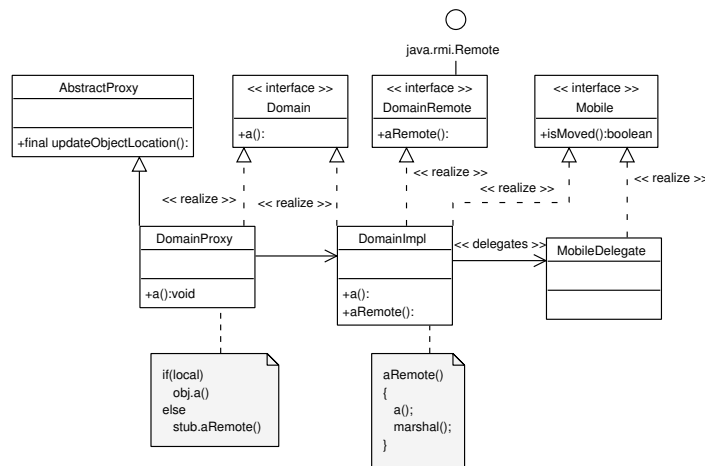


Figure 5: Proxy design for the proposed mobility subsystem

The following sections detail the responsibilities of each class and interface, particularly in contrast to the interfaces and classes in PTM.

5.4.1 Domain

The `Domain` interface represents the original interface of the class. In PTM this interface extended the `java.rmi.Remote` interface, which allows classes to have RMI stubs generated and hence methods can be invoked remotely. In order for the `rmic` tool to correctly create the RMI stubs, all methods in the interface must throw a `java.rmi.RemoteException`. As this is a checked exception it must be caught and dealt with, as discussed in Section 3.8.

In order to reduce the presence of checked exceptions and therefore improve developer transparency, the proposed mobility subsystem separates the two concerns of domain interface and remote method availability into separate interfaces. The `Domain` interface is a direct extraction of the existing interface of the class, with no additional extensions or modifications. This interface is used to ensure that the `DomainImpl` and the `DomainProxy` retain the same interface and can be used interchangeably when required by the creation process, allowing a local object to be used in place of a proxy when mobility is not desirable. The interface also ensures that anywhere where the original unprocessed classes is expected, the proxy will work in its place. As the proxy only implements the `Domain` interface, the code can be guaranteed not to throw any additional exceptions.

5.4.2 DomainRemote

The `DomainRemote` interface in the mobility subsystem performs the same function as the `Domain` interface in PTM. The new proxy structure uses a variation of *pass by-refvalue* (Dahm, 2000). When objects are passed as parameters to remote calls, they are placed into a `Hashtable` which is returned after the method has finished execution. To accommodate this change in method signatures, each method identified in the *Domain* interface has a suffix (e.g. `remote`) appended to it. Thus

```
public String getValue();
```

would become

```
public Hashtable getValueRemote();
```

This arrangement addresses the problem identified in Section 3.5 where many RPC mechanisms (e.g. RMI) imposes a mixture of *call-by-copy* and *call-by-reference*. This new arrangement increases transparency to the developer as all calls behave with the same *call-by-reference* semantics as pure Java.

Note however that this approach can only be guaranteed to work with simple threading models, since more complex concurrent threading models would cause this approach to fail when parameters are concurrently modified in overlapping calls.

5.4.3 DomainImpl

The `DomainImpl` class contains the original ‘business logic’ of the class. This is the object that the mobility subsystem allows to migrate between devices. This class realizes both the `Domain` and `DomainRemote` interfaces. The realization of the `Domain` interface is completed using code from the original class. To realize the `DomainRemote` interface the `DomainImpl` class delegates all calls from the remote method to the local method and places both the parameters and return values into a hashtable that is returned to the proxy. A sample remote method for the local method `public String getValue(ArrayList a);` is listed in figure 6.

```
public Hashtable getValueRemote(ArrayList a)
{
    Hashtable h = new Hashtable();
    h.put("a", a);
    h.put("returnvalue", getValue(a));
    return h;
}
```

Figure 6: Pass By Refcopy

The `DomainImpl` also contains the additional methods required for implementation of the `Mobile` interface, however all of those calls are delegated to a `MobileDelegate` instance. The `Mobile` interface is required to ensure synchronisation, as required by weak mobility during movement, and to enable proxy reference counting for distributed garbage collection. The synchronisation mechanisms are required to ensure that classes are not migrated during execution of a method which could cause inconsistencies in the object. When new proxies are created they register a unique identifier with the implementation. When no longer referenced, the proxy is finalised and notifies the implementation they are proxying. When the proxy count is reduced to zero, the implementation notifies the Transport Manager currently referencing it that the application no longer requires the instance, the Transport Manager then relinquishes all references to it. As no other classes hold references to the object, it is collected by standard Java garbage collection routines.

5.4.4 DomainProxy

The `DomainProxy` is a “remote proxy” (Gamma et al., 2003). The proxy realizes the `Domain` interface and as such can be substituted for the original class. The proxy is responsible for locating and calling methods on the implementation, which may be on a local or remote machine. The behaviour for finding implementations when they have migrated is inherited from a common base class `AbstractProxy` which follows a chain of “forwarding addresses” (Fowler, 1986) left at Transport Managers until the instance is found.

A solution for the limitation of PTM’s proxy design presented in Section 3.5 whereby inheritance was not supported, was found by revising the structure of inherited proxies, whilst still maintaining transparency, thus distinguishing this solution from previous ones of FarGo, JavaParty, Voyager and PTM. Figure 7 shows the relationship between proxies of inherited classes. Both `ObjectA` and `ObjectB` in the diagram have supporting classes (according to Figure 5) that are not shown in the diagram. Proxies compiled with this arrangement negate the need for specialised code to handle inheritance and other language features such as polymorphism, as it is the JVM that resolves relationships and calls correct methods.

Each proxy contains a copy of the unique identifier assigned to the implementation it is proxying, a remote stub to the Transport Manager where the object resides and an remote stub to the object itself. If the object is local the proxy also contains a local reference, the mechanics of this mechanism are discussed further in Section 5.5.

In contrast to the frameworks discussed in the literature review, the proposed mobility subsystem ensures mobile objects are *always* referred to via proxy, even `this` references. When the code is pre-processed, all references to `this` that leave the scope of the method such as return values or parameters to methods and constructors, result in the creation of a proxy for the `this` instance.

The `Domain` interface ensures all the same methods are available in the mobile object as the

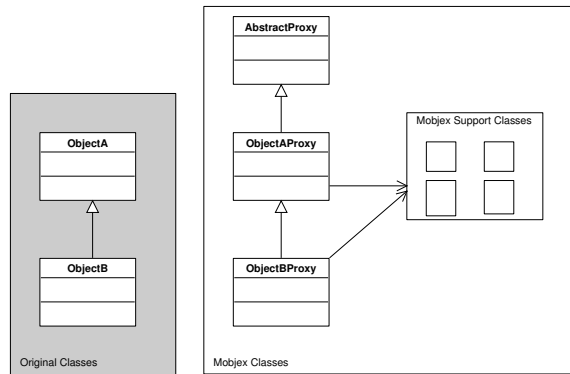


Figure 7: Proxy Inheritance

original and causes no additional exceptions to be thrown. Combined with the pass-by-refvalue implementation of the `DomainRemote` interface, this further addresses Research Question 1, as mobile objects can now be used transparently in all of the same locations as their original counterparts with no modification.

5.5 Managing Local and Remote References

Use of local references where possible, regardless of class type, was one of the key performance optimisations identified for implementation in the mobility subsystem under discussion. The implementation of local and remote references is also required to provide solutions for Research Question 2.

Invoking methods via a remote interface, introduces significant overhead as discussed in Section 3.6. For this reason, the `DomainProxy` maintains a local reference to its `DomainImpl` whenever possible.

The approach implemented is illustrated in Figure 5. When updating references to recently migrated objects, the proxy queries the local Transport Manager for the application to see if the mobject is located locally. If a reference is available locally it is obtained and a boolean flag indicating this is set. When this value is true, calls to the proxy result in delegation to the local reference, until either the proxy or mobject are migrated. Thus, it is the proxy that is responsible for calling methods on the local reference whenever possible, in preference to the slower remote interface. Empirical tests highlighting the benefits of this approach are presented in Section 6.3.

5.6 Static Methods

Transparent use of static methods is one area of mobile object frameworks that is yet to be adequately addressed. Currently of all the frameworks reviewed, including the proposed mobility subsystem, JavaParty (Philippsen and Zenger, 1997) is the only one with any support for static methods in mobile objects. The issues with the approach taken by JavaParty are described below.

Firstly, when objects are created on a local machine, then migrated, additional remote calls are required to contact the creating machine's `RemoteClass` which contains the remotely accessible static methods.

Secondly, in a mobile object framework, the use of static methods and instance variables should be discouraged, as they impose a more procedural structure to applications. Applications incorpo-

rating design patterns such as the “Singleton” or “Factory” (Gamma et al., 2003) which make use of static methods may run with significantly lower performance in a distributed environment, due to the constant calls between proxy and the static methods located in the non-mobile class. Due to the non-mobile nature of the class, the benefits made by moving the class ‘instance’ may be mitigated.

Finally, the approach used in JavaParty means that an application is permanently tied to the host. In ad hoc networks, it is possible that a node may want to leave the network and not return. In such a case it should be possible for all object instances to be relocated onto remaining hosts so the application may remain operational.

Support for immutable static objects could be provided if the mechanism was modified such that the static component was replicated among all devices with an instance of the class. If this approach was used for mutable objects however, it leads to problems of synchronisation, concurrency and network performance. The object could potentially be made mobile, however this approach requires further research on the impact to existing code.

All of the approaches listed above have issues in regards to transparency. Java does not allow for the declaration of static methods in interfaces, therefore it is not possible to declare two classes as having the same static method in a generic manner without using inheritance. Hence, to support static methods, all static method calls on the original object would need to be pre-compiled such that they call the proxy for that object. Additionally, static methods cannot be exposed via the remote interface, hence the static component would need to be regular object.

It has been shown previously in Section 5.4 that it is possible to migrate objects using standard languages features of polymorphism and inheritance. Therefore, in regards to Research Question 1, there are limits to which types of objects can be made mobile capable. Objects using static methods are unable to be migrated without significant performance overhead using the current methods developed in JavaParty.

5.7 Error Handling

The two interface proxy model discussed in 5.4 enables the first improvements in regards to reliability as required to answer Research Question 3. As the `DomainProxy` no longer exposes a remote interface, exceptions required for the RPC mechanism (for example RMI specific `RemoteException`) can be caught internal to the proxy, improving transparency.

The introduction of the `MobjexRuntime` and its ability to be referenced locally has resulted in try-catch expressions being no longer being required by the modified creation code, enabling better scoping for variables.

Exceptions thrown during migration are also no longer caught by the client and are instead handled closer to the cause by the Transport Manager or Runtime.

As covered in Section 3.8, a fundamental part of any mobile object system destined for use in ad hoc environments is error correction. The following error strategies are proposed as a means for answering Research Question 3 without compromising the transparency gains introduced in response to Research Question 1.

Error handling in the mobility subsystem described in this thesis has been implemented as a “chain of responsibility” (Gamma et al., 2003). Errors are sent along a chain of ‘error handlers’, giving each handler an opportunity to rectify the situation. Each component of the system (as described in 5.1) is an error handler and as such is given an opportunity to handle the error. The end of the chain goes to the user, but only when all other handlers are unable to correct the error. The following sections details the possible actions taken by each component of the mobility subsystem and are covered in order of the default chain.

5.7.1 Proxy

The simplest solution for many network failures is simply to try again. For this reason, the proxy's strategy is to maintain a count of failed accesses and continue to call the method until such time as the threshold is met. Frequently occurring transient failures result in an error notification being propagated to the Service and System Controller level, where it can be decided whether relocation of the object is an option. A complementary approach to this is to 'back off' for a random period of time and then attempt the communication again. If the object cannot be reached after repeated attempts, (indicating a permanent network failure), it becomes the Transport Manager's responsibility to attempt to rectify the situation.

5.7.2 Transport Manager

If a remote object can't be contacted after a period of time, the framework must take steps to establish communication with a replacement object, so the client application can proceed.

A replacement object that closely resembles the original object will cause the least impact on the running application. Using check-pointing and rollback mechanisms it is possible to get a copy of the original object, albeit in an earlier state.

There are many situations where a rollback mechanism could be exploited. Given the case of object migration an 'all or nothing' approach can be implemented whereby it can be assured that object migration will either work correctly or the object can be reconstructed on the originating host.

If a node was to crash after receiving an object and the object was not accessed at that location, a roll back to a previous state would result in no information loss. A rollback mechanism would also allow nodes to crash, yet be able to reconstruct all objects when re-activated.

Without the benefit of extensive code analysis it would be difficult to implement the method of partial rollback as proposed by Strasser and Rothermel and discussed in Section 3.8. Each object needs to have defined points where it is safe to rollback. Alternatives to defining checkpoints manually include timed saves and saves on beginning and exit of each method in the class. Timed saves would need to be calibrated carefully as it is possible that the object could be saved while in an inconstant state, resulting in incorrect data when reconstructed. On the other hand persisting the object on every method call could result in unnecessary I/O, thus reducing system performance.

The other issue that needs to be taken into consideration is how the internal data is saved. Each of the classes in the MobJeX system is `Serializable` however using the default mechanism for class serialization and logging the results would create a significant performance bottleneck.

As the Transport Manager is able to directly manipulate the proxy and its internal references it would be the ideal place to put the mechanisms for object check pointing.

If the Transport Manager is unable to create a replacement object, responsibility falls to the Runtime.

5.7.3 Runtime

Given the expenses associated with object check-pointing and rollback mechanisms it may be cheaper to simply create a new object. This approach is advisable if the class is immutable or has no instance variables of importance, as any data contained within the object will be lost.

The external declaration discussed in 5.2 could be revised to include declaration of whether objects should be check-pointed or replaced.

Hence the strategy of the Runtime is to create a new object of the same type and set the proxies internal reference to it, such that subsequent invocations will be on the new object.

Figure 8 illustrates the sequence for handling communication errors for a replaceable object.

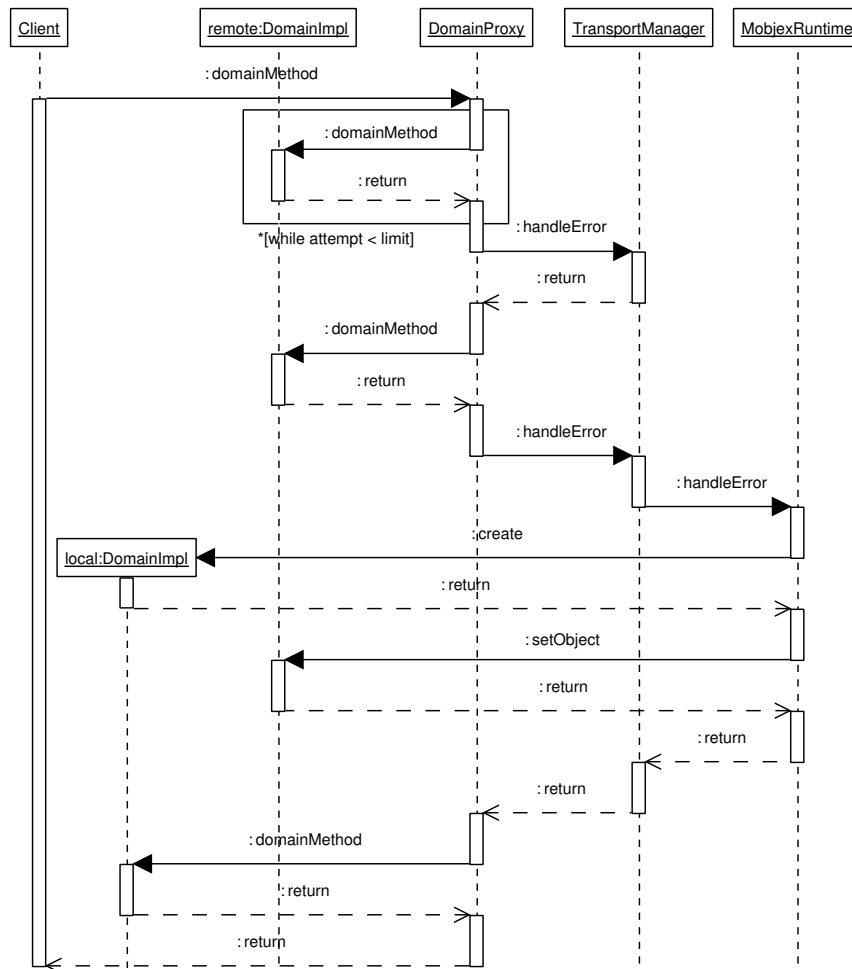


Figure 8: Error Handling Sequence Diagram

5.7.4 Service

Repeated failure to contact a remote object is indicative of a prolonged node or network failure. Every such failure should be reported to the Service, which can then deem a given remote Service unreachable. If a remote Service is declared unreachable then object are relocated from that host to an alternate host wherever possible.

Consider the scenario where a mobile computer and a desktop have a have a shared distributed environment with an application running. If the mobile device begins to leave the environment with some object still on a fixed PC, the network signal degradation will cause the Services to declare each other unreachable. The decision of whether to move all objects to the PC or the Mobile computer is an arbitrary one, for this reason, this approach requires one Service or Runtime to be designated in control of the application. The decision as to which device is in control should be deferred to the application rules or deployment descriptor, however one implementation is to have an object declared as being the *head* object, with all objects migrating towards it as required. The head object could be

considered to be the user interface component.

5.7.5 System Controller

If the previous strategies fail or are unavailable for the given object, then the responsibility falls upon the System Controller to find and detect the error. As the System Controller has a model of each of the Runtimes it can make decisions to move a whole group of objects such as a complete Transport Manager or Runtime to a new location.

In effect this would enable the isolation of poorly connected nodes, as no Mobjects would reside on them. In the situation where a large network splits into multiple smaller networks, application objects can be re-arranged to better reflect the network topology. Algorithms for network partitioning are beyond the scope of this thesis and are an area for future work.

5.7.6 Summary

In summary, modifications made to the architecture has meant client code is no longer required to catch or throw any additional exceptions and infrastructure issues have been relegated to the framework.

Error handling in the mobility subsystem has been implemented as a chain of error handlers as illustrated in Figure 8, in order to address Research Question 3. At each stage in the system there are appropriate actions that can be taken in order to correct errors resulting from network or hardware failure while using the framework.

Depending on the configuration of the object, attempts may be made to reconstruct it from an earlier saved version. However this requires that backup copies be made and stored in persistent storage using resources to do so. The alternative is to create a new object of the same type, although if the object is reconstructed from scratch then all instance variables will be lost. Hence in the process of application deployment it is important to carefully consider which of the strategies can be applied to each class of object.

Repeated failure to contact objects at a given nodes is an indication of node failure and should result in objects being moved away from that CE.

6 Evaluation

In order to evaluate the performance of the proposed system, an implementation was created and a number of empirical tests were developed and executed.

The first test provides comparative performance benchmarks between a number of mobile object frameworks for object migration in order to provide quantitative validation of the benefits stated in section 5 and to provide an answer for Research Question 2. The test provides quantitative times for the relocation of differently sized objects when performed on the different frameworks.

The second test creates a series of links in the manner of a linked list and then migrates selected links to a remote machine in a staggered pattern. This test provides measures of migration and reference updating speeds in order to provide further details to Research Question 2.

To answer elements of all Research Questions and to assess the fitness for ad hoc networks of the newly implemented mobility system, each test has been repeated over wired and wireless networks, and then again over a wireless network using a hardware-constrained device.

6.1 Limitations

Most of the tests were executed using two Pentium 4 2.8 GHz with 1,024 MB of RAM, running Java 1.4.2_04. The latest version of FarGo (0.2.2) is distributed in a compiled form for which Java 1.1.8 is the latest version capable of running it. Hence the FarGo tests were executed on the same PCs but with the older version of the JVM.

Tests run under Java 1.4.2_04 utilised a native timing library enabling more accurate statistics than the JVM was able to provide. In the case of FarGo, less accurate system timing was used in order to achieve JVM compatibility. This older library provided 1ms resolution as long as total elapsed time was greater than a 15ms threshold. Each test run on the desktop machines was executed with a heap size of 384MB for constancy and due to the large numbers of objects being created.

Each of the tests uses template methods to ensure that the same segments of code are tested and timed in each of the respective frameworks.

Perry's Transport Manager was not included in the review as it's prototypical state, which served as starting point for research in the thesis was non-stable. Its major limitation was an inability to correctly migrate objects back when that object's proxy had been migrated to a remote host.

6.2 Remote Get/Set Test

In order to provide a performance comparison between the frameworks identified in the literature the first empirical test was devised. This test was intended to show up relative strengths and weaknesses of each framework.

Firstly, a simple Java object that exposed a single get and set method, was modified to meet the mobile capable requirements of each framework (Sections 3.2 and 5.2). Objects were initially created on the local machine, then moved to a remote host and finally moved back to the original host. In each location the objects have a 'set' and 'get' operation applied to them in order to assert the impact of the local referencing discussed in Section 5.5 on performance. Calls were made using objects with serialized sizes of ≈ 100 bytes (104), 1KB, 10KB and 100KB.

The first call to an object in a new location is recorded separately in order to give approximate overheads for the reference updating strategy used in the framework.

In order to provide relative migration times for different sized mobile capable objects the tests were repeated for objects padded with values making the original Java objects have a serialized footprint of ≈ 100 bytes (104), 1KB, 10KB and 100KB respectively.

The tests were run for ten, one hundred and one thousand objects using a cross over network cable providing a consistent speed of 100 Mbps. The tests suite was then repeated over an 802.11b wireless link, providing speeds of up to 11 Mbps. Complete results for the tests are presented in Appendix B.

Initial analysis of the results showed that time taken for all frameworks was proportional to the size of the parameter and the number of objects. Therefore, the one hundred object test, which all frameworks were able to complete is used for discussion. It is a reasonable assumption that the larger the object parameter to a remote call the longer the call will take due to network overhead. To minimise the impact of parameter transportation, only the smallest object size is discussed.

Figures 9 and 10 present a graphical comparison of the test results for one hundred objects using the 104 byte parameter values over the wired and wireless networks respectively.

JavaParty is provided in two forms, a version that utilises the KaRMI (Philippsen et al., 2000) RMI replacement and a version using Sun's RMI implementation, however both versions use the same algorithms internally.

Optimising Transparency, Performance and Reliability in a Mobility Subsystem

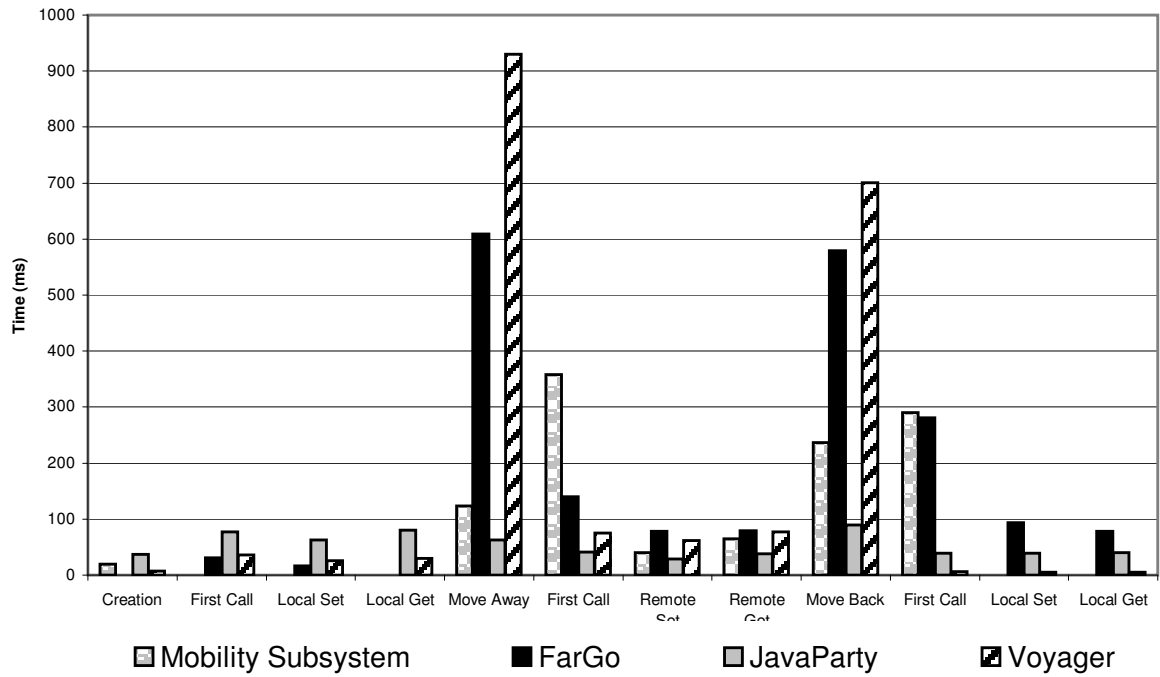


Figure 9: Remote Get/Set Test - Wired

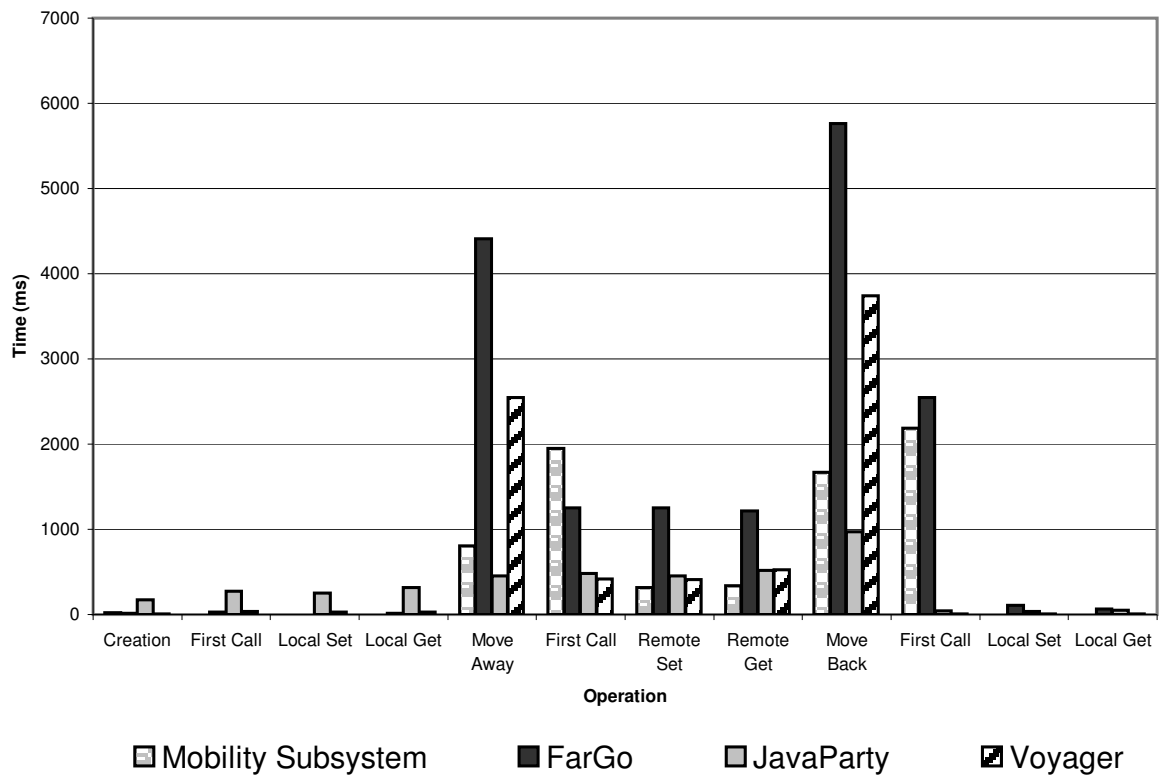


Figure 10: Remote Get/Set Test - Wireless

As RMI is available as a standard library on most JVMs and provides good portability between different versions of the JVM, it was chosen as the RPC mechanism for the initial implementation of the mobility subsystem in this thesis.

The KaRMI library is designed as a more efficient implementation of RMI and provides significant performance gains as demonstrated by the two graphs in Figure 11, which show a comparison of total times for ten objects running the first test over wired and wireless networks.

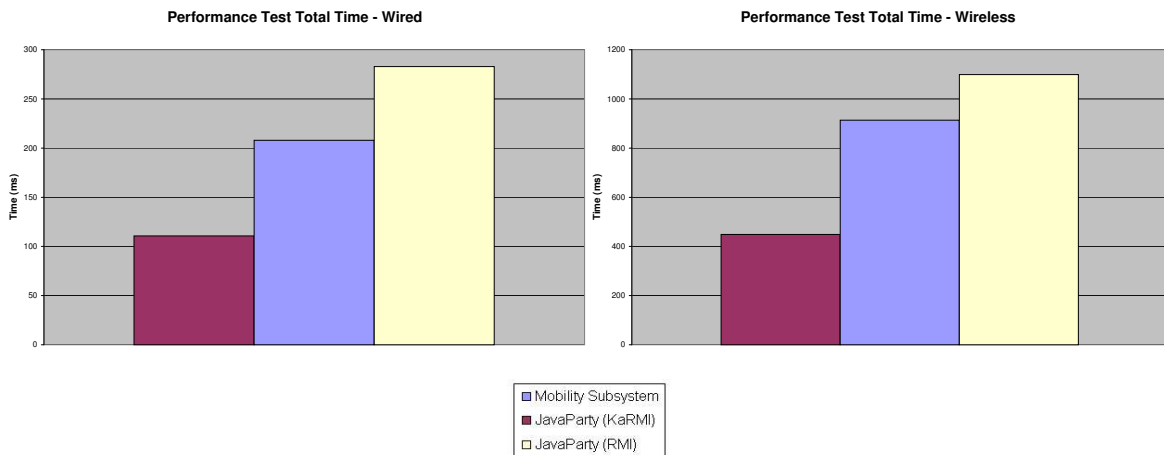


Figure 11: Relative times for Mobility Subsystem, JavaParty (KaRMI) and JavaParty (RMI)

The KaRMI version of JavaParty has demonstrated significant performance increases over both the RMI version of JavaParty and the mobility subsystem discussed in this thesis. This performance difference shown is regardless of network type, with total time of wired and wireless being proportionate to network speed (illustrated by Figure 11). The same algorithms are used in both versions of JavaParty, hence the higher test times for the RMI version of JavaParty over the mobility subsystem indicate a heavy reliance on network calls, which are optimised out in the KaRMI version. Performance gains could be realized in the mobility subsystem by refactoring to incorporate KaRMI or similar systems such as Java Wireless RMI (Campadello et al., 2000) however this may result in a reduction of portability and would need to be tested in heterogeneous ad hoc environments.

The RMI version of JavaParty appears to be less robust than its KaRMI counterpart and was unable to run tests when the number of objects reached one hundred. For this reason, the KaRMI version is used in further comparisons. In a similar fashion to the mobility subsystem in this thesis, FarGo only exists in a single version, which also uses Sun's RMI implementation. Voyager uses a proprietary remote procedure call mechanism for which details are unavailable due to its proprietary status.

Despite several attempts, FarGo was able to execute the one thousand object test over the crossover link but failed attempts on the wireless network. Due to various framework level exceptions, Voyager and JavaParty were unable to complete the tests for one thousand objects on either the wired or wireless links.

For the mobility subsystem, times for the 'move away' operation were less than those of the 'move back' operation. As previously mentioned in Section 5, the system is designed for use with a System Controller capable of direct contact with all Runtimes in the system. As the System Controller is currently under development, migration back to the local device required code within the test call methods on the remote Runtime. This arrangement, which resulted in additional remote calls, would not be present in a complete system, leaving both operations with identical time values.

The focus of the optimisation in the mobility subsystem was on reducing call overhead for local calls. Figure 9 demonstrates that the call overhead for objects before and after migration was significantly less in the mobility subsystem than in the other frameworks. Total time for all calls, both local and remote was 105 ms compared with 212 ms, 287 ms and 376 ms for Voyager, JavaParty and FarGo respectively. The significant time delay present in Voyager and JavaParty may be as a result of their proxy designs, whereby implementations are always referred to via remote interface. The call durations for FarGo increases significantly after migration, indicating that while initial complet references may be local, new local complet references are not resolved when available.

The benefit of the active proxying approach taken by FarGo and Voyager is exhibited in the form of faster first access times, which in both the wired and wireless scenarios was only slightly higher than subsequent invocations. The failing of this approach is also visible, both FarGo and Voyager exhibited greater movement times compared to JavaParty and the mobility subsystem even when combining migration and first call times. FarGo took the greatest time over all, although this may be a result of its dated virtual machine environment and RMI libraries.

FarGo fared particularly poorly in the wireless test, with times increasing more than the other frameworks for the 100 byte object. The total time taken was more than 830% greater than the crossover values compared with less than 650% exhibited by the other frameworks. These characteristics indicate a heavy reliance on the network.

Reducing network traffic is an important issue in mobile networks as the bandwidth is reduced as devices proceed further from access points. Voyager exhibits the opposite affect, with time taken for 100 byte objects being only 396% greater, more than 200% less than the mobility subsystem and JavaParty. However Voyager had the slowest total time for the wireless tests and the second slowest time for the wired test.

While Voyager call times are only slightly higher on average than JavaParty and the mobility subsystem, migration times were significantly slower than both these systems, increasing the overall time taken to run the test. In the context of real applications, multiple quick calls to an object per location would offset this cost of a longer migration. FarGo on the other hand demonstrated slower call times than all of the other frameworks as well the slowest migration times.

Due to the custom serialization routines present in its KaRMI layer, JavaParty exhibited the fastest object movement and call times remotely. The mobility subsystem displayed faster migration times than Voyager and FarGo, but slower than JavaParty, the cause of which is believed to be the combination of RMI and default serialization used, as discussed earlier. Local call times in the mobility subsystem were the fastest of all the frameworks.

6.3 Remote Stagger Test

To provide a comparison of object mobility times and further answer Research Question 2, a second test has been devised.

A mobile capable linked list object with recursive traversal ability has been developed for each framework. Linked lists of 10, 100 and 1000 nodes were created on the local machine. The entire list was traversed beginning at the first node. Every second node was then migrated (staggered) to the remote host and the traversal repeated. Next, the remaining nodes were migrated to the remote host and again the list was traversed. The above process was then repeated in reverse with all nodes eventually being migrated back to the original host.

Figures 12 and 13 present a graphical comparison of the test results for a linked list of one hundred objects over the wired and wireless networks respectively.

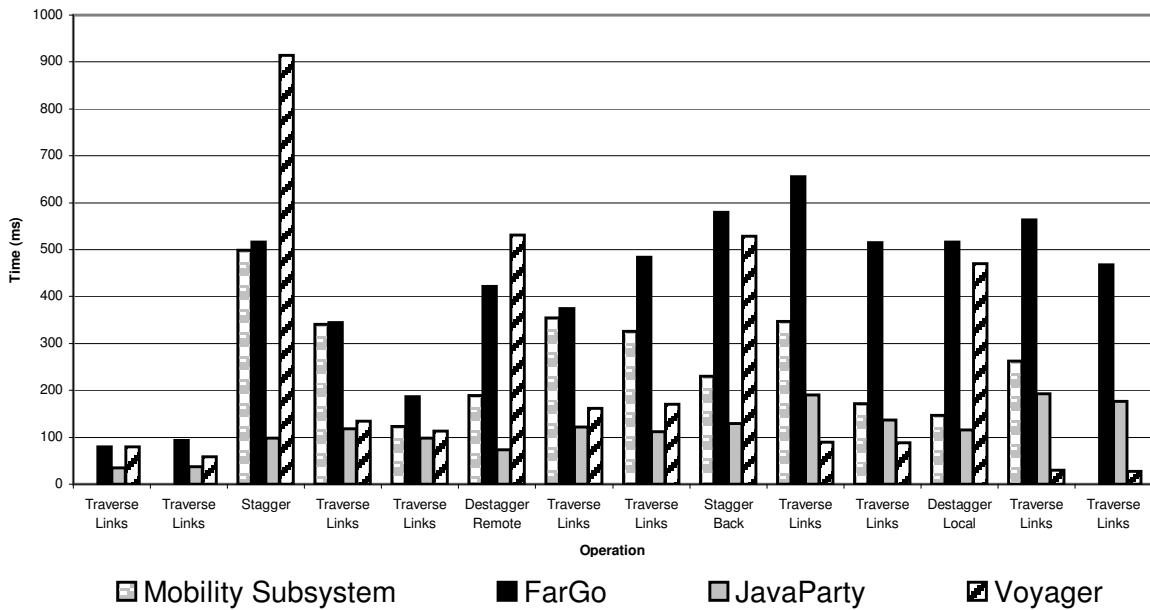


Figure 12: Remote Stagger Test - Wired

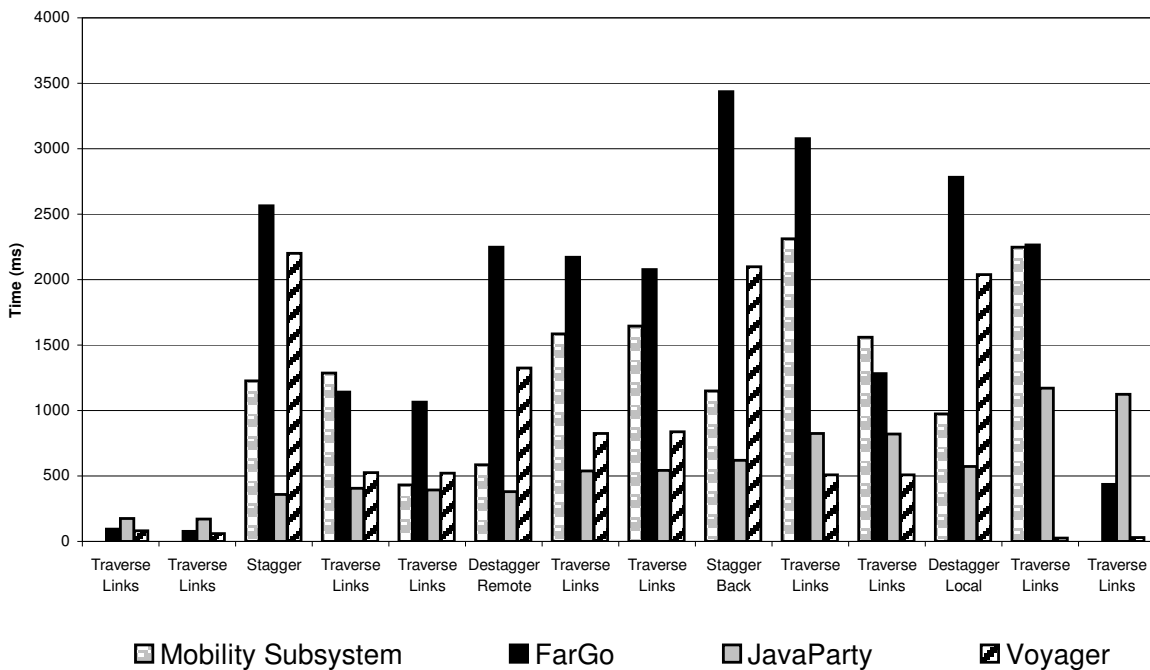


Figure 13: Remote Stagger Test - Wireless

This test has provided clear outcomes in the context of Research Question 2, with significant performance advantages being realised by the local object references, both before migration and upon return. The benefits are not so significantly demonstrated with all objects located on the remote host, however this is believed to be due to RMI inefficiencies as discussed in the previous section. Both FarGo and the mobility subsystem use RMI and exhibit similar performance characteristics, which

are not present in Voyager or JavaParty.

This test demonstrates the mobility subsystem providing better migration performance compared to both FarGo and Voyager. The increased performance arises from use of forwarder references and an optimised reference updating strategy as opposed to the active proxy approaches implemented in FarGo and Voyager.

This test once again highlights the more efficient RPC mechanism present in JavaParty with call time being lower than the other frameworks in the wired network environment. In the wireless environment the traversal times for JavaParty were more than proportionately higher than the wired network times, again demonstrating the reliance of the design on high speed networks and thus is not as suitable for ad hoc networks as the mobility subsystem presented in this thesis.

6.4 NGI Device Test

The previous two tests have demonstrated that the mobility subsystem works with efficiency on both wired and wireless networks. In order to show that the system can provide benefits in ad hoc networks and on hardware-constrained devices, the tests were repeated using one desktop machine and a 400MHz Dell Axim PDA.

Attempts were made to get each of the mobile object frameworks running successfully on the PDA using IBM's J9 Java Virtual Machine (version 5.6). Only the mobility subsystem under discussion could be deployed in this environment, although it should be noted that it is also the only one designed explicitly for this purpose. Voyager was provided in the form of an executable that could not be run on Pocket PC, due to reliance on network features not available in the Pocket PC environment, it was not possible to get FarGo or JavaParty running either.

The tests were run and completed successfully, however in the interests of brevity and with no point of comparison, complete results have been omitted. Nevertheless, the tests showed that ten 100KB objects could be migrated in 13966 ms. This translates to $\simeq 1.4$ seconds per 100KB object. While the test objects were Java objects that simply contained arrays to pad out object size, more useful objects could be represented by a 100KB Java object. Examples of objects this size are a user interface component for a wide range of systems or classes with large lists or arrays of values such as an address book, and many smaller productivity applications. Objects with a smaller serialized footprint, that may be usefully migrated to a NGI device such as a PDA include XML transformers (Perry, 2003) or single user interface dialogs.

6.5 Error Handling Test

As discussed in Section 5.7, the revised mobility subsystem has been implemented in such a way that errors are propagated through a chain of error handlers until the error is appropriately dealt with. In order to test the feasibility of this approach, a test was devised whereby an immutable object was migrated to a remote host, before the network link was terminated and a call made to the object.

Only the mobility subsystem discussed in this thesis was able to correctly handle this erroneous situation. When the network connection was re-established before the limit specified in the proxy was reached (three tries in the implementation under investigation), the method was successfully invoked on the remote object. Hence a transient network in an ad hoc environment can be dealt with by repeated connection attempts. The other systems threw exceptions to the console and terminated. If the network was permanently disconnected, the error was propagated up to the Runtime, which was then able to correctly create a new instance of the class and set the proxy reference to it. The proxy then returned the result of the method invocation against this new object.

This test has shown that the error strategies implemented as part of Section 5.7 have provided increased reliability and successfully addressed Research Question 3 in a transparent manner.

7 Conclusion and Future Work

This thesis has presented a mobility subsystem, designed for integration within the MobJeX architecture. It provides improvements over existing systems by allowing more transparent object migration between hosts in ad hoc NGI networks. Removing framework level error handling from client code has increased transparency to developers, whilst enabling the system to run on as any machine with a Java VM has increased transparency to both users and the system.

The system described herein has provided performance improvements that can be utilised by frameworks designed for use both inside and outside the NGI arena.

By dealing with transient network failures within the mobility subsystem, reliability has been increased, providing a more robust environment for mobile application developers and users.

Areas of future work for this system include the investigation of alternate remote procedure languages, for example SOAP or XML-RPC, that may provide increased portability by enabling applications to cross programming language domains and well as performance gains. Potentially this could allow Java applications to run within .NET virtual machines.

Potential performance gains to be investigated include transparently integrating asynchronous calls and 'futures' such as implemented in Proactive (INRIA, 2004).

The mobility subsystem presented in this thesis has been included into the current prototype of MobJeX and has transparently facilitated early load balancing tests based on automated rule evaluation.

Acknowledgements

I would like to thank my supervisor, Dr. Caspar Ryan for the invaluable assistance provided during the development of this thesis. I would also like to acknowledge the help and support of fellow honours student, Lachlan Donald throughout the year. Thanks also to Adam Boas for his hard work on the development of the MobJeX pre-compiler and administration GUI.

References

- Acharya, A., Ranganathan, M., and Saltz, J. (1997). Sumatra: A language for resource-aware mobile programs. *Mobile Object Systems: Towards the Programmable Internet*, pages 111–130.
- Baude, F., Caromel, D., Huet, F., and Vayssière, J. (2000). Communicating mobile active objects in java. In Marian Bubak, Hamideh Afsarmanesh, R. W. and Hetzberger, B., editors, *Proceedings of HPCN Europe 2000*, volume 1823 of *LNCS*, pages 633–643. Springer.
- Baude, F., Caromel, D., Mestre, L., Huet, F., and Vayssière, J. (2002). Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland. IEEE Computer Society.
- Ben-Shaul, I., Cohen, A., Holder, O., and Lavva, B. (1997). Hadas: a network centric framework for interoperability programming. *Proceedings of the Second Cooperative Information Systems, 1997.*, pages 120–129.
- Bloch, J. (2004). *Effective Java Programming Language Guide*. Addison-Wesley.
- Bruneton, E., Lenglet, R., and Coupaye, T. (2002). ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*.
- Campadello, S., Koskimies, O., Raatikainen, K., and Helin, H. (2000). Wireless java rmi. In *Fourth International Enterprise Distributed Object Computing Conference, 2000 (EDOC 2000)*, pages 114–123.
- Caromel, D., Klauser, W., and Vayssiere, J. (1998). Towards seamless computing and metacomputing in java. In Fox, G. C., editor, *Concurrency Practice and Experience*, volume 10, pages 1043–1061. Wiley & Sons, Ltd.
- Carzaniga, A., Picco, G. P., and Vigna, G. (1997). Designing distributed applications with a mobile code paradigm. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, USA.
- Chatzigiannakis, I. and Nikolettseas, S. (2004). Design and analysis of an efficient communication strategy for hierarchical and highly changing ad-hoc mobile networks. *Mobile Networks and Applications*, 9(4):319–332.
- Chiang, C.-C. (2001). Wrapping legacy systems for use in heterogeneous computing environments. *Information and Software Technology*, 43:497–507.
- Dahm, M. (2000). Doorastha: a step towards distribution transparency. *Submitted to JavaGrande*.

- Donald, L. (2004). System modelling and rule evaluation in an adaptive mobile object framework. Unpublished Honours thesis, RMIT University, Melbourne Australia.
- Fowler, R. J. (1986). The complexity of using forwarding addresses for decentralized object finding. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 108–120. ACM Press.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2003). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Haumacher, B. and Philippsen, M. (2001). Exploiting object locality in JavaParty, a distributed computing environment for workstation clusters. In *Proceedings of the Compilers for Parallel Computers (CPC2001)*, pages 83–94.
- Haumacher, M. (2003). JavaParty 1.07g. Available on: <http://www.ipd.ira.uka.de/JavaParty/>. release November, 3rd 2003.
- Holder, O. (1999). Fargo programming guide. Technical Report EE Pub No. 1194.
- Holder, O., Ben-Shaul, I., and Gazit, H. (1999a). Dynamic layout of distributed applications in fargo. *21st International Conference on Software Engineering (ICSE99)*, pages 163–173.
- Holder, O., Ben-Shaul, I., and Gazit, H. (1999b). System support for dynamic layout of distributed applications. In *International Conference on Distributed Computing Systems*, pages 403–411.
- INRIA (2004). Proactive. Available on: <http://www-sop.inria.fr/oasis/ProActive>. release May, 27th 2004.
- Izatt, M., Chan, P., and Brecht, T. (1999). Agents: towards an environment for parallel, distributed and mobile java applications. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 15–24. ACM Press.
- Jing, J., Helal, A. S., and Elmagarmid, A. (1999). Client-server computing in mobile environments. *ACM Comput. Surv.*, 31(2):117–157.
- Lindholm, T. and Yellin, F. (1999). *The Java™ Virtual Machine Specification*. Addison-Wesley, second edition.
- Moyer, S. and Umar, A. (2001). The impact of network convergence on telecommunications software. *IEEE Communications*.
- Perry, S. (2003). Object transport and reference management in the mobjex framework. Honours thesis, RMIT University, Melbourne Australia.
- Philippsen, M., Haumacher, B., and Nester, C. (2000). More efficient serialization and rmi for java. *Concurrency: Practice and Experience 12*, pages 495–518.
- Philippsen, M. and Zenger, M. (1997). JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242.
- Raverdy, P. G., Gong, H. V., and Lea, R. (1998). Dart: A reflective middleware for adaptive applications. In et al., O. L., editor, *Proceedings of the Workshop on Reflective Programming in C++ and Java*, pages 37–45, OOPSLA 98, Vancouver, Canada.

- Recursion Software Inc (2004). Recursion software - voyager home page. Available on: <http://www.recursionsw.com/voyager.htm>. release August, 20th 2004.
- Ryan, C. and Perry, S. (2003). Mobjex: A declaratively configurable java based framework for resource aware object mobility. *On The Move Federated Conferences (OTM '03) Workshops*.
- Ryan, C. and Westhorpe, C. (2004). Application adaptation through transparent and portable object mobility in Java. In *Proceedings of 2004 International Symposium on Distributed Objects and Applications*.
- Shannon, B. (2003). Java 2 Platform, Enterprise Edition (J2EE) : XML Schemas for J2EE Deployment Descriptors. Available on: <http://java.sun.com/xml/ns/j2ee/>. release November, 24th 2003.
- Shapiro, M. (1986). Structure and encapsulation in distributed systems: the Proxy Principle. In *Proceedings 6th International Conference on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA). IEEE.
- Spiegel, A. (2000). Automatic distribution in pangaea. In *Proc. Workshop on Communications-Based Systems (CBS 2000)*.
- Strasser, M. and Rothermel, K. (2000). System mechanisms for partial rollback of mobile agent execution. In *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, page 20. IEEE Computer Society.
- Sun Microsystems (2003). Java remote method invocation - distributed computing for java. Available on: <http://java.sun.com/products/jdk/rmi/reference/whitepapers/javarmi.html>. release May, 31st 2003.
- Weinsberg, Y. and Ben-Shaul, I. (2002). A programming model and system support for disconnected-aware applications on resource-constrained devices. *Proceedings of the 24rd International Conference on Software Engineering, 2002 (ICSE 2002)*, pages 374–384.
- Wheeler, T. (2002). Reducing development effort using the Voyager ORB. Resursion Software White Paper.

A Sample Code Transformation

Simple Object

```
1 public class Domain
2 {
3     public int domainMethod(StringBuffer param1)
4     {
5         // set/return dummy values
6         param1.append("mobjex");
7         return 99;
8     }
9 }
```

Domain Transformation

```
1 public interface Domain
2 {
3     public int domainMethod(StringBuffer param1);
4 }
```

DomainRemote Transformation

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3
4 public interface DomainRemote extends Remote
5 {
6     public Hashtable domainMethod(StringBuffer param1)
7         throws RemoteException;
8 }
```

Implementation Transformation

```
1 public class DomainImpl implements Domain, DomainRemote,
2     Mobile, Serializable
3 {
4     private MobileDelegate mobileDelegate
5         = new MobileDelegate();
6
7     // remote interface to the domain method
8     public Hashtable domainMethodRemote(StringBuffer param1)
9         throws java.rmi.RemoteException
10    {
11        // Hashtable contains changed parameters for return
12        // for simulation of pass by reference semantics
13        Hashtable results = new Hashtable();
14        results.put("param1", param1);
```

```
15
16     // call actual method implementation and
17     // place return value into hashtable
18
19     results.put("mobjexReturnValue",
20     new Integer(domainMethod(param1)));
21     return results;
22 }
23
24 // local/domain interface to the domainMethod
25 public int domainMethod(StringBuffer param1)
26 {
27     // initiate update in proxy if object has moved
28     if(mobileDelegate.isMoved())
29     {
30         throw new MovedException("Object has moved");
31     }
32
33     // set/return dummy values
34     // this comes from the original class
35     param1.append("mobjex");
36     return 99;
37 }
38
39 // call through implementation to MobileDelegate
40 public synchronized void setMoved(boolean moved)
41 {
42     mobileDelegate.setMoved(moved);
43 }
44
45 // further Mobile interface callthroughs omitted for brevity
46 ...
47 }
```

Proxy Transformation

```
1 public class DomainProxy extends AbstractProxy implements
2     Domain
3 {
4     public DomainProxy() { }
5     public int domainMethod(StringBuffer param1)
6     {
7         while(recoverableError == true)
8         {
9             try
10            {
11                // call locally or remote
```

Optimising Transparency, Performance and Reliability in a Mobility Subsystem

```
12         if (local)
13             return ((Domain)object).domainMethod(param1);
14         else
15             {
16                 // call remote implementation and retrieve
17                 // marshalled return and parameter values
18                 java.util.Hashtable results=((DomainRemote)stub)
19                     .domainMethodRemote(param1);
20                 // deep copy results from Hashtable to param1
21                 Duplicator.deepCopy(results.get("param1"),param1);
22                 // unbox the return value
23                 return ((Integer)
24                     results.get("objexReturnValue")).intValue();
25             }
26     }
27     // thrown when previously local target has moved
28     catch (MovedException me)
29     {
30         // update stub/object link
31         updateObjectLocation();
32         return domainMethod(param1);
33     }
34     // thrown when remote target has moved
35     catch (java.rmi.NoSuchObjectException nse)
36     {
37         updateObjectLocation();
38         return domainMethod(param1);
39     }
40     // system level communication error handled by the error chain
41     catch (RemoteException re)
42     {
43         handleError();
44     }
45     catch (DuplicationException de)
46     {
47         throw new ProxyException(
48             "Could not deepCopy object", de);
49     }
50     }
51     throw new ProxyException(
52         "Could not connect to remote object");
53 }
54 }
```

B Complete Test Results

Framework	No of Objects Operation	10				100				1000			
		100 byte	1 kb	10 kb	100 kb	100 byte	1 kb	10 kb	100 kb	100 byte	1 kb	10 kb	100 kb
Mobility Subsystem	Creation	7	13	11	16	20	31	62	148	283	301	465	975
	First Call	0	-	-	-	0	-	-	-	1	-	-	-
	Local Set	0	0	0	0	0	0	0	0	1	0	0	1
	Local Get	0	0	0	0	0	0	0	0	0	1	0	0
	Move Away	23	38	48	143	123	135	206	510	941	832	1196	5423
	First Call	63	-	-	-	358	-	-	-	1918	-	-	-
	Remote Set	7	10	11	50	40	64	61	251	183	211	377	2956
	Remote Get	6	7	14	27	65	44	71	258	211	222	378	2774
	Move Back	45	50	65	168	236	241	286	664	1660	1500	1838	6124
	First Call	57	-	-	-	291	-	-	-	2416	-	-	-
	Local Set	0	0	0	0	0	1	0	0	0	0	0	0
	Local Get	0	0	0	0	0	0	0	0	0	0	0	0
FarGo	Creation	0	16	0	16	0	31	32	78	94	172	281	1875
	First Call	0	-	-	-	31	-	-	-	141	-	-	-
	Local Set	0	0	16	31	16	31	47	172	140	172	579	2547
	Local Get	16	0	0	15	0	0	31	156	141	140	343	1625
	Move Away	47	47	63	93	609	594	797	1235	10406	10250	11578	13485
	First Call	16	-	-	-	140	-	-	-	1422	-	-	-
	Remote Set	15	0	0	47	78	93	125	390	1078	968	1171	6125
	Remote Get	16	16	15	63	79	63	94	360	860	954	1110	4031
	Move Back	78	78	78	141	579	609	813	1234	13375	12828	14984	20531
	First Call	32	-	-	-	281	-	-	-	4656	-	-	-
	Local Set	0	16	15	78	94	110	156	500	1235	1031	1500	4828
	Local Get	15	0	16	78	78	62	125	500	1156	969	1328	5469
JavaParty	Creation	6	76	81	82	37	122	126	246	Could Not Complete			
	First Call	18	-	-	-	78	-	-	-				
	Local Set	5	14	21	58	62	77	95	379				
	Local Get	9	7	11	34	81	65	89	302				
	Move Away	23	14	17	51	62	68	75	168				
	First Call	7	-	-	-	41	-	-	-				
	Remote Set	4	4	6	31	29	37	64	245				
	Remote Get	5	5	6	31	39	44	53	228				
	Move Back	14	19	21	46	90	85	102	341				
	First Call	8	-	-	-	39	-	-	-				
	Local Set	5	6	8	59	39	55	65	349				
	Local Get	8	7	12	39	40	35	64	300				
Voyager	Creation	10	5	8	32	7	21	58	130	Could not complete			
	First Call	15	-	-	-	36	-	-	-				
	Local Set	3	4	6	45	26	24	25	134				
	Local Get	6	3	3	10	30	26	17	51				
	Move Away	179	150	171	198	930	864	852	1211				
	First Call	13	-	-	-	75	-	-	-				
	Remote Set	13	11	18	65	62	72	121	320				
	Remote Get	8	7	11	31	77	60	97	227				
	Move Back	148	120	102	34	700	622	629	250				
	First Call	2	-	-	-	6	-	-	-				
	Local Set	2	2	1	7	5	6	9	96				
	Local Get	1	2	3	6	5	6	9	48				

Figure 14: Remote Get/Set Test - Wired Network Values

Framework	No of Objects Operation	10				100				1000			
		100 byte	1 kb	10 kb	100 kb	100 byte	1 kb	10 kb	100 kb	100 byte	1 kb	10 kb	100 kb
		Mobility Subsystem	Creation	7	14	11	16	20	32	62	148	289	306
	First Call	0	-	-	-	0	-	-	-	1	-	-	-
	Local Set	0	0	0	0	0	0	0	0	1	0	0	2
	Local Get	0	0	0	0	0	0	0	0	0	1	0	0
	Move Away	113	134	376	2276	809	1082	2875	21229	7192	10879	27678	213982
	First Call	218	-	-	-	1952	-	-	-	18240	-	-	-
	Remote Set	35	73	258	2085	317	730	2558	20900	3057	7145	25397	210101
	Remote Get	41	78	258	2111	339	715	2556	21606	3068	7092	25613	211753
	Move Back	216	312	461	2445	1670	2093	4493	23367	15696	19750	42390	231040
	First Call	283	-	-	-	2184	-	-	-	24354	-	-	-
	Local Set	0	0	0	0	0	0	0	0	0	0	0	0
	Local Get	0	0	0	0	0	0	0	0	0	0	0	0
FarGo	Creation	0	15	0	16	16	15	32	78	Could Not Complete			
	First Call	0	-	-	-	32	-	-	-				
	Local Set	0	0	16	31	0	16	46	188				
	Local Get	15	0	0	16	15	16	16	156				
	Move Away	437	437	641	2469	4407	4766	6656	24672				
	First Call	141	-	-	-	1250	-	-	-				
	Remote Set	125	172	359	2203	1250	1734	3578	22203				
	Remote Get	125	156	360	2203	1219	1625	3563	22031				
	Move Back	532	531	734	3016	5765	5547	7203	26656				
	First Call	266	-	-	-	2547	-	-	-				
	Local Set	16	15	16	78	109	110	125	453				
	Local Get	0	16	0	78	62	78	156	547				
JavaParty	Creation	17	265	178	306	170	267	365	404		Could Not Complete		
	First Call	56	-	-	-	275	-	-	-				
	Local Set	19	38	87	607	255	383	1168	9247				
	Local Get	22	30	84	627	319	395	1229	9576				
	Move Away	46	45	148	611	451	584	1358	8458				
	First Call	46	-	-	-	485	-	-	-				
	Remote Set	42	75	233	1900	452	697	2348	19526				
	Remote Get	57	79	260	1991	518	723	2532	20046				
	Move Back	124	128	297	2080	969	1270	2974	20497				
	First Call	7	-	-	-	40	-	-	-				
	Local Set	6	8	14	58	36	40	73	350				
	Local Get	6	6	10	38	54	51	74	322				
Voyager	Creation	9	5	8	33	7	21	58	130	Could Not Complete			
	First Call	15	-	-	-	37	-	-	-				
	Local Set	3	4	6	46	27	24	26	136				
	Local Get	6	3	3	10	30	27	17	52				
	Move Away	324	368	580	3347	2548	2778	5507	33556				
	First Call	45	-	-	-	414	-	-	-				
	Remote Set	47	79	259	2109	408	726	2563	20992				
	Remote Get	50	68	265	2114	525	739	2692	21050				
	Move Back	430	441	706	33	3742	4079	6803	262				
	First Call	4	-	-	-	8	-	-	-				
	Local Set	2	3	4	8	7	7	13	111				
	Local Get	1	2	1	7	6	5	11	46				

Figure 15: Remote Get/Set Test - Wireless Network Values

Crossover

	Mobility Subsystem			FarGo			JavaParty			Voyager		
	10	100	1000	10	100	1000	10	100	1000	10	100	1000
Traverse Links	0	0	0	15	79	1703	4	35	193	8	80	291
Traverse Links	0	0	1	0	93	1688	4	38	169	7	58	223
Stagger	216	498	2091	172	516	6125	3	98	599	321	914	4965
Traverse Links	47	341	1685	32	344	4141	2	119	585	20	134	884
Traverse Links	8	123	766	31	187	3906	4	99	399	19	113	837
Destagger Remote	38	189	1258	62	422	6500	46	73	345	80	531	4006
Traverse Links	61	354	2712	32	375	4828	23	122	630	24	161	1193
Traverse Links	35	326	2429	31	484	5562	9	111	629	27	170	1157
Stagger Back	59	231	1647	109	579	6172	17	130	657	138	529	3509
Traverse Links	51	347	3098	47	656	5657	36	191	1309	12	89	718
Traverse Links	23	172	1453	31	515	5484	30	137	1112	17	88	709
Destagger Local	31	146	1143	47	516	6906	12	116	461	71	471	3357
Traverse Links	33	263	2250	47	563	6860	43	193	1591	8	29	226
Traverse Links	0	0	0	16	468	5234	42	177	1560	6	27	218

Wireless

	Mobility Subsystem			FarGo			JavaParty			Voyager		
	10	100	1000	10	100	1000	10	100	1000	10	100	1000
Traverse Links	0	0	0	0	93	1593	21	175	1809	8	81	297
Traverse Links	0	0	1	16	78	1579	19	173	1749	8	61	226
Stagger	375	1228	8576	812	2563	29390	62	358	3133	361	2202	24370
Traverse Links	137	1284	11540	125	1141	11781	41	404	3510	57	526	7738
Traverse Links	50	430	4099	94	1062	13938	24	392	3375	55	522	7322
Destagger Remote	71	587	5686	188	2250	28937	29	378	2945	132	1323	18524
Traverse Links	740	1587	16224	172	2172	23235	49	537	4625	85	823	12070
Traverse Links	117	1646	15161	171	2078	22234	47	544	4603	85	839	13405
Stagger Back	179	1148	10086	407	3438	35703	65	619	5611	263	2099	43662
Traverse Links	246	2314	24364	281	3078	32625	89	825	9860	43	508	7758
Traverse Links	135	1558	16694	94	1281	14844	80	822	8401	50	507	7276
Destagger Local	135	974	9592	218	2781	31578	49	573	5592	194	2037	41047
Traverse Links	210	2246	24984	172	2266	26688	143	1171	11546	8	28	226
Traverse Links	0	0	0	32	437	7312	126	1124	11336	5	29	219

Figure 16: Remote Stagger Test - Wired and Wireless Values